[illegible]

UNIVERSITY OF MANCHESTER

DEPARTMENT OF COMPUTER SCIENCE

The MU5 Compiler Writers Manual

JULY 1974

Contents

Chapter 1 Introduction and Design Criteria

1.1 General Introduction

This manual describes the MU5 compiler writing facilities. These include the common target language for compilers (CTL), the organisation of MU5 compilers and a syntax processing package. MU5 autocode, the language in which most compilers are written, is not described here but in the MU5 autocode manual.

MU5 autocode does not include any syntax defining facilities. However if a compiler requires a top-down syntax directed scan the syntax processing package described in Ch.2 may be used. This package accepts syntax in a form similar to that accepted by SPG and creates as output an encoded form of the syntax together with a scan procedure. This output is in the form of MU5 autocode statements so may be concantenated with the rest of the compiler, written in autocode, before it is compiled.

Each MU5 compiler is in the form of a library procedure which may be called by a supervisor or other program. The relevant conventions are described in Chapter 3. The compilers also have access to input/output and other standard library material outlined in this chapter and described in detail elsewhere (MU5 Autocode manual, MU5 supervisors manual).

MU5 compilers compile into the common target language (CTL) and not into binary code. This makes the compilers partially machine independent and facilitates their transfer to other MU5 - like machines. Chapters 4,5 and 6 of this manual describe this language. The main computational facility is introduced in Chapter 4. The language is described in full in Ch.5. Ch.6 contains some further examples of the use of CTL.

Chapter 2 The design of MU5 CTL and Autocode. In this section a paper discussing major design decisions concerning the compiler writing system is reproduced.

The MU5 Compiler Target Language and Autocode

P.C.Capon, D.Morris, J.S.Rohl and I.R.Wilson

Department of Computer Science
University of Manchester

Abstract

In this paper the design of the software implementation language for the MU5 machine is considered. This has two representations, the written language, MU5 Autocode, and the parametric compiler target language (CTL).

Introduction

At an early stage in the design of the MU5 software it was decided to introduce a compiler target language (CTL) into which the high level languages would be translated. For each high level language a translator would be provided to convert from the language to CTL while a single compiler converts from CTL to machine code. The objective was to simplify individual translators by forcing the CTL to as high a level as possible. For example, the CTL contains declarations with the characteristics of those found in high level languages so that name and property list management problems are passed to the CTL compiler. This scheme enables the mode of compilation, for example output in semi-compiled form or loading for immediate execution, to be determined within the CTL rather than within each translator.

Subsequently, a further role for the CTL emerged. The MU5 translators could be used on a range of machines provided a CTL compiler could be written for each machine. This machine independence could extend over machines with significant structural differences provided the data and address formats were compatible. This idea is summarised in Fig. 1. It is similar to the UNCOL (Strong, Wegstein Tritter, Olsztyn, Mock and Steel, 1958) idea except that, whereas UNCOL attempted to span the significant differences between existing machines, the CTL has been designed to suit machines originating from MU5. There is, however, a more significant difference: the communication between the translators and the CTL compiler is two-way. Some of the CTL procedures return information to the translators. For example there is a procedure for interrogating property lists. It is this which allows the whole property and name list organisation to be contained within CTL. The CTL does not have to be encoded in character form by the translators then decoded by the CTL compiler. Instead there is a CTL procedure corresponding to each type of statement, so that the CTL is a body of procedures rather than a written language. The main input parameter of each procedure is a vector whose elements define the nature of the statement. In the case of an arithmetic assignment these elements comprise a sequence of operator operand pairs. Only a small increase in compile time results from using the CTL procedures to generate code, because they form part of a natural progression from source to object code.

A loss of run time efficiency could arise from the translators losing the ability to control completely the code which is generated. This problem is largely irrelevant with MU5 because of the high level nature of the order code. For example, the addressable registers serve dedicated functions which correspond to identifiable features of the high level languages. Also the machine dynamically optimises the use of the fast operand store (Ibbett, 1971). If CTL were to be implemented on machines considerably different from MU5 in these respects then some theoretical inefficiency might result. In practice it is difficult to obtain compilers which compile optimum code, so that the inefficiency may be no worse than that already tolerated on many machines.

In the overall software structure the CTL is the instruction set of the MU5 virtual machine (Morris, Detlefsen, Frank and Sweeney,1971). Hence compatibility in the notional MU5 range of machines is at the CTL rather than the order code level. There is an associated written form of CTL, MU5 Autocode, which is the lowest level of programming language and which is used for system programs.

Design Considerations

Two principal decisions have determined the overall characteristics of the CTL and the Autocode. The first of these was that the CTL and the Autocode should be structurally the same language. It is thus possible for the CTL compiler to generate the Autocode equivalent of a program in any source language. A number of minor advantages stem from this ranging from the debugging of compilers to the hand optimisation of important programs. In the light of past experience it was also considered advantageous for the compilers to be written in the same language as they generate.

The second decision was that the CTL and the Autocode should be a high level representation of the MU5 machine code. For example, it will be seen that the declarations relate to physical data items in the machine rather than logical data types. Also the variables are typeless, as are operands in the machine, permitting arbitrary manipulation using any kind of arithmetic. Consequently in MU5 Autocode information about data structures is embedded in the code rather than just in the declarations as in PL/1 or Algol68. However, it is not clear that, on balance, any significant loss of clarity results from this, particularly since operand accessing in MU5 is very flexible. Furthermore efficiency considerations will often dictate that such structures be carefully designed to fit the machine. Additional practical considerations reinforced this decision. Firstly, because the hardware and software of MU5 will be commissioned together, it was considered preferable for the language to reflect the hardware accurately. Secondly, the dependence of the rest of the software on the CTL and the Autocode necessitates a short time scale for their development.

The Autocode representation is the best way of describing the structure of both this and the CTL. An example of a procedure for sorting an array in descending order using linear selection is given in Fig.2. From this the basic language structure should be apparent. In the following sections the form of data, the operations available and the overall control structure are described.

The Autocode computation statements

Each arithmetic computation to be performed requires an implicit or explicit specification of the type and size of arithmetic required. The autocode provides many arithmetic modes but no particular one is considered to be the fundamental mode. It is assumed that only those modes justified by the primary use of a machine are provided in hardware the rest being provided by software. The arithmetic modes are signed and unsigned integer, real and decimal of size 32, 64 or 128 bits and a Boolean mode. In MU5 32-bit signed and unsigned integer, 32- and 64-bit real, and Boolean modes are provided in hardware together with some special functions to aid the software implementation of other modes. The mode is specified at the start of each statement and is following mainly by operator operand pairs. Each of these pairs generally corresponds to a machine instruction; hence the code compiled is closely controlled.

The operator precedence is strictly left to right in contrast to most high-level languages. There are several reasons for this. Firstly, the calculations in systems programs are often of a logical rather than a mathematical nature, and use operators for which precedence rules are not well established. Secondly, it is easier to see that efficient code is being compiled when evaluation is left to right than when implicit stacking of partial results is taking place. Thirdly, since different languages have varying precedence rules an equal precedence convention is the most convenient for use in the target language. Precedence can be forced by the use of bracketed sub-expressions which explicitly demand the stacking of a partial result on the opening bracket, and the application of a reverse operation on the closing bracket. This is shown in the following example of a typical statement equivalent to the Algol

```
E := (A+B)/(C+D)

R64, A + B/(C + D) => E
```

R64 is the 64-bit real mode of calculation; A, B, C, D and E are operands, and / + and => are the divide, add and store operators respectively. In MU5 this statement would translate to:

```
ACC = A          ::set the floating-point accumulator to the values of A
ACC + B          ::add the value of B
ACC *= C         ::stack the partial result and load the value of C
ACC + D          ::add the value of D
ACC Ø STACK      ::reverse divide by the stacked partial result
ACC => E         ::store the result in E
```

Operands and declaratives

The names which are used to represent operands must be declared before use. Thus single pass compilation is possible. The user has close control over the store layout and implicit declarations are not permitted. The scope of the declaratives is organised on a block structure basis. The basic items which may be declared are scalars, vectors and strings. The declaratives specify the operand size which for vector elements, where the widest variation is possible, may be 1, 4, 8, 16, 32, 64 or 128 bits. Scalars are recognised in the machine design and special hardware is provided in MU5 to take advantage of their existence (Ibbett, 1971). Vectors are accessed indirectly by means of descriptors. The descriptor, a stored scalar quantity, specifies the origin, bound and element size for a vector. Vector operands consist of the vector name and a subscript expression of arbitrary complexity. An example of the use of vectors is:

```
R64, X[I x N] + Y[J - 2] => Z
```

In MU5 a modifier register, B, is used to evaluate subscripts so this statement translates to:

```
B = I                ::compute first subscript in B register
B x N
ACC = X[B]           ::load required element of X into accumulator
B = J                ::compute second subscript
B - 2
ACC + Y[B]
ACC => Z
```

The autocode also provides for more complicated data structures such as operands accessed through several levels of descriptors and multidimensional arrays. These cases are always explicitly described rather than following implicit rules. Hence, for example, if X is a vector of vector descriptors, X[I][J] causes element J of the Ith vector to be accessed. Sometimes, the address of a data item, rather than its value, is required. In this case, the built-in function 'ADDR' is used. The use of a simple operand, ADDR, or sub-scripting enables operands to be manipulated in any way required. Also new operands may be created by combining or partitioning existing data structures using the facilities provided for manipulating descriptors.

The allocation of store for these data structures may be dynamic or static. In the latter case store allocation is controlled by declared areas. An example of a static vector declaration is:

```
VEC / $AREA [64, 100] A
```

This declares a vector with 100 64-bit elements numbered 0-99 in the store area AREA. A descriptor of the vector is placed in the local namespace of the current procedure and may be referred to as A.

Autocode control statements

The order of execution of statements in a program is determined by various control statements. These are intended to encompass the corresponding features of standard high-level languages. Apart from readability requirements the possibility of a variety of control hardware in different machines forced these statements to a high level.

A Boolean facility similar to that of Algol 60 is provided. This requirement is catered for at the hardware level in MU5 partly because the cost was small and partly because of local interest in non-numeric programming. The general form of the conditional statement, and the conditional expression, is also similar to that of Algol 60.

A relatively restricted looping facility is provided. Because there are significant structural differences in the various high-level languages it is expected that compilers will usually generate the corresponding conditional statements. The simple facility provided deals with the frequently occurring cases for which special hardware, such as test and count instructions, might exist.

Procedures

A principle design aim of MU5 has been to incorporate the concept of recursive procedures at the hardware level (Kilburn, Morris, Rohl and Sumner, 1969). It therefore follows that the Autocode includes this facility in a form which reflects those in existing high-level languages.

Therefore procedures have static or dynamic namespaces and parameters which are expressions, corresponding to Algol call by value parameters, or descriptors. Descriptor parameters enable reference, substitution, procedure and label parameters to be simply programmed. Procedures which are used as functions yield a result handed back in the accumulator, in which case they may be called in the course of evaluating an expression.

In Fig. 2 it can be seen that a procedure is preceded by a specification. This specification gives the mode of each parameter and of any result yielded by the procedure, while the procedure heading gives only the formal parameter names. Further, the specification must be given before the first call. Thus, the compilation of procedure calls is simplified because the parameters modes are known.

An Example of the parametric form of CTL

An example is now given of the way that this written language is parameterised to form the CTL. Suppose that an Algol translator wishes to translate:

```
x := y + 10
```

where x and y are declared integer.

The corresponding autocode statement is:

```
I32, y + 10 => x
```

The translator must do two things to process this statement:

1. Assemble a parametric form of the statement into a vector.
2. Call the CTL.COMPUTATION procedure with the vector as parameter.

This procedure then generates the corresponding MU5 binary instructions, semi-compiled or other forms.

Suppose that the translator is assembling the parametric form into a vector CODE declared:

```
VEC [32, 21] CODE
```

then the elements of code are assigned as follows:

```
CODE [0] : Computation is in I32 mode and next operand is a name
     [1] : Name y
     [2] : Operator +, next operand is a constant
     [3] : Constant 10
     [4] : Operator =>, next operand is a name
     [5] : Name x
     [6] : Terminating mark
```

The vector therefore contains an operator operand sequence. Each word containing an operator also describes the type of the operand following. The operator is held in the top 16 bits and the operand type in the bottom 16. Considering, in the preceding example, one such element of CODE in more detail, since => is operator 9 and a name is an operand type 16, CODE[4] = % 00090010 in hexadecimal.

A name is replaced at the lexical analysis stage by an internal identifier, an integer, which is handed back to the translator by the CTL.ADD.NAME procedure. Such integers are placed in CODE [1] and CODE [5]. This form of operand assumes the use of the standard form of name and property lists mentioned previously.

CODE [0] which is specially coded to indicate the mode of the sequence can be regarded as describing a load operation. The complete hexadecimal representation of the previous example is:

```
  CODE [0] : %80150010
       [1] : Integer corresponding to y, (internal identifier)
       [2] : %00012001
       [3] : %0000000A
       [4] : %00090010
       [5] : Internal identifier corresponding to x
       [6] : %00320000
```

When this information has been assembled a call:

CTL.COMPUTATION (CODE)

can be made, and the CTL procedure generates the code.

Conclusion

The Autocode and CTL have been implemented in a simulated MU5 system on an ICL 1905E. One translator, for Atlas Autocode is already running in this system. The development of others for Algol, Fortran and PL/1 is well advanced. The MU5 implementation awaits the commissioning of the hardware after which the translators should be transferred without modification.

A compiler for a subset of the Autocode which generates 1900 code is also available. This is being used to develop operating system modules which will also be transferred to MU5. The 1900 code generated is sufficiently good for these modules to be used as part of the 1905E operating system (Morris, Frank, Robinson, Wiles, 1971).

References

Ibbett R. N. (1971) The MU5 Instruction Pipeline. Computer Journal (to appear)

Kilburn T., Morris D., Rohl J. S. and Sumner F. H. (1969) A system design proposal. Information Processing 68, North Holland Publishing Co., Amsterdam pp806-811.

Morris D., Detlefsen G. D., Frank G. R. and Sweeney T. J. (1971) The structure of the MU5 operating system. (submitted with this paper)

Morris D., Frank G.R., Robinson P.H. and Wiles P.R.(1971) The supervisors of the MU5 operating system (to be published).

Strong J., Wegstein J., Tritter A., Olsztyn J., Mock O. and Steel T., (1958) The problem of programming communication with changing machines. CACM Vol.1., No.8.

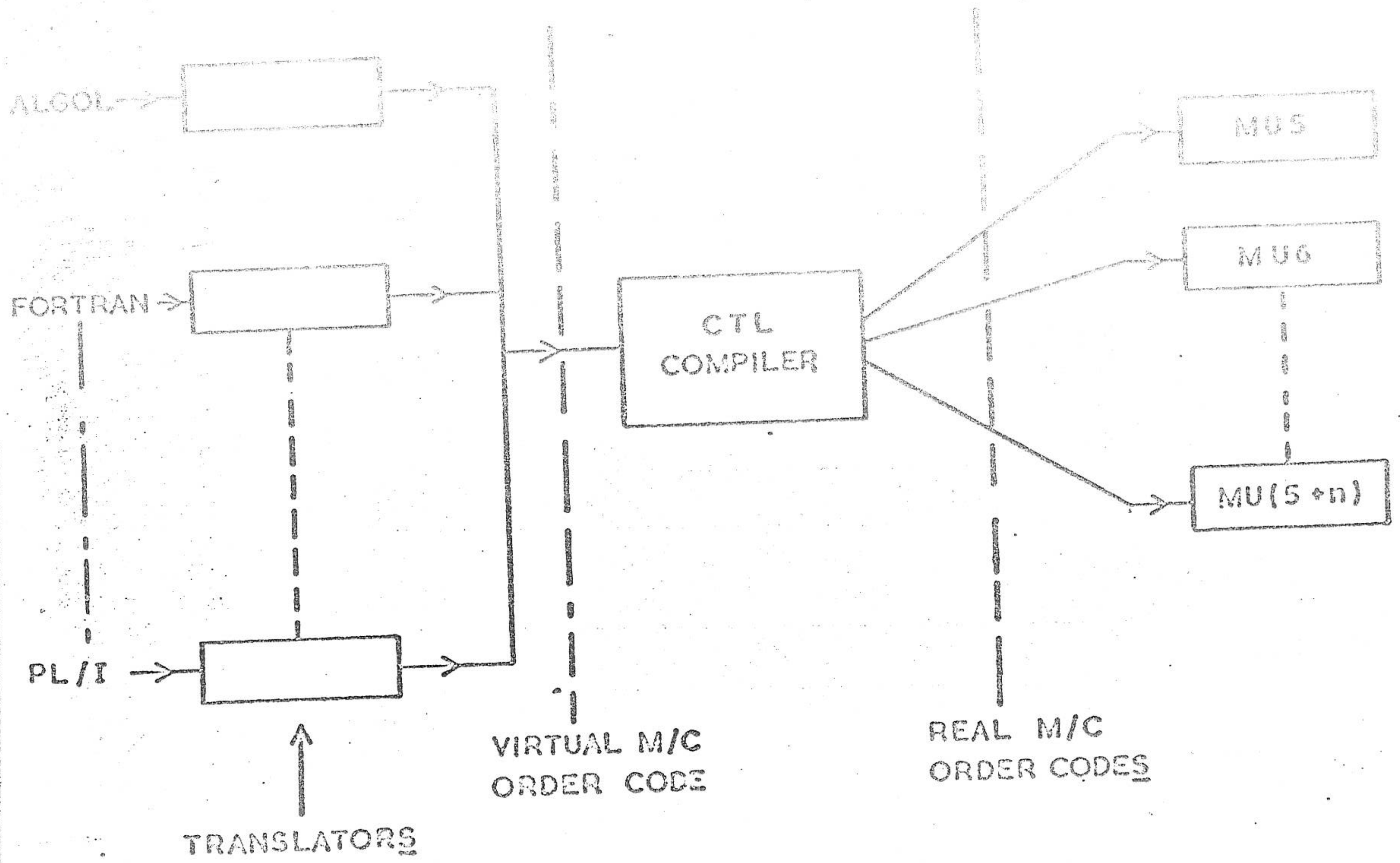

ALGOL
FORTRAN
PL/I
TRANSLATORS
VIRTUAL M/C
ORDER CODE
CTL
COMPILER
REAL M/C
ORDER CODES
MU5
MU6
MU(5+n)

FIG 1

```
PROC SORT(A,N)
PROC SPEC SUB.OF.MAX(S,I32,I32)I32
V32,P,SUB
V64,DUMP
        PROC SUB.OF.MAX(A,P,N)
        V32,SUB,I
        I32,P => SUB
        CYCLE I = P+1,1,N
        IF[R64,A[I]>A[SUB]]THEN
        I32,I => SUB
        CONTINUE
        REPEAT
        RESULT = SUB
        END
CYCLE P = 1,1,N-1
I32,SUB.OF.MAX(A,P,N) => SUB
R64,A[SUB] => DUMP
R64,A[P] => A[SUB]
R64,DUMP => A[P]
REPEAT
RETURN
END
```

Fig. 2.

An example of an MU5 Autocode procedure.

The syntax processing package is now described in the software tools manual.

Pages 14 - 21 are deleted.

Chapter 3 Compiler System Organisation

3.1 Introduction

The compiling system will include implementations of the following languages: FORTRAN, ALGOL, PL/1, AA and MU5 AUTOCODE.It has been designed to fulfil several aims:

1. To avoid a multiplicity of compilers for the various modes of compilation (e.g., load and go, semi-compiled or interactive).
2. To make use of shared procedures for like tasks in different compilers. The existence of only one copy of a particular compiler in the store is a consequence of the store organisation.
3. To facilitate mixed language programming and standardise library conventions.

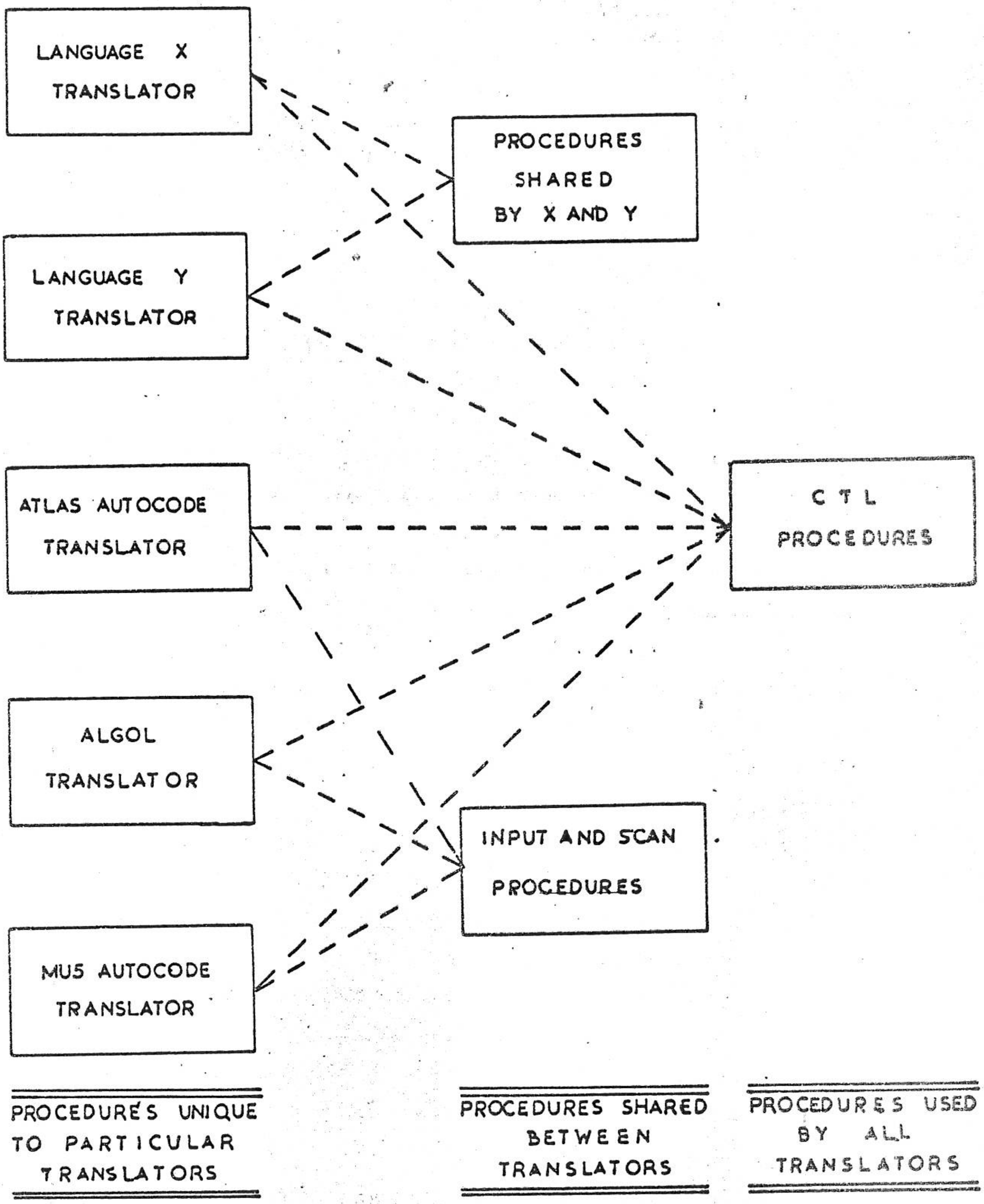
The compilers are pure procedures which reside in the part of the virtual store which is common to all processes (segment numbers > 127). They may be called by any process but in a simple job the compiler will be entered as a result of a statement in the program control sequence. Informatic such as input and output file names and the mode of operation required, is transferred to the compiler by means of parameters. In a simple job these parameters are generated from the program control sequence.

Each translator notionally generates autocode statements from the source text. These statements are in the form of CTL parameters so the translator calls the CTL procedures to output written autocode and/or generate relocatable code. The relocatable code may then be filed and/or loaded into store. The files of autocode or relocatable code generated by this process may subsequently be translated or loaded, respectively.

In the initial implementation the CTL will load for immediate execution and the relocatable form will not be provided.

The CTL procedures are used by all translators. Other procedures, such as input or syntactic analysis may be shared between translators. This is summarised in Fig. 1.

Fig. 1. Organisation of code in the MU5 compiling system


LANGUAGE X TRANSLATOR
LANGUAGE Y TRANSLATOR
ATLAS AUTOCODE TRANSLATOR
ALGOL TRANSLATOR
MU5 AUTOCODE TRANSLATOR
PROCEDURES SHARED BY X AND Y
INPUT AND SCAN PROCEDURES
C T L PROCEDURES
PROCEDURES UNIQUE TO PARTICULAR TRANSLATORS
PROCEDURES SHARED BETWEEN TRANSLATORS
PROCEDURES USED BY ALL TRANSLATORS

3.2 Compiler Parameters

Since one of the design aims is the avoidance of a multiplicity of versions of a compiler the mode of compilation is determined not by the compiler used but by the parameters supplied to the compiler. Some of these parameters are used by the compiler itself while others are just handed on to the CTL procedures. A compiler is called in the same way as any procedure. The statement:

<COMPILER NAME> (P_1, P_2, P_3)

assembles into a sequence to load the parameters P_1 . . P_2 . . P_3 onto the stack and call the compiler. The use of these three parameters is common to all compilers. (P1 and P2 are regarded as I32 quantities).

P1 = compiling mode

P1 = 0 : normal mode and all other parameters take their default settings (This facilitates simple jobs where the compiler is called by the program control sequence. See the MU5-MX/3 Basic Library Manual for further details of program control sequences)

P1 = 1 : normal mode but other parameters do not assume default settings

P1 = 3 : the compiler perm is not present

P1 = 4 : interactive compiling

P1 = 5 : Perm being compiled

P1 = 6 : Perm being extended; some name and property lists already present

P1 = 7 : Perm being extended; no name and property lists present

If the top bit of P1 is set, all high level computation vectors handed to the CTL are printed in hexadecimal on output stream zero.

The next bit (%40(7)) should be set for the initialisation of a two-pass compiler.

Bit %20(7) should be set when compiling a common procedure.

P2 = running mode

P2 = 0 normal

Various bits can be set in P2 as follows:

Least significant bit(0) = 1 production run (This may inhibit certain checking)

bit 1 = 1 full overflow check

2 = inhibit array bound check

3 = full array bound check

The top bit gives the number of segments required for dynamic vectors if this is greater than one.

Other bits so far unallocated.

P3 = Library directive. If P3 = 0 then only the system library and compiler perm are required. Otherwise P3 is a vector descriptor. Each element of the vector will give the file number of a library file to be scanned. Initially only the system library is provided.

Program and data listing with or without conventional page and line numbers is organised by the input device controllers. At the end of compiling a compiler should exit (usually to the program control sequence). The program may then be separately filed or entered by other program control sequence statements.

If the compiler input comes from a stream other than the current input a program control sequence statement is used to switch compiling to that stream. If compiler monitoring is to be output on a different stream this stream is selected before the compiler is entered. Other special effects can be achieved by calling CTL.INITIALISE (with a ** command) before the compiler is entered.

3.3 Library procedures

An arbitrarily large collection of library procedures will eventually be allowed within the system library or private libraries. These will usually be held in source form and need not be completely self contained. In the initial implementation only COMMON library procedures are implemented. These are held in fully compiled form and must be completely self contained. A self contained procedure has no non-local variables except those corresponding to absolute names which are conventionally in fixed positions in store (e.g., current input numbers). Such a procedure has no non-local procedure calls except to other COMMON procedures. If subprocedures exist they are concealed and do not appear as separate procedures to a calling program.

User programs may access COMMON procedures by a COMMON directory attached to this set of procedures. Compilers are organised so that all common procedures are automatically available to users. There will be a supervisor called SYSTEM LOAD to add new procedures to the COMMON library, and load them into the common part of the virtual store. For full details see the MU5-MX/3 library manual.

3.4 CTL Areas

The present 'load and go' implementation of CTL creates segments for static areas at compile time. Datavectors may be put into these areas. Future implementations may postpone the creation of segments until run-time.

Normal programs should only use a minimum of such segments, hence:-

(1) Datavectors should normally be read only and in the code segment. [Area identifier Zero in CTL.DATAVEC places the datavector in the code segment].

(2) CTL will create the Algol/AA dynamic vector segment at run-time.

3.5 Filing Programs

A general purpose filing mechanism is defined. Filing a program means compacting all areas associated with it into a single segment. This is done by the library procedure DEFINE. A library procedure LOAD will recreate all the segments which existed at DEFINE time and unpack the areas into these segments. The program may then be entered using ENTER. The library procedure CALL consists of LOAD followed by ENTER. [For the specifications of these procedures see MU5 - MX3 Library Manual].

The format of a filed program segment is:-

```
Jump to 1st instruction (planted by CTL)
   control information  (planted by CTL and DEFINE)

   code                 (planted by compiler using CTL)

   control information  (planted by DEFINE)

   initialised areas of other segments  (moved here by DEFINE)
        (including compile time names and properties and the
         linetable, if required)
```

Programs are suitable for filing only if their compile and run addresses, as specified to CTL.INITIALISE, are at the start of a segment.

3.6 Perm Definition

The Perm code, compiled in any of the modes 5, 6, 7 with compile address the perm segment, should be defined as a file by DEFINE. The name/property lists are added or not depending on the setting of the second parameter to DEFINE. When CTL.INITIALISE is called in a mode in which perm name/property lists are present (i.e., compiling modes 0, 1, 6) these are read to form the initial name/property lists for the program. For compiling modes 6, 7 all new code compiled follows the Perm code already defined.

Chapter 4 Introduction to the Common Target Language Computational Statements

The common target language is the target language of all the high level language translators. It is not a written language, but consists of a set of procedures. The parameters for a call of one such procedure comprise a 'statement' in CTL.

The highest level of CTL is a parametric form of the MU5 Autocode. This is the form used by MU5 translators. More basic declarative and operand specification is provided for range compatibility.

There is a CTL procedure or group of procedures corresponding to each MU5 autocode statement. The remainder of this chapter introduces the CTL equivalent of the Autocode computational statement from the point of view of the MU5 Translator which is using Autocode as a conceptual target language. The following chapter contains a detailed summary of the computational facilities. It also deals with control and declarative facilities.

Suppose that an Algol translator wishes to compile a translation of:-

```
x := y + 10
```

where x and y are declared real.

The corresponding autocode statement is:-

```
R64, y + 10 => x
```

The translator must do two things to process this statement:-

1) Assemble a parametric form of the statement into a vector.

2) Call the CTL COMPUTATION procedure with the vector as parameter. This procedure then generates the corresponding code.

Suppose that the translator is assembling the parametric form into a vector CODE declared:-

VEC [32, 20] CODE

then the elements of code are used as follows:-

CODE [0]:	Computation is in R64 mode and next operand is a name
[1]:	Name y
[2]:	Operator + and next operand is a constant
[3]:	Constant 10
[4]:	Operator => and next operand is a name
[5]:	Name x
[6]:	Terminating mark

The vector therefore contains an operator operand sequence. Each word containing an operator also describes the type of the operand following. The operator is held in the top 16 bits and the operand type in the bottom 16.

Example:-

=> is operator 9

A name is an operand of type 16 so CODE [4] = %00090010 in hexadecimal.

A name is replaced at the itemise stage by an internal identifier, and integer handed back to the translator by the CTL ADD NAME procedure. Such integers are placed in CODE [1] and CODE [5].

CODE [0] can be regarded as describing a load operation. This word must be explicitly present in CTL, there is no implicit mode. It is coded in a special way described in the operator tables in section 2. Reference to these tables shows that the hexadecimal representation of the previous example is:-

```
CODE [0]:   %80260010
     [1]:   Integer corresponding to y
     [2]:   %0001100E
     [3]:   Pointer to 64 bits containing floating point 10  ?↓
     [4]:   %00090010
     [5]:   Integer corresponding to x
     [6]:   %00360000
```

(Note floating point 10 could be held in CODE[3] and CODE[4] Elements 4, 5, 6 would then become 5, 6, 7. Element 2 would be %0001000E.) ≡

When this information has been assembled a call:-

CTL.COMPUTATION (CODE)

can be made.

A general computation sequence will contain subexpressions particularly since there is no operator precedence.
e.g.,

I32, A + (B * C) => D

These subexpressions break up the usual flow of operators and operands. Using the vector CODE again this becomes:-

CODE [0]:	I32 Mode, named operand following
[1]:	Name A
[2]:	+, bracketed subexpression follows (special operand type)
[3]:	Dummy, named operand following
[4]:	Name B
[5]:	*, named operand following
[6]:	Name C
[7]:	), another operator follows (special operand type)
[8]:	=>, named operand following
[9]:	Name D
[10]:	Terminating mark

It can be seen that special operand types (in fact group 2 operands) were used in CODE [2] and CODE [7] to indicate that another operator word followed immediately and that the usual sequence was interrupted.

Subexpressions are sometimes of different modes, e.g.,

I32, A + [I64, B * C] => D

The implicit conversion is carried over to the CTL.

Using CODE again:-

CODE [0]:	I32 mode, name follows
[1]:	Namo A
[2]:	+, [ ] subexpression follows (special operand type)
[3]:	I64 mode, name follows
[4]:	Name B
[5]:	*, name follows
[6]:	Name C
[7]:	], another operator follows
[8]:	=>, name follows
[9]:	Name D
[10]:	Terminating mark

After the introduction of another special operand type at CODE [2] the subexpression follows exactly the same form as the main computation.

In both sorts of subexpression the bracketing must match up correctly, otherwise a fault will be signalled.

Some of the operands in a typical statement will involve vector and array accesses. There are separate pseudo-operators for each case, e.g.,

I32, A[J] => D

where A is a vector.

```
    CODE [0]:    I32, name follows
         [1]:    Name A
         [2]:    [ subscripting, name follows
         [3]:    Name J
         [4]:    ] subscripting, operator follows
         [5]:    =>, name follows
         [6]:    Name D
         [7]:    Terminator
```

For several stages of indirection:-

I32, A[I][J]

becomes

```
    CODE [0]:    I32, name follows
         [1]:    Name A
         [2]:    [ subscripting, name follows
         [3]:    Name I
         [4]:    ][ continue indirection, name follows
         [5]:    Name J
         [6]:    ] subscripting, operator follows
         [7]:    Terminator
```

Use of the [subscripting and [dope vector operators automatically implies that the following expression is evaluated in modifier mode whose effect is identical to I32. If this is not required a mode word must follow the [operator thus introducing a subexpression of different type.
e.g.,

```
                    I32, A[[R64, B + C]]

      CODE [0]:     I32, name follows
           [1]:     Name A
           [2]:     [ subscripting, [] subexpression follows
           [3]:     R64 mode, name follows
           [4]:     Name B
           [5]:     +, name follows
           [6]:     Name C
           [7]:     ] subexpression, another operator follows
           [8]:     ] subscripting,  another operator follows
           [9]:     Terminator
```

Procedure calls are another case of complicated operands, here special () operators are provided for the parameters. If the parameter mode has been specified in the procedure specification then this mode will be used to evaluate the parameter. (If not the current mode or mode specified is used). e.g.,

```
                    R64, P(A, B) => C

      CODE [0]:     R64, name follows
           [1]:     Name P
           [2]:     ( parameter, name follows
           [3]:     Name A
           [4]:     , parameter, name follows
           [5]:     Name B
           [6]:     ) parameter, another operator follows
           [7]:     =>, name follows
           [8]:     Name C
           [9]:     Terminator
```

One of the possible modes of computation is the Boolean mode. The arithmetic operators are not applicable in this mode but &, V, ≠ are as well as two additional operators, equivalence and not. e.g.,

B, A NOT & C => D

```
CODE [0]:   Boolean, name follows
     [1]:   Name A
     [2]:   NOT, operator follows
     [3]:   &, name follows
     [4]:   Name C
     [5]:   =>, name follows
     [6]:   Name D
     [7]:   Terminator
```

Conditional expressions and subexpressions may be used. They have the same power as the Autocode facility but there are slight stylistic differences. A distinction is made between conditions involving just a relation and those involving a Boolean expression.

```
R64, IF [A = B] THEN X + Y => Z ELSE B => A

    CODE [0]:    R64, condition follows
         [1]:    Relation, mode word follows
         [2]:    I32, name follows
         [3]:    Name A
         [4]:    Compare, name follows
         [5]:    Name B
         [6]:    IF ≠, displacement
         [7]:    9 (relative pointer)
         [8]:    Use hanging operator, name follows
         [9]:    Name X
         [10]:   +, name follows
         [11]:   Name Y
         [12]:   =>, name follows
         [13]:   Name Z
         [14]:   SKIP, displacement
         [15]:   5
         [16]:   Use hanging operator, name follows ←
         [17]:   Name B
         [18]:   =>, name follows
         [19]:   Name A
         [20]:   Terminator ←
```

The IF ≠ operator terminates the relation and hence the evaluation in I32 mode. The condition (=) is reversed and delayed until the point when the test takes place. The point in the vector to which control is transferred is specified explicitly by a relative pointer. A similar explicit SKIP replaces the ELSE 'operator'. The dummy operator at code [8] and [16] is a special operator (operator 37) meaning apply the hanging operator which in this example is a load; in the next example it is +.

R64, A + (IF [P = Q] THEN A ELSE B) => C

```
CODE [0]:   R64, name follows
     [1]:   Name A
     [2]:   +, condition follows
     [3]:   Relation, mode word follows
     [4]:   I32, name follows
     [5]:   Name P
     [6]:   Compare, name follows
     [7]:   Name Q
     [8]:   IF ≠, displacement
     [9]:   5 ------------------------------------------.
     [10]:  Use hanging operator, name follows          |
     [11]:  Name A                                      |
     [12]:  SKIP, displacement                          |
     [13]:  3 -------------------------------------.    |
     [14]:  Use hanging operator, name follows <---|----'
     [15]:  Name B                                 |
     [16]:  =>, name follows <---------------------'
     [17]:  Name C
     [18]:  Terminator
```

Note that the brackets which are needed to delimit the conditional expression in Autocode are redundant in CTL as the SKIP operator transfers control to the correct point.

R64, IF P & Q THEN A => B ELSE C => D

P and Q are Boolean and a Boolean expression is involved in this computation.

```
CODE [0]:   R64, condition follows
     [1]:   COMPLEX CONDITION, name follows
     [2]:   Name P
     [3]:   &, name follows
     [4]:   Name Q
     [5]:   IF FALSE, displacement
     [6]:   7 ─────────────────────────────────┐
     [7]:   Use hanging operator, name follows │
     [8]:   Name A                             │
     [9]:   =>, name follows                   │
     [10]:  Name B                             │
     [11]:  SKIP, displacement                 │
     [12]:  5 ──────────────────────────────┐  │
     [13]:  Use hanging operator, name follows ←┘
     [14]:  Name C                          │
     [15]:  =>, name follows                │
     [16]:  Name D                          │
     [17]:  Terminator ←────────────────────┘
```

Chapter 5 Definition of CTL

5.1 Introduction

In this chapter each facility of CTL is described in detail.

5.2 Computation

This section is subdivided as follows:

(a) Introduction

The specification of the procedure which processes computational statements is:

PROC SPEC CTL.COMPUTATION (S)

The parameter is the descriptor of a vector with 32-bit elements. This vector contains the sequence of operators, operand types and actual operands defining the computation. The existence of three notional registers is assumed. These are an accumulator of the correct arithmetic type and size, a B register for subscript evaluations, and a D register for accessing secondary operands. A user may ignore these registers or make explicit reference to them according to standard rules. The initial 'operator', a load, specifies the mode of computation and whether the use of the B and D and ACC registers is to follow standard rules during the computation. The subsequent operands must then be of the specified size but do not have a type of arithmetic associated with them.

Each 32-bit word containing an operator (in its top half) also contains a specification of the type of the operand following. The operand itself, or a pointer to it, usually occupies the next 32-bit word:-

```
<   32 BITS      > <  32    > <          32         > <   32   >
|       OPERAND   |          |              OPERAND  |          |
|MODE |  TYPE     | OPERAND  |OPERATOR |    TYPE     | OPERAND  |
```

Occasionally there are two consecutive operators so that the operator operand sequence is broken. Special operand types indicate this. It occurs at the start and end of subexpressions. Subexpressions involve stacking the result so far when necessary, evaluating the subexpression, applying a mode conversion if the subexpression is of differing mode, and then combining with the stacked quantity using a reverse operator. The unary operators CONDITION and NOT also result in two consecutive operators.

A few of the operands are more than 32 bits long. In this case either the operand itself is pointed to or several operand words follow one other. For details see operand types.

The operator words are divided thus:-

```
< 16 BITS > <   16 BITS      >
| OPERATOR | OPERAND TYPE    |
0        15 16              31
```

(b) The Modes

The 16 operator bits of the initial or mode word are coded as follows:-

(Top bit)

Bit 0 = 1	to distinguish from other operators.
Bit 1 = 1	if the user permits CTL to optimise the loading of registers. In general he can allow this except when addressing an operand in two different ways within a sequence. This bit also enables the CTL to remove any register loading implied by bits 2, 3 and 4 which is redundant.
Bit 2 = 1	if standard use is to be made of the accumulator ACC.
Bit 3 = 1	if standard use is to be made of the B register.
Bit 4 = 1	if standard use is to be made of the D register.

In the description of the use of the special operands ACC, B and D later in this section the effect of these bits is fully explained.

Bits 5-7 Spare.

Bits 8-15 are used for mode information as follows:-

Bit 8 Tag used to inhibit printing in fault monitor.

Bits 9-12	TYPE:	1 =	Boolean	(size 1 bit only)
		2 =	Fixed signed	
		3 =	Fixed unsigned	
		4 =	Floating point	
		5 =	Decimal	
		6 =	Subscript	(size 32)
		7 =	Descriptor	
		8 =	Label	
		9 =	Procedure	
		10 =	Source	
		11 =	Dest	
		12 =	Complex	(size 64 or 128 bits) DR 7, 6
		13 =	Store to store sequence	(size 8 bits) 0111, 110
		14 =	Long arithmetic	3 E
Bits 13-15 (bottom bits)	SIZE:	0 =	1 bit	(Boolean type only)
		3 =	8 bits	
		5 =	32 bits	
		6 =	64 bits	
		7 =	128 bits	(Floating point, decimal and complex only)

This coding of type and size is used whenever such information is needed, not just in computations.

(c) Table of operators

For other operators the form is:-

| 0 | OPERATOR NO. |

Hexa-decimal	Decimal		
%0	0	DUMMY	NO ORDER COMPILED
1	1	+	
2	2	-	
3	3	*	
4	4	/	
5	5	⊖	REVERSE -
6	6	ø	REVERSE DIVIDE
7	7	A	SCALE } BY I32 QUANTITY
8	8		CIRCULAR SHIFT } BY I32 QUANTITY
9	9	=>	STORE
A	10	V	
B	11	&	
C	12	≢	
D	13	≡	(BOOLEAN MODE ONLY)
E	14	¬	NOT (BOOLEAN MODE ONLY)
F	15	D+	YIELDS DOUBLE LENGTH RESULT
10	16	D-	YIELDS DOUBLE LENGTH RESULT
11	17	D*	YIELDS DOUBLE LENGTH RESULT
12	18	H+	operands SINGLE/HALF, single length result
13	19	H-	
14	20	R/	REMAINDER STACKED
15	21		Rounded integer and decimal divide
16	22	A	EXPONENTIATION BY I32 QUANTITY
17	23	$	UNARY
18	24	$$	FURTHER UNARY
19	25		CONDITION INVOLVING BOOLEAN OPERANDS
1A	26		SIMPLE RELATION (FOLLOWED BY MODE WORD)
1B	27		COMPARE
1C	28	IF =	TEST AND JUMP TO OPERAND OR SET BOOLEAN
%1D	29	IF ≠	

%1E	30	IF <	
1F	31	IF ≤	
20	32	IF >	
21	33	IF ≥	
22	34	IF	TRUE (TEST BOOLEAN RESULT)
23	35	IF	FALSE (TEST BOOLEAN RESULT)
24	36	SKIP	JUMP TO OPERAND
25	37		USE HANGING OPERATOR (AFTER CONDITION)
26	38	)	END OF SUB EXPRESSION
27	39	]	END OF SUB EXPRESSION OF DIFERENT MODE
28	40	[	OPEN DOPE VECTOR OPERATOR
29	41	,	SUBSCRIPT OPENING (DOPE VECTOR)
2A	42	,	SUBSCRIPT SEPARATOR (DOPE VECTOR ACCESS)
2B	43	]	DOPE VECTOR CLOSING OPERATOR
2C	44	(	PARAMETER OPENING
2D	45	,	PARAMETER SEPARATOR
2E	46	)	PARAMETER CLOSING
2F	47	[	OPEN INDIRECTION SEQUENCE (SUBSCRIPTING)
30	48	][	CONTINUE INDIRECTION
31	49	]	CLOSE INDIRECTION SEQUENCE
32	50	INDIRECTION	SEVERAL LEVELS OF INDIRECTION
33	51	HESITATE	MU5 Z CODE
34	52	SWOP	
35	53	*=	STACK AND LOAD
36	54	TERMINATOR	

<u>Notes</u>

1) A computation vector must have a correctly matched bracketing structure for subexpressions, parameters, subscripts.

2) The Boolean mode of computation has a restricted set of operations. Boolean expressions are completely evaluated. The operators V & ≠ ≡ ¬ =>, conditional operators and operand accessing 'operators' are permitted in Boolean mode.

3) The initial operator, 'load', may be used anywhere in a computation not just at the start or after [. The only sensible use of the facility is at the top level of expression where it effectively allows several computations to be concatenated within a single call on the CTL.

(d) Description of the Operators

0	dummy	dummy operators and operands may be placed anywhere, except in composite operands, in a computation and have no effect. The dummy operator is used after (.

Simple Arithmetic

1	+	Arithmetic add of following operand to notional accumulator.
2	-	Subtract operand from accumulator.
3	*	Multiply accumulator by operand.
4	/	Divide by operand. The quotient is truncated towards zero in fixed and decimal mode and rounded in floating mode. Not available in unsigned mode.
5	θ	Reverse subtract accumulator from operand.
6	ø	Reverse divide. Divide operand by accumulator. The quotient is truncated towards zero in fixed and decimal mode and rounded in floating mode. Not available in unsigned mode.

None of these operators are available in Boolean mode.

Double Length Arithmetic

15	D+	Add operand to produce double length result.
16	D-	Subtract operand to produce double length result.
17	D*	Multiply by operand to produce double length result. The arithmetic mode before the operation and the operand must be 32-bit or 64-bit. After the operation the mode is automatically 64-bit or 128-bit respectively. Not available in 128-bit or Boolean mode. D+ and D- are not available in floating point mode.
18	H+	The operand is half the length of the accumulator and added to it.
19	H-	The half length operand is subtracted from the accumulator. These operations, only available in 64-bit and 128-bit mode, enable the accumulation of double length results. Not available in floating point mode.

Operators 15-20 inclusive cannot be used with operands which involve implicit or explicit stacking.

Remainder Divide

20 R/ The remainder is stacked so can be accessed by the operand 'stack'. It satisfies the rule:-

dividend = quotient * divider + remainder

The quotient is truncated towards zero and remains in the accumulator. Not available in unsigned or Boolean mode.

Rounded Division

21 ROUNDED DIVISION Fixed point and decimal modes only. The quotient is rounded to the nearest integer, instead of being truncated towards zero.

Scaling and Shifting

7 SCALE The operand must be an I32 variable represented n. Scale has the following effect:-

I mode	:	multiply by 2^n (n signed)
U mode	:	shift logically n places left (negative n implies a right shift)
R	:	multiply by 2^n
D	:	multiply by 10^n

8 CIRCULAR SHIFT

The operand must be an I32 variable represented n. Circular shift then shifts the bits of the accumulator n places left circularly (negative n implies right shift). The number of bits shifted depends on the accumulator size, 32 or 64 bits, but not on type. Not available in 128-bit modes.

Store

9 => Stores the current accumulator value at the operand specified while preserving the accumulator value.

Logical Operators

10	V	Logical OR.
11	&	Logical AND
12	≢	Logical non-equivalence. The operation is performed bit by bit between the current accumulator and the operand regardless of accumulator type.

Boolean Operators

13	≡	Equivalence is available in Boolean mode only.
14	NOT	Not is a Boolean postfix unary operator which qualifies the previous result. It does not require any operand so will usually be followed by another operator.

Exponentiation

22 Exponentiate

The exponentiation operator performs exponentiation by repeated multiplication. The following operand must be I32. A run time fault is generated by any attempt at 0 ⭡ 0, and for a negative exponent in integer arithmetic.

Conversion Operators

23	$	Convert accumulator.
24	$$	Mode change.

There are two unary operators $ and $$ which are used for mode conversion. Each is followed immediately by a conversion operand in the operand type bits with the same format as a mode word giving the mode after conversion. (Another operator word therefore follows immediately). $ performs conversion of the result so far while $$ indicates a mode change without conversion. Conversion between Boolean and arithmetic modes is illegal. Conversion is also achieved by a subexpression of different type as in MU5 autocode. Note that a subexpression must yield a result of the type specified by the initial operator so that although $ and $$ may be used inside subexpressions a single use of one of these operators in a subexpression is not sensible. Most conversions are straightforward but for real to integer conversions and when shortening a floating quantity (e.g. R64 to R32) the result is rounded to the nearest integer (by adding 0.5) unless bit 5 was set in the mode, in which case the result is truncated towards zero.

Miscellaneous

51 Hesitate

The Hesitate operator is a unary operator with no effect on the accumulator. On MU5 it delays the next instruction so that any interrupts resulting from the previous instruction occur before the next instruction is obeyed.

52 Swop

SWOP is a unary operator which may be followed by another operator. It exchanges the current accumulator and the last stacked value.

54 Terminate

A complete sequence must end with a terminator. Any information in the vector beyond the terminator is ignored.

38)

End of sub-expression. A subexpression which is introduced by a special pseudo-operand type is terminated by this operator.

39]

End of sub-expression of different mode. The special pseudo-operand [introduces sub-expressions of different mode which are terminated by this operator. The conversion applied is as for \$ (operator 23).

(e) Complex arithmetic

Complex arithmetic is performed on two 64 bit floating point quantities, the real part and the imaginary part (128 bit complex) (and also two 32 bit floating point quantities i.e. 64 bit complex).

The operands of 128-bit complex are regarded by the CTL as 128 bit operands. Complex vectors are described by descriptors with size bits 128.

The usual CTL operators 0-12, 22-24 and operand accessing operators are provided in complex mode.

(f) Subscript mode

It is possible to specify the mode of computation as subscript mode. This is identical to I32 mode except that, firstly, when standard use if made of the registers the result is in B and the accumulator is preserved. Secondly, a dope vector subscript calculation commencing with operator follows operand type and the [(dope vector) operator and terminating with the corresponding] is permitted as an operand. This type of sequence will normally be used when pre-evaluating a complex subscript.

Examples

(i) Subscript, A + B * C

(ii) Subscript, [A, I, J]

Descriptor Sequences

Described in section 5.3.

(g) Conditional Operators

A simple relation is of the form:-

(i)	Relation operator (26)
(ii)	Mode word
(iii)	Arithmetic sequence
(iv)	Compare operator (27)
(v)	Arithmetic sequence
(vi)	Testing operator (28 - 33 inclusive)

Operator 27 may only appear following operator 26 and 28-33 must follow operator 27.

The testing operator must be followed by a displacement operand (type 1), giving the relative displacement of the point in the computation vector to which control is to be transferred, unless the relation is used within a Boolean expression.

Each test operator must lead to an occurrence of operator 37 which is null and means apply the hanging operator which occurred before the IF operator.

The skip operator (36) can only occur after a testing operator and the following arithmetic sequence. It is used to implement the ELSE expression of MU5 autocode.

A complex condition commences with the condition operator (25) which causes a switch to Boolean mode and terminates with an IF TRUE (34) or IF FALSE (35) operator. This operator must be followed by a displacement operand.

Such conditions may involve relations as sub-conditions. In this case no displacement follows the relation testing operator (28-33) but instead the special operand 37.

Examples of the use of these facilities were given in Chapter 4.

The Composite Operand Pseudo-Operators

There are 3 types of composite operand involving pseudo operators. These are:-

(1) Indirect operands

(2) Array operands involving use of dope vectors

(3) Procedure call operands

These types may occur together as described below.

In all cases the primary operand of a composite operand (the initial descriptor or procedure name) must be marked with the composite operand bit.

(1) In all indirect operands the initial operand is treated as a descriptor which may then be modified and which is then used to access an item. This item may be the data item itself or used in a further stage of indirection.

The initial operand (which is usually simple but which could possibly be a composite operand of types 2 or 3) is followed by the [indirection operator (47). This operator is followed by operator follows and] if no modification is required. Otherwise it introduces a subscript mode sequence of arbitrary complexity. (This might involve nested composite operands of any type). No load operator is required since the [implies a change to subscript mode. The sequence is terminated by a][continue indirection (48) or a] close indirection (49) operator. In the former case a further null operand or subscript sequence follows for a second level of indirection. Indirection may continue to any level. In the latter case the indirection is complete and the operand is fully described unless it forms part of a dope vector or procedure call operand (see below).

If the indirection is unmodified at every level then the indirection (50) operator provides a shorter method of specification. This operator follows the initial operand (instead of [) and is followed by a 32-bit integer literal which specifies the number of levels of indirection required. This literal must be positive or zero. If it is zero it is interpreted in a special way as a form of automatic indirection. The indirection continues as long as type 2 descriptors are found. As soon as another type of descriptor occurs this is treated as describing the data item required.

Note that descriptors of type 3.3 cause automatic indirection until type 3.3 is no longer present as a result of a single indirection operation.

Examples

(i) A [I][J]

Take the descriptor A and modifier I to access a descriptor. Use the descriptor with modifier J to access a data item.

(ii) A INDIRECT 1

A points to the operand.

(iii) A INDIRECT 3

A points to a descriptor which points to a descriptor which points to the item accessed.

(iv) A[null][null][null][B[J]]

In this example there are 3 unmodified indirections followed by a modified indirection the modifier itself involves an indirect access. In this particular case the number of orders obeyed would usually be reduced by evaluating B[J] beforehand in a separate sequence. However, the following example should result in optimal code:-

(v) A [B[J]]

(2) Dope vector accesses conform to the following pattern:-

(a) Initial operand, which gives the descriptor of the array being accessed. This operand may be simple or of any of the composite types.

(b) [operator (40).

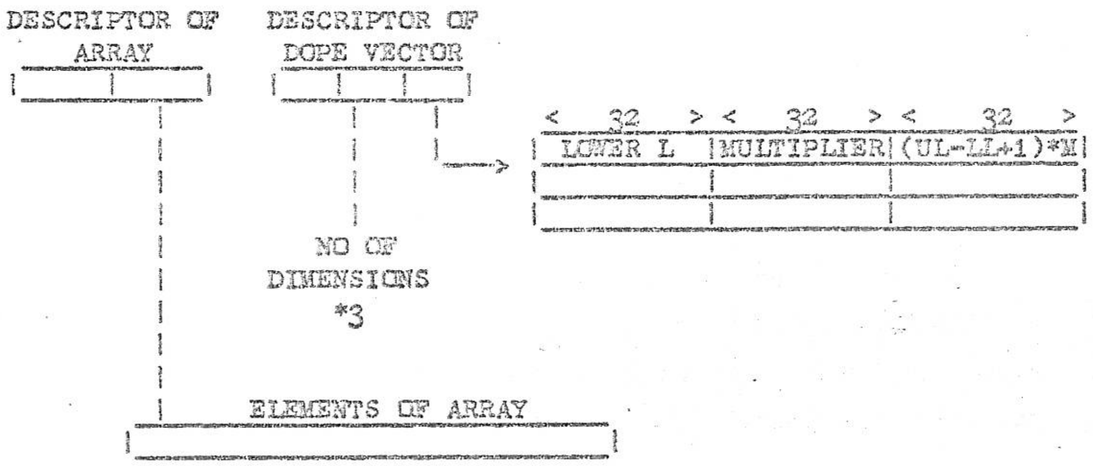
(c) Operand which yields the descriptor of the dope vector of the array being accessed. The form of the dope vector is described below. This operand may be simple or of any of the composite types.

(d) , operator (41). This operator implies a subscript mode sequence following.

(e) Sequence of subscript mode sequences separated by , operators (42). These sequences may be arbitrarily complicated.

(f)] operator (43). This operator terminates the dope vector construction.

The dope vector facility is described diagramatically thus:-



The dope vector contains 3 elements for each subscript. The first is a lower limit to be subtracted from the subscript. The second is a multiplier by which the subscript is then multiplied. The third is a limit against which the resulting value is tested for possible bound check failure.

Examples

In these examples the indirection operators are represented ⁅ and ⁆ to avoid ambiguity. The operands tagged composite are marked with a prime.

(i) A' [B, I, J]

(ii) A'⁅I⁆ [B, J, K, L]

(iii) A' [B'⁅J⁆, C'⁅I⁆, K]

(iv) A' [B, I, J] [C, K, L]
In this example the initial operand is accessed by the dope vector mechanism.

(v) A' [B, I, J]⁅K⁆⁅L⁆
In the example a dope vector access is the initial operand of an indirection sequence.

(vi) A' [B, I, J] [C' [D, I, J], E'⁅K⁆, L]
In this example the array descriptor and the dope vector descriptor are both obtained by dope vector accesses.

(3) Procedure call operands. A call of a functional procedure (i.e., a procedure which yields a result) with or without parameters is a composite operand of the form:-

(a) Procedure name operand.

(b) (operator (44). This must be present even if there are no parameters.

(c) Computation sequences for each parameter separated by , operator (45) (distinct from the , used for dope vector operations although in fact no ambiguity can arise). This part of the procedure call operand is null when there are no parameters, (i.e. operator follows then closing bracket is sufficient).

(d)) operator (46). This must be present.

Parameter expressions are evaluated in the mode specified in the corresponding procedure spec. If no mode information is given in the spec then each parameter must commence with a mode word giving the parameter type. [The operand type following '(' or ',' should be 32 (operator follows). The mode is then given in the first 16 bits of the next word. Note that operand type 34 (mode follows) implies evaluation of the parameter in a different mode from that specified and must have a ']' before the next ',' or ')']. Note that a procedure may be specified more than once. If so it is the users responsibility to call it with the correct number of parameters.

Procedure call operands may involve other composite operands.

<u>Examples</u>

(i) P'(A, B, C)

(ii) P'(A'[I], C)

(iii) P'(I32 A, B)

(iv) P'(A)[I]
Procedure call yielding a descriptor.

(v) Q'(A)(B)
Procedure call yielding a procedure name used in another procedure call.

(i) Basic Operand Types

The coding of the operand type information within the second 16 bits of an operator word is now considered.

Top or A bit = 1 if address of operand is required.

It is possible for address of to be used in conjunction with a label operand. This yields a 32-bit quantity which may subsequently be used for control transfers. Normally used in 32 bit modes.

Second or D bit = 1 if 64 bit descriptor of operand is required.

The A and D bits are interpreted statically or dynamically as appropriate. The descriptor created has a bound one and type zero. Descriptor of may be used in conjunction with a procedure name. In 128 bit modes a 128-bit quantity by which the procedure may subsequently be referenced is yielded. In 64 bit modes a restricted version (with no display manipulation) used for Fortran parametric procedures is yielded. Descriptor of label generates a 64 bit quantity used in non-local jumps.

Third bit

A 64-bit descriptor of the operand is created. The first 32-bits of the operand are treated as a literal to be placed in the type and bound field. These are followed by an operand of the usual type which gives the origin field.

Fourth or P bit = 1 if next 32 bits is to be treated as a pointer to the operand rather than the operand itself.

Fifth bit = 1 This is the start of a composite operand, parameters or subscripts following.

Sixth bit = Procedure of. The 128-bit operand is to be treated as a procedure name variable. Result undefined unless the operand is a copy of a 128-bit quantity created by using 'descriptor of procedure name'.

Seventh bit = left or most significant half of 128 or 64 bit operand is required. (not in 128 bit modes)

Eighth bit = right or least significant half of 128 or 64 bit operand is required.

The remaining bits give the operand type:-

Hexa-decimal	Decimal	
%0	0	NULL (OPTIONAL SPECIFICATION OF REDUNDANT OPERAND)
1	1	Displacement for jump in conditional expressions
2	2	NAME IN THE LOCAL NAMESPACE OF A PROCEDURE
3	3	PARAMETER OF A PROCEDURE
4	4	ABSOLUTE NAME WITHIN THE NAME SEGMENT
5	5	NAME WITHIN DECLARED AREA IN ANOTHER SEGMENT
6	6	PROCEDURE NAME
7	7	SPECIAL OPERAND, B, D, STACK, ACC etc
8	8	Reserved for WHEN statement (LABEL)
9	9	FIXED ABSOLUTE NAME
A	10	64 BIT AREA NAME
B	11	ACTUAL PARAMETER
C	12	long result element
D	13	32 BIT LITERAL
E	14	64 BIT LITERAL
F	15	128 BIT LITERAL
10	16	Internal identifier
11	17	Address of properties
12	18	THEN operand (only used in WHEN statements)
20	32	ANOTHER OPERATOR FOLLOWS, after) or]
21	33	SUB EXPRESSION (FOLLOWED BY A DUMMY OPERATOR WORD. (This dummy word gives the operand type).
22	34	CHANGE MODE [FOLLOWED BY A MODE WORD
23	35	THIS OPERATOR IS FOLLOWED BY CONDITION OR REIN
24	36	ANOTHER OPERATOR FOLLOWS after NOT (BOOLEAN ONLY)
%25	37	SET BOOLEAN ACCUMULATOR ON THIS TEST

Coding of operands

32-bit literal: The 32 bits just contain the literal. Boolean literals are also represented in a 32-bit literal. If the least significant digit is 1 the literal is true, if it is zero the literal is false.

64 and 128-bit literal: The computation vector must either contain a pointer to 64 or 128 bits, respectively, or alternatively 2 or 4 32-bit operand words are used.

Local name: The bottom 18 bits give the address of the name, relative to the start of the local namespace, in 8-bit units. The top 4 bits give size, the next 10 bits give a relative textual level number; e.g., current level = 0, next outer level = 1 etc. Top 4 bits = 0 if operand is current arithmetic size (the usual case). Exceptionally when an operand of wrong size is required top bit = 1, next 3 contain size coded as before.

Parameter: Similar to local name.

Absolute name: Similar to local name except that the textual level field is not used.

Area name: Similar to local name except that the textual level field is now used to contain an area number. This area number is allocated to the area by a CTL procedure when the area is declared.

Procedure name: Similar to local name. The address bits contain a CTL reference number used to obtain properties of the procedure. In the case of COMMON procedures these bits give the directory entry for the procedure. (They are the index into this directory negated). This form is used when a procedure is explicitly addressed.

Special operands: See below.

Label operand: The operand is the CTL low level reference number.

Fixed absolute name: References one of a small number of absolute names which have fixed conventional meanings throughout the system. These names are not declared but are just used. At present 64 32-bit names exist and are provisionally allocated as follows:-

0 - 7	used as output parameters of organisational procedures. Also used as working space by CTL.
8 - 13	used by input output library procedures.
14 - 63	Used as working space by CTL. 14 - 25 also used at run time as accumulator dumps etc. The operand is coded as a byte displacement in the range 0 - 252.

64-bit area name: Sometimes the displacement in an area is greater than can be held in 18 bits. In this case 64 bit area name is used. The first 32 bits contain the area number as before with the size in the top 4 bits. The next 32 bits contain the displacement.

Actual parameter: Only used as described in section 5.6.

Long result element: Only used as described in section 5.6. The operand is the number of the result.

Displacement: used in conditional expressions. The 32 bits contain an integer representing the relative displacement of the skip in the computation vector.

Internal identifier and address of properties. These operands presuppose the existence of the standard form of property list, which is created automatically by the high level declarative procedures, for each operand referred to. Type 16 or internal identifier is the number which a name is replaced by when the name is added to the namelist. Type 17 assumes that the correct set of properties has already been found and that the address of these properties is handed over.

No further type specification is required since all the relevant information is contained within the property list.

The form of these lists is described in section 5.9.

(j) Special Register Operands

The registers of the CTL machine can be accessed in restricted circumstances for optimisation. The restrictions are:-

(1) No registers are preserved over procedure calls.
(2) The relevant standard use bit in computations must be set.
(3) No registers are preserved over store to store computations.
(4) Registers are not preserved on entry to a CYCLE.

The operands are (note pointer to operand, left part and right part cannot generally be used, => store operator should only be used with STACK. ADDR (STACK) is not allowed).

1. ACC. ACC can be used to reference the value of the notional accumulator so far or to reference a previous result at the start of a new sequence. It is not preserved over arithmetic mode changes but will be preserved between consecutive computations of the same mode, if other instructions do not intervene. Store to ACC is illegal.

2. B. Use of the operand B as the first operand in a subscript expression means that the value of the last subscript calculated is used. This value is the last non-null indirection modifier, dope vector modifier or subscript computation result. The final value is preserved over further sequences until the next modification if the B standard bit is set. B cannot be used as an 128 bit operand.

3. D. If D is specified as an operand the descriptor used depends on the previous use of D. D is preserved over further sequences when the D bit is set until the next indirect access. In an indirect sequence D holds the descriptor which when modified by the final modifier, if any, gives the operand. In a dope vector sequence D contains the descriptor of the whole array. D may also have been set by a descriptor manipulation sequence (see section 5.3). D cannot be used as an 128 bit operand. (left part of D in DTOP).

4. STACK. Explicit use of the stack is subject to the following constraints:-

   (i) Quantities stacked must be unstacked to the same size of operand.

   (ii) They must be unstacked in the same procedure.

   (iii) They must be unstacked in reverse order of stacking.

   (iv) They must be unstacked at the same level of bracketing of subexpression as when they were stacked.

   (v) They must be unstacked at the same depth of cycle nesting as they were stacked.

5. SOURCE (used in store to store operations).

6. DEST (used in store to store operations).

7. DBOT

8. DTOP

(k) Explicit unstacking

```
PROC SPEC CTL.UNSTACK (I32, I32)
```

If explicit use is being made of the stack UNSTACK enables a number of stacked quantities of the same size to be discarded without actually unstacking. The first parameter specifies the size of item being unstacked (usual coding). The second parameter is the number of items to be unstacked. All the previous rules about explicit use of the stack apply.

e.g., to unstack 7, 32-bit quantities previously stacked:-

```
CTL.UNSTACK (5, 7)
```

5.3 Store to Store Operations

A store to store operation is carried out between two strings of 8-bit items or in some cases between an operand and a string. Store to store operations destroy the values in all the special registers.

A store to store operation is invoked by a call of:-

PROC SPEC CTL.STS (S)

The parameter is a vector of 32-bit elements which describes the operation and is laid out in the following order, operator, source operand if any, destination operand, mask and byte operand or mask and filler, count if any, conditional sequence if any.

If the store to store sequence occurs in a computation vector, the operator is preceded by 32 bits giving: mode STS ; operator follows.

Element 0: The store to store operator.

These are:-

String to string

1	=	Compare
2	=	Move
3	=	Move 0
4	=	Check overlap
5-7	=	Logical string to string operations

Byte to string

8	=	Scan =
9	=	Scan ≠
10	=	Move byte
11-13	=	Logical byte to string operations

Table operations

14	=	Table translate
15	=	Table check

Accumulator packing

16	=	Unpack
17	=	Zero unpack
18	=	Zero ignore
19	=	Pack

Element 1: Mode word introducing source sequence and giving source operand, omitted in the case of byte to string and accumulator packing operations. This must yield a string descriptor. Special register operands may not be used except for previous source (SOURCE). The form of this sequence is described on p65. It terminates with operator 63.

This is followed by destination sequence. This must always be present. The previous destination (DEST) is the only permitted special register operand.

A mask and byte operand is then specified for byte to string operations. This consists of a simple operand type word followed by the operand. The bottom 8 bits are treated as source byte, the next 8 as mask. This operand may not involve the use of D and hence must be a simple operand, variable or literal. The operand is 32 bits of which the least significant 16 are used.

A mask and filler is specified instead for string to string and accumulator packing operations. This operand must be a literal. It is contained in the least significant 16 bits of a 32-bit literal. [A mask and byte/filler should be specified for operators 4, 5, 6, 7, 14,15,18, 19].

A count operand is then specified as an operand type I32 literal followed by the operand itself. This literal must be positive. It specifies the number of destination bytes to be operated on. If it is zero the count in the destination field is used instead. In any case the literal count must be less than or equal to the bound field of the destination. (If it is not the effect is undefined).

Some operations allow this to be followed by a conditional sequence of IF operators and label operands.

The store to store specification ends with the terminate operator, or the] operator.

Store to Store Operators

(a) The string to string operators

The source and destination are described by string descriptors including possibly SOURCE and DEST. Only bits corresponding to zeros in the mask are operated on. If the source expires before the count the operation continues using the filler as source. After the operation SOURCE and DEST define the remainder of the source and destination strings. If in compare a mismatch occurs this includes the byte on which mismatch occurred.

1 = Compare: Two strings are compared terminating when two corresponding bytes differ or when the count expires. The result of the final comparison can be tested, for this purpose the bytes are treated as unsigned integers. The source is considered to be on the left of the compare operator, the destination on the right. The computational relation testing operatots are used for the test.

2 = Move: The source string is moved into the destination string. Effect undefined for overlapping strings. Conditions not allowed.

3 = Move overlapped: The source string is moved into the destination string correctly even if the strings overlap. Conditions not allowed.

4 = Check overlap: If the start of the destination string lies within the source string then condition is set <u>true</u> otherwise <u>false</u>. Mask, filler and count are not applicable. SOURCE and DEST still define the complete strings. Result may be tested by conditional sequence.

Logical operators:-

5=	V	logical OR
6=	&	logical AND
7=	$\not\equiv$	logical not equivalence

No filler or mask is available for the logical operations.

(b) The byte to string operators

In these operations there is no source string. After the operations SOURCE is unchanged for all except the logical operations. After these it is undefined. A mask and source byte are specified as the bottom 16 bits of an operand. This operand is a literal or simple variable. The bottom 8 bits are used as source. The next 8 are a mask. Only bits corresponding to zero bits in the mask are operated on. Other bits in the destination remain unchanged in those orders which alter the destination and are treated as if zero in all operations. After the operation DEST defines the remainder of the destination string, including the byte on which the operation ceased if SCAN or COMPARE finds what it was looking for.

8 Scan = The source is compared with each byte of the destination in turn until an unequal byte is found or until the count expires. The result of the final comparison can be tested by a conditional sequence using the relation testing operators. The source byte is considered to be on the left of the compare operator and the destination on the right.

9 Scan ≠ As scan except that the scan ends on an equal byte. Condition is false for ending on an equal byte true if string expires. [Or the relation testing operators may be used, as for 'Scan='].

10 Move byte The source byte is propagated throughout the destination string.

Logical operations The relevant logical operation is performed between the source byte and each byte of the destination placing the result in the destination.

11=	V	logical OR
12=	&	logical AND
13=	$\underline{\neq}$	logical not-equavalence

No masking takes place in these operations which, unlike other byte to string operations, destroy SOURCE.

(c) Table operations

In these operations source describes a vector containing a table rather than a string. After the operation SOURCE still defines the source vector while DEST is the remainder of the destination. Literal source, filler and mask are ignored. The result is undefined if the source is not type 0 or 2 or not of the correct size.

14 Table translate: Each byte in the destination string is replaced by the nth byte in the source vector where n is the numeric value of the destination byte. If n is greater than the size of the source vector bound check interrupt occurs. Conditional sequence not allowed.

15 Table check: The value n of each destination byte is used to access the nth bit of the source bit vector. If this is zero the operation continues otherwise it terminates with DEST giving the destination byte for which a one occurred. Bound check fail may occur. Conditional sequences may be used to check termination on a 1 bit which gives false otherwise the result is true.

(d) Accumulator packing operations

These operations work between the accumulator ACC and the destination string. No source of any form is specified; instead the accumulator mode is given as in computation. A mask/filler literal may be required. The result of unpacking is undefined if the given mode is not the same as the current dynamic accumulator contents. SOURCE is undefined after these operations but DEST has the usual meaning.

16 Unpack: A decimal value in the ACC (set by a previous computational sequence) is translated into a character sequence in the destination. The least significant 4 bits of each byte are found from the most significant 4 bits of ACC and the most significant bits of the byte are 0110 (ISO zone bits) or in the case of any non-zero mask the most significant 4 bits of the filler. After each byte is filled the ACC including sign is shifted left one decimal place (i.e., fixed point 4-bit shift rather than a decimal shift). Condition sequence not allowed.

17 Zero unpack: Similar to unpack except that:-

(i) The destination value is always the filler.

(ii) The operation ceases if after a shift the most significant 4 bits of ACC are non-zero.

A conditional sequence may be used to test whether a non-zero digit was encountered (false). If the unpack was completed the result is true.

18 Zero ignore: Identical to zero unpack except that the destination is disregarded except for decrementing the bound field as in zero unpack and the digits unpacked are discarded.

19 Pack: For each destination byte ACC is shifted left one decimal place and then the least significant 4 bits of the byte forced into the least significant 4 bits of the ACC. Shift overflow may occur. The top 4 bits of each byte are ignored.

These operations may be extended to apply to arithmetic modes other than decimal.

Descriptor Manipulation Sequences

Descriptors consist of two parts; the bound and type field and the origin field.

Sometimes an operand can be directly manipulated in the D register and computational modes are provided for this purpose. These sequences commence within implicit load as usual and have the following restricted set of functions:

Hexa-decimal	Decimal	
%38	56	DTOP = operand Replace the top half (bound and type field) of the D register by the 32-bit operand. The origin field of D is unaltered.
39	57	DBOT = operand Replace the origin field of the D register by the 32-bit operand. The top half of D is unaltered.
3A	58	DBOUND = operand Replace the bound field of the D register by the least significant 24 bits of the 32-bit operand. The type and origin fields of D are unaltered.
9	9	D => operand Store the D register in the 64-bit operand.
%3B	59	MOD Scale the 32-bit operand as for modification, add it to the origin field of D, and subtract (unscaled) from the bound field. Bound field check fail may occur with any type of descriptor. MOD operates on all types of descriptors. If this function is applied to automatic indirect (Type 3.3) descriptors the indirection occurs first and the MOD operation on the resulting descriptor. If the function is applied to a procedure call descriptor a procedure call results. The operand may be negative but this will cause bound check fail unless the bound check interrupt is inhibited.

%3C	60	MDR The action of MOD takes place followed by replacing the contents of the D register by the element it describes i.e., D = D[0].
3D	61	SMOD Similar to MOD but does not scale the modifier, does not alter the bound field and does not bound check, regardless of descriptor type. There is no automatic indirection or procedure call on these types of descriptors.
3E	62	RMOD Similar to SMOD except that the operand is 64 bits and includes the type and origin fields to be placed in D.
35	53	D *= Stack the D register and load the operand to D.
% 36	54	Terminate (in computational sequence) End the D sequence.
% 3F	63	End source or destination sequence within store to store sequence.

The computation or store to store procedures are used to invoke D sequence The most significant half of the initial 32-bit word of the vector determines the type of sequence.

D: (Mode % 803E) Operations on D as described above. These operations destroy SOURCE and DEST of store to store operations. A D sequence may be a sub-sequence within another computational sequence.

SOURCE: (Mode %8056) A similar sequence working on SOURCE of store to store operations. Such a sequence destroys DEST and D, and may not be used as a sub-sequence within another computational sequence.

DEST: (Mode %805E) A similar sequence working on DEST of store to store operations. Such a sequence preserves SOURCE but destroys D, and may not be used as a sub-sequence.

The initial word is followed by a sequence of operator words and operands. The operands may be general operands as described under computations and so may involve subscripting operations. The sequence ends with a termination operator.

5.4 Control and Conditional Facilities

The first control statement described is the WHEN statement which provides conditional jumps to labels in the current procedure by a parameter format very similar to that for conditional computational sequences.

PROC SPEC CTL.WHEN (S)

The parameter is the descriptor of a vector with 32-bit elements. This vector is similar to a COMPUTATION vector. However, it must commence with a CONDITION or RELATION operator. After this a sequence to be tested, identical to those previously described, is followed by one or more IF operators. Each IF operator is usually followed by the name or properties of a label. A basic label operand, type 8, could be used; the actual operand is a 32 bit label reference number generated as described in section 5.8. An IF operator may in fact be followed by any 32-bit operand. If the operand is not a label and has been created by any means other than use of address of label the result is undefined. The labels should be local. The sequence must be terminated after the final IF operation. The value of the accumulator after a WHEN operation is undefined except when the only tests compare the current accumulator value with zero, in which case it is preserved.

e.g.,

```
            IF [ACC = 0], -> L1    preserves the accumulator.
```

Example The MU5 autocode statement:-

```
            IF [A = B], -> L1
```

correspond to a call of WHEN with parameter CODE as follows:-

```
        CODE [0]:    RELATION operator, mode word follows
             [1]:    I32, name follows
             [2]:    Name A
             [3]:    Compare, name follows
             [4]:    Name B
             [5]:    IF =, named label follows
             [6]:    Label L1
             [7]:    Terminator
```

It is possible to avoid writing explicit labels and jumps when a set of statements is obeyed conditionally, as in MU5 autocode. In this case the single IF operator is followed by a special THEN operand. When the condition is true then the following statements up to ELSE or CONTINUE will be obeyed. If ELSE appears the statements between ELSE and CONTINUE are obeyed only when the condition is false. This is implemented by the parameterless procedures:-

```
PROC SPEC CTL.ELSE()      (must follow a use of WHEN with a
                           THEN operand)
PROC SPEC CTL.CONTINUE() (must follow WHEN or ELSE)
```

<u>Example</u>

```
IF [A = B] THEN                          IF [A ≠ B], -> L1
S1                                       S1
ELSE         is equivalent to            -> L2
S2                                       L1: S2
CONTINUE                                 L2:
```

The CYCLE procedure is provided to take advantage of any special looping functions provided.

PROC SPEC CTL.CYCLE (S)

The parameter is computation vector containing an operand type, an operand, and a terminator coded as before. The possible types are very restricted.

(i) A 32-bit integer literal. The instructions between CYCLE and REPEAT are obeyed the number of times specified. Quantities must not be left on the stack between CYCLE and REPEAT for this type of cycle.

(ii) The name of a 32-bit scalar variable (64 bit area name not allowed). The variable must be set to a suitable starting value before the cycle is entered. The instructions within the CYCLE are obeyed at least once.

(iii) The special operand B. In this case B must have been set by the previous instructions. It must not be altered within the CYCLE. The instructions within the CYCLE are obeyed at least once.

PROC SPEC CTL.REPEAT (I32, S)

If CYCLE form (i) is the used the parameters are ignored.

Otherwise the first parameter is a 32-bit signed integer which is the increment made to the controlled variable before the ending test.

The second parameter is part of a computation vector commencing with a COMPARE operator and ending with an IF operator followed by a terminator. This ending condition expression is evaluated in I32 mode each time the end of the loop is reached.

Examples: (in symbolic notation)

```
(i)     CYCLE (3)
         . . . .
        REPEAT

(ii)    I32, 1 => B
        CYCLE (B)
        A[B] => C[B]
        REPEAT (+1, ≤ 10)
```

The general form of MU5 autocode CYCLE and REPEAT therefore corresponds to the following CTL statements:-

(i) A computational sequence to assign the starting value to the controlled variable.

(ii) The CYCLE statement with controlled variable as operand.

(iii) The instructions between the CYCLE and REPEAT.

(iv) The REPEAT statement with step and final value.

The autocode translator may translate particular CYCLE's to one of the more optimal forms wherever this is possible.

Labels The code compiled for a jump instruction depends on whether the label concerned is local to a block or non-local. In a two-pass language this information is usually known but in a single pass forward references and non-local jumps cannot usually be distinguished. In this case a forward local reference should be compiled. If at the end of the procedure the reference is outstanding an exit sequence can be planted at that point.

Forward and backward local references are therefore dealt with by:-

```
                    PROC SPEC CTL.GOTO (I32)
          and       PROC SPEC CTL.JUMP (S)
```

GOTO is used for a simple jump. The parameter is the internal identifier of a label. Otherwise JUMP is used and the parameter is a vector containing an operand type and operand. The operand will be an I32 variable which is being used as a label. Literal operands are not permitted nor are procedure name operands.

A label is declared at the point at which it occurs by:-

```
                    PROC SPEC CTL.LABEL (I32, S)
```

The I32 parameter is the internal identifier corresponding to the label. The second parameter is a descriptor of private properties, if any. In both CTL.GOTO and CTL.LABEL, if the label name has already been declared at the current level, other than as a label, and in CTL. LABEL if the label is already defined, a fault occurs. New label properties are added unless the identifier (first parameter) is negated. A non-local jump one dynamic level out is provided by:-

```
                    PROC SPEC CTL.EXIT.TO (S)
```

where the parameter is an operand as before.
This form of jump removes the namespace of the procedure being left.
A procedure is provided to enable the user to determine which forward label references are outstanding:-

```
                    PROC SPEC CTL.NON.LOCAL.REFS (S)
```

The internal identifiers of labels and procedures (tagged with a 1 in the top bit) for which there are outstanding references in the current procedure are planted in the vector with 32-bit elements given as parameter, followed by a zero. The user may then generate EXIT TO statements or fault messages as appropriate.

In some compilers it is necessary to perform non-local jumps interpretively. In this case each block or procedure must be provided with an exit sequence using the CTL.POSTLUDE facility described in section 5.6. These sequences are used in conjunction with the non-local jump interpreter. This interpreter is entered using:

```
CTL.EXIT.JUMP(S)
```

The parameter of this procedure is computation vector describing a 64 bit operand. This operand is created using descriptor of label. It contains both a label address of the label to be jumped to and a value of NB to ensure that the correct recursive call of the relevant block or procedure is found.

Note: A label in a subprocedure cannot be accessed from an enclosing prowedure when the descriptor is created (i.e., similar scope rule as for variables).

<u>Switch jumps</u>

A switch to a number of labels, depending on the valu of an I32 variable can be made. The switch labels may be specified in a switch declaration:-

```
PROC SPEC CTL.SWITCH (I32, S, S)
```

where the parameters are:-

(i) Internal identifier of the switch.
If the name has already been declared at the current level a fault occurs. New properties are added unless this parameter is negated.

(ii) Either a vector of switch labels, represented by internal identifiers. These should correspond to local labels declared elsewhere in the current procedure.
Or a descriptor with zero origin field which just gives the number of switch elements in the bound field. In this case the labels are not separately named but declared using the procedure given below.

(iii) A descriptor of a set of private properties for the switch. Zero if no private properties. Non zero bound, zero origin gives number of properties without filling them in.

If the switch labels are not named in the second parameters then they are not declared using CTL.LABEL but by calls of:

```
                PROC SPEC   CTL.SWITCH LABEL(I32,I32)
```

The first parameter is the internal identifier of the switch while the second is the number of the element being declared.

The switch is then used by:-

```
                PROC SPEC CTL.JUMP.SW (I32, S)
```

where the parameters are:-

(i) Name of switch as an internal identifier.

(ii) Computation vector specifying an I32 calculation determining the switch index.

The switch elements are numbered from zero upwards.

5.5 High Level Data Declaratives

There are high level declaratives for both scalars and structures which correspond closely to the facilities in the autocode.

The scalar facility is:-

```
PROC SPEC CTL.SCALAR.DEC (I32, S, I32, S, S)
```

This procedure performs two functions:-

(i) It creates a set of properties for each name declared.

(ii) It allocates space for a sequence of consecutive names within either the local namespace or some other area (fer area declarations see later in this section).

The names are handed to the procedure in the form of a vector of internal identifiers. This means that the standard name and property list environment must exist and that the names have already been replaced by the corresponding internal identifiers.

(a) The first parameter is a 32-bit integer containing the size of each declared variable coded in the bottom 3 bits as for computations. The size may be 1 bit, 32 bits, 64 bits or 128 bits. The parameter may also optionally contain the type of the variable coded as for computations in the next 4 bits. This type information is not used by the CTL but may be inspected by the translator.

(b) The second parameter is a descriptor of the 32-bit element vector of internal identifiers. If the first such identifier is negated and properties (of any of the identifiers) are already present in this block (a fault) new properties are not addeed, otherwise they are.

(c) The third parameter gives the type of area or position at which the names are being declared.

0	=	local
-1	=	absolute
1	=	area name
2	=	name of previously declared scalar (for equivalence)
3	=	vector element (for equivalence)
4	=	area element (+ byte displacement)

(d) This parameter gives the actual position of the names which are being declared and its use depends on the value of parameter 3 as follows.

0, -1,	irrelevant and ignored
1	The origin field contains an internal identifier corresponding to the area name.
2	The origin field contains an internal identifier corresponding to the scalar name onto which the new variables are being equivalenced. This must be a previously declared static or local dynamic item whose declaration is still valid.
3	The parameter is the descriptor of a computation which defines a vector name and a subscript expression.
4	Descriptor of a computation vector giving area [byte displacement].

(e) The bound field of the final parameter is the number of bytes of additional private properties, for use by the translator, for each declared name. The origin field is zero if no properties are supplied but otherwise the descriptor describes the properties to be copied.

Vectors and Strings

The high level declarative facility for vectors and strings is:-

```
PROC SPEC CTL.STRUCTURE.DEC (I32, S, I32, S, S, S)
```

This procedure can be used for static or for dynamic declarations. In the latter case suitable instructions will be compiled. Note that no declarative is provided for multi-dimensional arrays so array declarations should be programmed out.

(a) The first parameter is a 32-bit integer containing the size of each element in the bottom 3 bits; sizes 1, 4, 8, 16, 32, 64 or 128 bits are permitted. The next 4 bits are optional element type information as for scalar declarations.

The top 8 bits of this parameter give the type bits for the structure descriptr being declared. In particular the top two bits are:

00 = vector
01 = string

The remaining bits will usually be zero.

A string may have only 8 bit elements.

(b) The second parameter is the descriptor of a vector of 32-bit internal identifiers. These may be previously declared 64-bit scalar quantities or new local declarations. Thus local descriptors will always exist, or be created. The first identifier is negated if properties are to be added (for any of the identifiers), when non-scalar ones already exist.

(c) The third parameter gives the type of area the elements are assigned to:-

0	=	Dynamic
1	=	Static area
2	=	Illegal
3	=	Previously declared static vector element (for equivalence)
4	=	area element

(d) The actual position of the elements. When the third parameter:-

= 0	this parameter is ignored
= 1	an internal identifier of an area (in the bottom 32 bits)
= 3	Descriptor of a computation vector which gives an element within a structure. [The structure name must be an identifier or property address (Types 16, 17)].
= 4	Element within area (descriptor of a computation vector giving area [byte displacement]).

(e) This parameter is the descriptor of a computation vector containing just the bound (i.e., number of elements, or maximum subscript + 1). This is in the form of a computation sequence which will be evaluated statically or dynamically as appropriate.

(f) Descriptor of the additional private properties required in the property lists for each structure declared as for scalar declarations,

Initialised data

PROC SPEC CTL.DATAVEC (I32, I32, S, S, I32)

This procedure directs the CTL to assemble literal data into an area.
The parameters are:-

a) Internal identifier corresponding to the name by which the data vector is referenced. If this is not previously locally declared a literal descriptor is created. If the previous declaration is a 64 bit scalar, code is planted to assign the literal descriptor to the variable. Otherwise, a fault occurs and new properties are added unless this parameter is negated.
b) Size of each element of the vector (bottom 3 bits). Optional type information in next bits, optional descriptor type information in top 8 bits.
c) Descriptor of the data.
d) Descriptor of private properties (if any).
e) Internal identifier of the area (usually not necessarily read only) in which datavec is to be placed. This must be an area declared by CTL (i.e., not type 3). If this parameter is zero, the datavector is placed in the code segment, with a jump round it.

If the origin of parameter 3 is zero, then the datavec elements are filled in, in order, by repeated calls on:-

PROC SPEC CTL.DATA(S)

The parameter is a computation vector containing the literal datavec element, which must be one of the following operand types:-

(i) a literal
(ii) address of a procedure or label
(iii) a named literal (forward references are allowed)
(iv) addr/Desc of a static item
(v) desc of a procedure (yields a 64 bit value)

No check is made on the number of elements given in this way. A call on CTL.DATA always refers to the next element of the last data vector which was specified with no data.

id size desc props

PROC SPEC CTL.CONST (I32, I32, S, S)

This allows a literal to be named. The parameters are similar to the first 4 parameters of CTL.DATAVEC. P3 is a vector descriptor of 32 bit elements which contain the constant.

Area declarations

When data items are not local or absolute names they must be within declared areas. All area names are global. A program will usually require at least one such area for structure elements. An area is obtained by use of the procedure:-

PROC SPEC CTL.AREA (I32, I32, I32, S, S)

where the parameters are:-

(i) The name of the area represented as an internal identifier.

(ii) The size of the area as an integer number of 8-bit bytes. This size is not limited by hardware considerations. If this size is given as zero the area is of unspecified size. The size will then be determined at load time but must not exceed one segment.

(iii) Area type: 1 position chosen by CTL and segment declared by CTL.
2 position chosen by user but segment declared by CTL.
3 area accessed by setting XNB to a variable. Segment must be declared by user.

In addition this may be tagged
4 segment exists already
8 segment should be read only.

(iv) Descriptor of computational sequence giving an operand. Area type 1, ignored. Area 2 - This may be a literal, a particular byte address at which the area starts. Area 3 - It may be an absolute or fixed absolute name for variable giving start address.

(v) Descriptor of private properties.

An area may be mapped in different ways in different sections of program. To achieve a new mapping the area is just redeclared. If the name has been previously globally declared other than as an area, a fault occurs. The new properties will be added unless the identifier (parameter 1) is negated.

PROC SPEC CTL.AREA.SIZE (I32, S)

Yields in the 2nd parameter the amount of the area (in 8-bit bytes) already used. The first parameter is the internal identifier of the area, or zero for local namespace, -1 for absolute namespace. Zero is returned if the name is not an area.

5.6 Procedure Organisation

Procedure Specification

The information which must be provided for each procedure is:-

(i) Type of procedure
(ii) Mode of result
(iii) Number of parameters

The following information may be provided:-

(iv) Type of each parameter (If unspecified the parameter size is 64 bits).

This information is given, in the parameters of the procedure:-

```
PROC SPEC CTL.PROC.SPEC (I32, I32, S, S, S)
```

The parameters of this procedure are:-

(1) Internal identifier of procedure. If this name is already defined at the current level (or globally if the relevant bit is set in parameter 2) other than as a procedure, a fault occurs. Procedure properties are added unless this parameter is negated.

(2) Type of procedure and mode of result. This mode may be NIL - no result, all computation modes, descriptor, label or procedure in the code given below. The top bits are used to specify the type of procedure.

= 00	general recursive
= 01	sub procedure
= 10	static Fortran type
= 11	general recursive, possibly with variable number of parameters

If the third bit, $^{2}20(7)$, is set to one, the procedure properties are added at the outermost level.

(3) A vector of 32-bit elements giving the parameter specification of each parameter. The bound of this vector determines the number of parameters. See below for details.

(4) Descriptor of private properties for procedure.

(5) Descriptor of a vector of descriptors. Each of these descriptors describes a set of private properties, one set for each parameter.

The detailed parameter specification given in parameter 3 is as follows. If a parameter is arithmetic then the type and size is specified in the same way as in a mode word for computation. Otherwise it may be a procedure, subscript, source, dest, descriptor, label or unspecified.

These are given as follows:-

0 = parameter type is not specified. Explicit mode specification must be given in each actual parameter computation. No check is made on the use of formal parameters. The parameter is assumed to be 64 bits in this case.

Arithmetic:	type/size as in computation
Descriptor:	type 7, size always 64 bits
Label:	type 8, size 32 or 64 bits
Procedure:	type 9, size 64 or 128 bits

For further information on output parameters see the section 'Stack results'.

A procedure must be specified before it is used or declared. It may be specified more than once. In this case the given specifications must be consistent except when variable number of parameters is being used. On each respecification, a new set of properties is added. This procedure may be used to specify a procedure parameter mentioned in the specification of the enclosing procedure.

If a compiler requires a common library procedure entry on its own namelist then a specification must be supplied by means of:-

```
PROC SPEC CTL.COMMON.PROC.SPEC (I32, I32, S, S, S, S)
```

The first 5 parameters are as for PROC.SPEC. The sixth parameter describes a string of characters giving the system (as opposed to user) name of the procedure.

Procedure Declaration

The body of a procedure declaration is introduced by:-

```
PROC SPEC CTL.PROC (I32, S, S)
```

where the first parameter is the internal identifier of the procedure, and the second parameter is the descriptor of a vector of parameter names (internal identifiers). The third parameter is a 32 bit vector descriptor used as follows:

Element 0 returns the address of the properties of the procedure entry on the property lists.
Element 1 onwards give user defined run time properties for the information vector associated with the procedure.

The body is terminated by:-

```
PROC SPEC CTL.END()
```

This has the effect of conceptually removing the level in the namelist created by entering the procedure.

Within the procedure two special statements may be used:-

```
      (1)    PROC SPEC CTL.RESULT.IS (S)
  and (2)    PROC SPEC CTL.RETURN()
```

RESULT IS takes the computation given by the parameter and uses the result as the value yielded by the procedure. Procedure exit immediately follows generation of this result. RETURN invokes procedure exit without yielding any results. The result is actually yielded in the accumulator for an arithmetic result, in the I32 accumulator for a label, in the U128 accumulator for a procedure, and in the D register for a descriptor.

When the user requires a non-local jump through a procedure he must call CTL.POSTLUDE () at the start of the exit sequence for the procedure. This need not be immediately preceding the RETURN instruction since the exit sequence may include instructions to reset user defined pointers before the actual RETURN. CTL.POSTLUDE makes the address of the exit sequence available to a non-local jump processor.

Stack Results

This facility allows a procedure to yield a result to the stack. A stack result is not restricted to a single item but may consist of several basic elements. After executing the procedure the result may be unstacked or it may be handed as a parameter or parameters to another procedure. It cannot be used for static, functional or parametric procedures but only in procedures which are called in a separate statement.

The parameter specification section of CTL.PROC.SPEC is used to supply details of the result items. These are specified as additional parameters preceding the ordinary parameters in the vector and each tagged with 1 in the most significant bit. These stack parameters are then named in CTL.PROC in the same way as ordinary parameters.

Within the procedure these stack results are referenced by name or by operand type 12 at the low level. Outside the procedure they are referenced by the special register operand STACK which enables the elements to be unstacked one at a time. If they are used as parameters to another procedure the CTL.PREPARE.PROC.ENTRY facility, described later, must be used.

Declaration of a procedure with a variable number of parameters.

If a procedure is specified as possibly having a variable number of parameters at a call the parameters must all be of the same type. When the procedure is declared a new specification must be given. Either the declaration has a fixed number of parameters, in which case an ordinary spec is given, or it has a variable number. In the latter case the procedure must be specified as having three parameters and being of type 11 (variable number of parameters). The first parameter must be of descriptor type, the second I32 and the third also I32. The heading is then given using CTL.PROC, again with three parameters. The first statement of the procedure must be:-

```
PROC SPEC CTL.VARIABLE.PARAMETERS()
```

This arranges that the descriptor (first parameter) is the descriptor of the parameters handed to the procedure at a call which are now accessed as a vector. The second parameter, an I32 variable, will contain the number of parameters for this particular call. The third parameter given in the spec is not an accessible parameter but a dummy which indicates the size of the elements in the created vector by the mode it specifies. Note that if the actual parameters are 128 bits, the descriptor will have 64 bit elements. Thus two accesses will be required for each parameter.

Procedure Call

A procedure may be called in one of three ways.

(1) By use of the procedure

    PROC SPEC CTL.CALL (S)

where the parameter is part of a computation vector corresponding to a procedure call composite operand.

(2) By inserting a procedure call composite operand (which should yield a result) into a general computation vector.

(3) If the parameters involve a large amount of calculation then a more basic method is used. Before evaluation of the first parameter the procedure

    PROC SPEC CTL.PREPARE.PROC.ENTRY (I32, S)

is called. The I32 parameter is the identifier of a label on the instruction which enters the procedure. The second parameter is the descriptor of a computation giving the procedure name operand.
Subsequent computations, may involve writing to the parameters of the procedure to be called, in order once only (using operand type 11 and zero operand bits); this is followed by procedure entry

    PROC SPEC CTL.ENTER (S)

The parameter is the descriptor of a computation describing the procedure name operand. This is the instruction which must be preceded by the label referred to by 'prepare procedure entry'. On exit control is returned to the following instruction.

Blocks

Sometimes it is necessary to introduce a new namespace without actual entering a procedure. This is achieved by:-

PROC SPEC CTL.BLOCK (S, S)

which creates a new namespace as well as a new level of nomenclature in the name and property lists. It is therefore like an open procedure without parameters.

PROC SPEC CTL.BEGIN (S, S)

This creates a new level of nomenclature, without creating a new namespace.

Both these types of block are terminated by a matching END. The parameters of BLOCK and BEGIN are:-

(i) A vector descriptor of 32 bit elements. On input element 0 = 0 except in the final pass of a multipass compiler. Then it is a property list address. Element 0 returns the address of the properties of the block. For BLOCK elements 1 onwards give user defined procedure directory information for the block.

(ii) Descriptor of private properties for the block.

Informal Procedures

If a procedure does not require any local names then it may be written just as a labelled section of code and called informally as a subroutine by:-

PROC SPEC CTL.STACK.LINK (I32)

The parameter is just a label, in the current procedure, to which control is to be returned on exit. If the name has already been declared at the current level, other than as a label, a fault occurs. New properties are added unless the parameter is negated. Stack link is followed by a jump to the label at the start of the subroutine. The last instruction of the subroutine is:-

PROC SPEC CTL.EXIT.STACK()

which exits on the most recently stacked link.

The rules for use of the stack within procedures, i.e., that quantities stacked must be recovered in the same procedure, apply also to informal procedures,

Textual Level Display

A display containing the start of the name space for the most recent procedure activation is held in absolute names. This display is updated automatically on procedure entry and exit. The size of this display may be given by directive.

The display may be stored and restored from locations in the user's store. Such storing is from the textual level given by a parameter up to and including the current level.

```
PROC SPEC CTL.STORE.DISPLAY  (I32, S)
PROC SPEC CTL.GET.DISPLAY  (I32, S)
```

The parameters are:-

(1) Number of textual levels to be changed - including current level. 0 means all up to current level.

(2) Computational vector yielding an operand to or from which the display is to be moved. This operand should be a descriptor of a suitable 8-bit element vector. (This enables use of store to store orders. However, length must be 4 x number of textual levels).

Tasking

Any procedure may be called as a PL/1 task. This is achieved by using a special call statement which creates a new name segment. A copy of the textual level display and procedure directory is made in the new segment. Additional information to be copied may also be specified.

The CTL statement for this is:-

```
PROC SPEC CTL.TASK (S, I32)
```

The first parameter is the descriptor of the procedure call computation vector. The second is the amount of additional information to be copied when the task is set up.

Thunks

A block entered by use of a procedure call descriptor is identical to a BEGIN block except that it is headed by:-

```
PROC SPEC CTL.THUNK()
```

rather than BEGIN. Within this block RESULT yields the descriptor to be used when the operand access which forced the procedure call is repeated.

Procedure variables

A 64 bit procedure variable (used in Fortran) consists of a descriptor of the information vector of the procedure. (Element 0 of this vector is the address of the first instruction). No display manipulation takes place for this type of procedure.

An 128 bit procedure variable consists of a textual display descriptor together with a descriptor as above. Automatic display manipulation takes place on calling such a procedure.

5.7 Miscellaneous Functions

compile mode run mode [illegible] descrip.

1 2 3 4 5 6 7 8

PROC SPEC CTL.INITIALISE (I32,I32,I32,I32,I32,S,PROC,I32)

Before any calls on other CTL procedures are made the CTL workspace, namelist and other descriptors must be initialised. This is done by calling CTL.INITIALISE. The first two parameters are compiler parameters 1, 2. The 3rd parameter (P3) is segment number for CTL workspace, name and property lists. This is normally chosen by CTL in which case P3 = 0. When compiling a common procedure this parameter gives the common procedure index number.

P4 = address (in 16 bit units) at which program is compiled. P4 = 0 usually and address is chosen by CTL. This cannot be in a common segment unless P4 = P5 or P5 = 0.

P5 = address (in 16 bit units) at which program is to run. Usually the same as P4 and often zero. [Note that P4, P5 need not give an address at the start of a segment, but they must do so if the program is to be entered by ENTER, or filed by DEFINE (except for Perm)].

P6 = compiler parameter 3.

P7 = the descriptor of a user defined procedure called when a fault is detected by CTL.

P8 = Perm segment number. This must be given when compiling with perm present (i.e., compiling modes 0, 1, 6, 7). The initial name and property lists will be read from the end of the perm segment (except for compiling mode 7).

CTL.INITIALISE may be called by a user (with a ** command) before the compiler is called. It will also be called whenever a new compiler is entered during a mixed language program. CTL will be initialised on the first call only. However, on each call, the fault procedure, P7, is taken to be the one required if it is non-zero. Thus the user wishing to call CTL.INITIALISE before calling the compiler can set this parameter zero.

PROC SPEC CTL.END.COMPILE()

This procedure must be called before the compiler finally exits to the program control sequence. It ensures that a starting address is set for ENTER and that FAULTY(V32 63/0), which is used as an ENTER switch, is set correctly.

Initialisation Sequences

The translator may direct that instructions are to be obeyed only once on entry to a program by the parameterless procedures:-

```
PROC SPEC CTL.PRELUDE()
PROC SPEC CTL.END.PRELUDE()
```

which may occur anywhere in a sequence of CTL calls. The operands which can be used in such a sequence are restricted. Compile time computations may enable CTL to minimise the number of instructions planted within these sequences.

```
PROC SPEC CTL.PROC.PRELUDE()
```

Instructions between CTL.PROC.PRELUDE and CTL.END.PRELUDE are obeyed each time a procedure is entered, after NB is set, but before other instructions of the procedure. If a user wishes to set a prelude at any position during the procedure, a dummy prelude (i.e., a call on CTL.PROC.PRELUDE, followed by a call on CTL.END.PRELUDE) must be specified as the first instruction in the procedure.

Input/Output

The input/output system is based on the conventions and COMMON procedures described in the MU5-MX/3 library manual.

Compile Time Errors

When CTL detects an error in the parameters a user defined fault procedure will be called. The nature of the fault will be indicated by an I32 parameter handed to the user procedure. These faults are listed on page 115.

Other Reply Information

CTL yields the current value of a modifier in the code vector in one of the fixed absolute locations used for CTL communication. (CODE.PTR V32 29/0). ADDR CODE V32/0 53

Run Time Errors

Run time errors, detected by hardware or supervisor, may cause an interrupt in the user program. The library fault procedure will be called, unless an alternative trap has been set by the user. Note that this requires a linetable set up by calling CTL.LINE(I32) at the start of each line. The parameter is the page/line number. (See page 118).

Debugging Aids

The following procedures are provided as debugging aids:-

1) PROC SPEC CTL.PRINT.PL ()

This procedure prints, on output zero, all names used, each with the corresponding property address for the first set of properties in its same name chain, followed by the CTL property list. The property list is printed in hexadecimal, each line consisting of the address followed by eight 32 bit words of properties.

2) PROC SPEC CTL.OUT.CV (S)

This procedure prints, on output zero, the computation vector given as parameter, in hexadecimal, up to the terminator. If there is no terminator, the length of the vector is determined from its bound.

If the top bit [²80(7)] of the compiling mode is set when handed as a parameter to CTL.INITIALISE, CTL.OUT.CV is called, by the CTL, for every high level CTL computation vector (except those given as parameters to CTL.SCALAR.DEC and CTL.STRUCTURE.DEC). The user can alter the setting of this flag dynamically by resetting the top bit of COMPILE.MODE, a fixed absolute name (V32 27/0).

Code Streams

PROC SPEC CTL.CODE.STREAM (I32)

Out of line sequences may be compiled by selecting a different code stream which remains selected until the call of CODE STREAM. The parameter is the code stream number.

Program Interrupt Mask

Instructions are provided to inhibit interrupts on certain conditions. Each condition is represented by a bit in a program mask which is altered as follows.

PROC SPEC CTL.SET.INT.MASK (I32)

For each 1 specified the corresponding interrupt is inhibited. For each zero the corresponding state is unchanged. (This procedure destroys BM.)

PROC SPEC CTL.RESET.INT.MASK (I32)
(destroys BM)

For each zero specified the corresponding interrupt inhibit is removed. For each one the corresponding state is unchanged.

When overflow interrupts are inhibited overflow bits may be set and inspected by:-

PROC SPEC CTL.TEST.OVERFLOW (I32, I32)
(destroys BM)

The first parameter gives the overflow digit or digits which are to be tested; the second is the internal identifier of a label to which control is to be transferred if any of these digits is set.

The maskable interrupts are as follows:-

0	=	Floating overflow
1	=	Floating underflow
2	=	Fixed overflow
3	=	Decimal overflow
4	=	Zero divide
5	=	Bound check overflow
6	=	Size overflow (in secondary operand accessing)
7	=	Index (B arithmetic) overflow
8	=	Short source string in store to store orders

(digit 0 is the left most digit in the 32-bit word).

5.8 Low Level Facilities

USERS ARE WARNED THAT THESE FACILITIES MAY BE MODIFIED OR WITHDRAWN DURING THE CURRENCY OF THIS MANUAL.

Many of the low level facilities are virtually identical to the corresponding high level procedure except that the capability for dealing with internal identifiers and property lists is removed. The facilities are listed with descriptions of differences from the corresponding procedures already described. The high level facilities are in fact implemented using those listed below.

(a) CTL0 (I,I,I,I,I,S,PROC) Similar to CTL.INITIALISE

(b) COMPUTATION (S) (The standard low level facility name is CTL1).
Operand types 16 (Internal identifier) and 17 (address of properties) illegal.
Operator types - all legal at present.

(c) STS (S) (CTL2)
Same restrictions on operands as for COMPUTATION.

(CTL3 is spare)

(d) WHEN (S) (CTL4)
Same restrictions on vector describing condition as in COMPUTATION. The special THEN operand is also not permitted. The ELSE and CONTINUE statements are not implemented at the low level.

(e) CYCLE (S) (CTL5)
Same restrictions on the operand as above.

(f) REPEAT (I,S) (CTL6)
Same restrictions on parameters as above.

The high level label procedures GOTO, LABEL and EXIT TO do not appear but are implemented by the following:-

(g) PROC SPEC LABEL SPEC (S, S) (CTL7)
Which precedes the first use of a new local label. The input parameter is null for a local label or the descriptor of a character string. The procedure yields an integer (second parameter) which the label can subsequently be referenced.

(h) PROC SPEC LABEL REF (S) (CTL8)

which causes an unconditional jump, to the label specified by the parameter, to be compiled. The parameter gives the relative textual level and the label number of the label in a vector if the jump is to a specific label. Other operands, which must be I32 quantities may be specified. This procedure is only suitable for local jumps since no change is made to the stack or namespace.

(i) PROC SPEC EXIT (S) (CTL9)

is a similar procedure which causes a jump out one level from the current procedure to the label specified as parameter and removes the namespace of the procedure being left. The postlude address for the proc/block must have been set by calling CTL42.

(j) PROC SPEC LABEL DEF (I32) (CTL10)

This procedure is used when the label is encountered in the program. It causes outstanding references to the label to be filled in.

(k) PROC SPEC NON LOCAL REFS (S) (CTL11)

This procedure yields the reference numbers of labels which have been specified but which have not been declared in the current procedure. The user can then provide non local jumps or fault messages as appropriate. The parameter is the descriptor of the vector of 32-bit elements in which the numbers are placed by the procedure. This procedure differs from the corresponding high level one, in that it does not check for undefined procedures.

(l) PROC SPEC SWITCH (S, S, S) (CTL12)

The switch name is specified as a descriptor of a set of characters or null if the switch is purely local. The second parameter is a vector of 32-bit elements. These are the numbers of the previously specified local labels corresponding to the switch jumps, (e.g., element 0 corresponds to switch jump with index 0). If the origin is zero, the bound gives the number of elements to be set later. The third parameter yields a number by which the switch can be referenced.

(m) PROC SPEC JUMP SWITCH (I32, S)   (CTL13)

The first parameter is the switch number given in the same way as a label number. The second is the descriptor of an I32 computation which yields the switch index used in the jump.

PROC SPEC SWITCH LABEL (I32, I32)   (CTL14)

The first parameter is the switch number; the second the element number for this label.

(n) PROC SPEC AREA (S, I32, S, I32, S)   (CTL16)

The parameters are:-

1) A string descriptor of the name of the area (not at present implemented) or zero if the area is only to be referenced by number.

2) The size of the area as an integer number of bytes. If this is zero the area is of unspecified size not exceeding one segment.

3) A descriptor pointing to a 32-bit variable. A result planted in this variable is an integer which can subsequently be used to reference the area. If the top of this parameter is zero the bottom 32 bits give a reference number input to redefine the area.

4) and 5) Area type and position as high level.

PROC SPEC AREA SIZE (I32, S)   (CTL15)

Given an area number (first parameter) area size yields (second parameter) the space so far allocated in the area as an integer number of bytes.

(o) SCALAR DEC and STRUCTURE DEC

These do not appear but are implemented by:-

PROC SPEC VARIABLES (I32, I32, I32, S)   (CTL17)

This is the basic declarative procedure which allocates a sequence of consecutive names or vector elements of a given size within the local namespace, absolute namespace or within a specified area. The output parameter of this procedure is the displacement of the first name so declared from namebase, the start of the name segment, or from the start of the area, respectively. This displacement is in 8-bit units. If a set of names is declared names start at intervals of 4, 8 or 16 units when the declared name size is 32, 64 or 128 bits, respectively. If the declared size is less than 32 bits, i.e., 1 bit, then the displacement is correctly scaled and each subsequent element starts 1 unit further on. Equivalencing is not permitted in this case. Boolean variables take up 32 bits each in the name segment.

The parameters are:-

1) A 32-bit integer containing the size of each declared variable.

0	=	1 bit	
2	=	4 bits	(for vector
3	=	8 bits	or string elements
4	=	16 bits	not for names)
5	=	32 bits	
6	=	64 bits	
7	=	128 bits	

2) The number of variables being declared.

3) The number of the area to which they are allocated.

0	=	Local
-1	=	Absolute
$1-2^{31}$	=	Declared areas

4) A descriptor pointing to a 32-bit integer in which the displacement is planted.

(p) PROC SPEC PROC SPEC (S, I32, S, S) (CTL18)

The first parameter becomes a descriptor of the name of the procedure or zero.
Second parameter is type and result mode as before.
Third parameter is parameter specification as before.
The last parameter yields the procedure reference number.
If top = 0, bottom = reference number of a respecified procedure.

PROC SPEC PARAMETRIC PROC SPEC (I32, I32, S, S) (CTL34)

This is to give the specification of a parametric procedure.
The first parameter (I32) is the parameter number (i.e., displacement/NB) within, and relative textual level of the surrounding procedure heading. Other parameters as for PROC SPEC.

(q) PROC (I32, S) (CTL19)

In the procedure heading the internal identifier is replaced by the reference number yielded by PROC SPEC. No information about parameters is given, but the descriptor parameter is the same as the third parameter of CTL.PROC.

(r) END () (CTL20)

Similar to high level procedure except that no name and property list organisation takes place.

(s) RESULT IS (S) (CTL21)

Same restrictions on parameters as for computation.

(t) RETURN () (CTL22)

Identical to high level procedure.

(u) BLOCK (S) (CTL23)

Similar to the high level procedure except that no name and property list organisation takes place. Parameter as first parameter at high level.
The BEGIN which does not create a namespace does not appear at this level, nor does the corresponding END.

(v) CALL (S) (CTL24)

Restrictions on parameter as for computation.

(w) STORE DISPLAY (I,S) (CTL25)

Restrictions on operand as for computation.

(x) GET DISPLAY (I,S) (CTL26)

Restrictions on operand as for computation.

(y) TASK (S,I) (CTL27)

Restrictions on first parameter as for computation.

(z) PRELUDE () (CTL28)

As high level.

(aa) END PRELUDE () (CTL29)

As high level.

(bb) SET INTERRUPT MASK (I) (CTL30)

As high level.

(cc) RESET INTERRUPT MASK (I) (CTL31)

As high level.

(dd) TEST OVERFLOW (I,I) (CTL32)

The second parameter is now a label operand.

(ee) UNSTACK (I,I) (CTL33)

As high level.

(ff) STACK LINK (I) (CTL35)

Restrictions on operand as for computation.

(gg) EXIT STACK () (CTL36)

As high level.

(hh) PREPARE PROC ENTRY (I,S) (CTL37)

Restrictions on operand as for computation.

(ii) ENTER PROC (S) (CTL38)

Restrictions on operand as for computation.

(jj) VARIABLE PARAMETERS () (CTL39)

As high level.

(kk) CODE STREAM (I) (CTL40)

As high level.

(ll) DATAVEC (I32, S, I32, S) (CTL41)

First 2 parameters as parameters 2, 3 at high level. Third parameter is area number. Fourth parameter yields a descriptor which may be used to reference the datavec. If Parameter 3 is zero, the datavector is placed in the code segment with a jump round it.

(mm) POSTLUDE () (CTL42)

As high level.

(nn) PROC.PRELUDE () (CTL43)

As high level.

(oo) COMMON.PROC.SPEC (I32, S, S, S) (CTL44)

Yields reference number. First 3 parameters as CTL18 parameters 2, 2, 3 except that fourth parameter is descriptor of system name of procedure.

(pp) END.COMPILE () (CTL45)

As high level.

(qq) EXIT.JUMP (S) (CTL46)

As high level.

(rr) CTL.PLANT.INST (I32)

The parameter contains, in the bottom 16 bits, a machine code instruction which is placed at the current position in the code.

(ss) DATA (S) (CTL47)

As high level, except that named literals are not allowed.

5.9 Procedures for Operating on Name Lists and Property Lists

The namelist is a mechanism provided by the CTL which associates a string of characters with a unique internal identifier. The internal identifier is a 32 bit integer. For the convenience of high-level language translators the top 2 bits (at least) of an internal identifier are zero. The property lists contain information about identifiers. In general, there is a set of properties for each declarative occurrence of a particular identifier. These sets are chained together in a list the start of which is given by the value associated with each identifier (which is not the identifier itself). Each set of properties contains some standard information used by the CTL and possibly also some additional properties used by an individual compiler.

Names will normally be replaced by the corresponding internal identifiers at the lexical analysis phase of compilation by calling:-

a) PROC SPEC CTL.ADD.NAME (S, S)

The first parameter is the descriptor of the string of 8 bit characters which form the name. Any 8-bit combination may be used in a name so the names used by a compiler are not restricted in any way. (However, if the CTL is being used to produce Autocode program, names may contain any visible symbol except ?. The name <NAME> will then be output as N?<NAME>? and this symbol string translated back to <NAME> on re-input of the program). The procedure looks up the name on the namelist and adds it if it is not already present. The namelist is independent of any block structure which may be reflected in the property lists. The second parameter describes a 32-bit quantity in which the corresponding internal identifier is placed.

Although the following procedures may be used by compilers they are provided primarily for use by the CTL declarative and code planting procedures.

b) PROC SPEC CTL.GENERATE.NAME (S, I32)

Given an internal identifier (second parameter) this procedure generates the descriptor of the corresponding character string. Thus the external representation of a name can be regenerated.

c) PROC SPEC CTL.START.BLOCK (S, S)
PROC SPEC CTL.START.PROC (I32, S)

Start block adds the standard block entry properties to the property list. All new sets of properties added are then within this block. The first parameter is the descriptor of an I32 variable to which the block identifier used in re-enter block is returned. The second parameter is the string descriptor of any private properties to be added. Start proc performs the same function for a procedure, assuming that a procedure specification has already been given. The first parameter is the internal identifier of the procedure. The second parameter is a descriptor of a vector of parameter identifiers. These procedures are required in the first pass of a two pass compiler which creates properties in the first pass and code in the second.

d) PROC SPEC CTL.REENTER.BLOCK (I32)

When the end of a block is reached the links from each identifier to the properties within that block (SAME NAME link) may be removed. In these circumstances RE ENTER block re-establishes the links. The parameter is the block identifier.

e) PROC SPEC CTL.END.BLOCK (I32)

The current block becomes the block which textually surrounds the block just ended so that new sets of properties are associated with this block. If the parameter is zero this is the only action. If the parameter = 1 then the link from IDEN (SAME NAME link) to the properties in this block is removed for each set of properties within this block. Use of this mode may speed up access to properties of non-local quantities.

f) PROC SPEC CTL.ADD.PROPERTIES (I32, S, S)

The I32 parameter is an internal identifier. The second parameter describes a set of CTL properties and also yields the address of the properties as remaining additional properties. The third parameter describes any private properties. A new set of properties for the I32 parameter, the vector described by the descriptor parameter is added to the property list. If there is already a set of properties in the current procedure or block a fault number is generated and the new properties added as well, unless the parameter II is negated (not added). If II = 0 PROPERTIES are created with no identifier. If the top bit of the second parameter is set, the properties are added at the global level.

g) PROC SPEC CTL.FIND.PROPERTIES (I32, S)

Given an internal identifier, first parameter, the set of properties within the current block, if such a set exists, or within the first enclosing block is located. The output parameter is then the position of this set within the property list vector or zero if no properties exist.

Once such a position is determined the calling procedure may then manipulate the property set in any way.
A similar procedure which looks up only locally is:

CTL.FIND.LOCAL.PROPERTIES (I32, S)

h) PROC SPEC CTL.FIND.N.PROPS (S, S, S)

This combines the action of ADD NAME and FIND PROPERTIES.
Parameters are:-

(i) Descriptor of name
(ii) Internal identifier output
(iii) Property list position output

Form of Property List Entries

A property list entry comprises some links, 32 bits of standard properties used by CTL and additional information. Sometimes additional properties are required by the CTL and sometimes by a translator for its own purposes.

The general form of such an entry is:-

(i) pointer to next property set for same name (SAMENAME)
(ii) pointer back to name list entry (IDEN LINK)
(iii) pointer to next item in current block (SAME BLOCK)
(iv) pointer to surrounding level (block or procedure) entry (LEVEL)
(v) pointer to additional CTL properties (if any)
(vi) pointer used by CTL - usually to area to which declarative refers
(vii) CTL properties (32 bits)
(viii) user private properties (if any)
(ix) CTL additional properties (accessed via pointer (v))

Since the size of the pointers (i) to (vi) is implementation dependent and may vary according to the number of sets of properties required it is essential to access information and pointers by the descriptors provided and not to rely on a particular implementation.

A 32-bit modifier such as the output parameter of CTL.FIND.PROPERTIES may be used in conjunction with the following descriptors:-

(i)	SAME.NAME	accesses pointer (i)	above	(V64 16/0)
(ii)	IDEN.LINK	accesses pointer (ii)	above	(V64 17/0)
(iii)	SAME.BLOCK	accesses pointer (iii)	above	(V64 18/0)
(iv)	LEVEL	accesses pointer (iv)	above	(V64 19/0)

(v) ADD.PROP.PTR obtains a new modifier which may then be used in conjunction with the ADD.PROP, ADD.PROP16 descriptors (same as PP32, PP16) to access the CTL additional properties (V64 20/0)

index

(vi) AREA accesses pointer (vi) above (V64 21/0)

(vii) CTL.PROP accesses 32 bits of CTL properties above (V64 22/0)

(viii) The private properties are regarded as a series of bytes which may be accessed in 1, 2 or 4 byte groups.

	PP8	accesses the first byte	(V64 23/0)
ADDPROP16	PP16	accesses the first two bytes	(V64 24/0)
ADDPROP	PP32	accesses the first four bytes	(V64 25/0)

Further private properties are accessed by adding a byte displacement from the first byte to the modifier and using one of the above descriptors.

TYPES

1	bool
2	Fix sign
3	Fix unsign
4	Float
5	Dec
6	Subscr.
7	Desc.
8	Lab
9	Proc
10	Src
11	Dst
12	Compl.
13	STS
14	Long

Use of the CTL Basic and Additional Properties

The 32 bits of CTL properties are usually divided as follows:-

(i) Top 4 bits: type of property list entry

(ii) Next 8 bits: mode i.e., type and size of quantity. Usually only size is checked but type may be used by translator. The most significant of these bits is used in run time fault monitoring. If it is set this variable is not printed.

(iii) Next 2 bits: Whether variable is local(0), absolute(1) or in a declared area(2), or large displacement(3).

(iv) Final 18 bits: Displacement in byte units of the variable from the start of the relevant name space or area. [For a 64 bit area name, the displacement is in the first additional property for a scalar, the fourth for a descriptor].

Types of property list entry:-

0 SCALAR: properties as above.
No additional properties. Area pointer (vi) is only used when the scalar is in a declared area.

1 DESCRIPTOR: as for scalar.
4 32 bit additional properties are defined as follows:-
(1) element size (or mode) (tagged negative if vector is dynamic)
(2) displacement of first element from start of area
(3) area within which elements are mapped out
(4) descriptor displacement, if > 18 bits.

2 LABEL: basic properties reserved for use by loader in a load and execute situation. The bottom bit is zero when the label is defined, 1 when the label is referenced only. AREA holds the low level label number.

3 UNDEFINED

4 PROCEDURE or BLOCK: Basic properties are reallocated:-

(i)	'PROCEDURE or BLOCK' (4 bits)
(iia)	Type of procedure or block (4 bits)

This is specified as follows:-
0 = procedure (general recursive)
1 = subprocedure
2 = static procedure
3 = procedure with variable numbers of parameters
4 = Block
5 = Begin
6 = Built in procedure (e.g., addr)
7 = Macro procedure (MU5 Autocode)
8 - 15 Similar to 0 - 7 for COMMON procedures

(iib) and (iii) Textual level (6 bits)
(iv) Current displacement used in allocating names within the procedure namespace (18 bits)

The area pointer position is used to hold the block number.
Additional properties:-

(i) Result mode (14 bits)
and (ii) Maximum displacement in namespace for allocating names (18 bits). This is set zero on procedure specification, but will be non-zero after procedure definition.
(iii) Low level reference number (16 bits)
(iv) Number of parameters (16 bits)
(v) Start of new SAME BLOCK links
(vi) Position at which ASF is held.

5 PARAMETER as for scalar.
An additional property is used to hold the type bits of the parameter. In the case of a procedure (or undefined) parameter another additional property holds a pointer to the relevant procedure spec entry, if any (= 0 if none).
The bottom 8 bits hold the user's parameter mode, for fault monitor printing.

6 MAPPING POINTER or AREA
Bit %80(6) is set for an area declared as type 3 (accessed by setting XNB to an absolute name).
Displacement = current displacement in bytes from start of area.
AREA holds the low level area number.
Note that all area names are global.
There is a 32 bit additional property containing the origin of the area (in bytes).

7 NAMED LITERAL
The size field indicates whether the literal itself in additional properties is held in 32, 64 or 128 bits of additional property.
The type field gives the form of the constant.
If there are no additional properties, then the literal is undefined, but has been referenced from within a datavector.

8 RESERVED NAME
Use of this depends on the language in use. For the current autocode implementation these are names which should not be redeclared.

9 SWITCH
Area contains low-level reference number. No additional properties.

10 - 14 Spare

15 User defined.

Chapter 6 Further Examples of CTL

This chapter contains a number of additional examples of the use of the CTL.COMPUTATION facility. These examples have been constructed primarily for use in CTL test programs. They mostly use the basic operand type 2 rather than internal identifiers. Each example consists of an autocode statement followed by the corresponding vector handed as a parameter to CTL.COMPUTATION. In general there is no correspondence between the autocode names used for operands and the actual operands used. The vectors are coded in hexadecimal, one element per line with some comments on the right hand side of each line.

```
I32,N1+N1*N2 V N3

80150002
00000008
00010002
00000008
00030002
00000010
000A0002
00000018
00360000
```

```
I32,A+[R64,B+C]

80150002
00000008
00010022
80260002
00000008
00010002
00000010
00270020
00360000
```

```
I32,B[I]

80150802
00000004
002F0002
00000008
00310020
00360000


I32,B[I+J]+C

80150802
00000004
002F0002
00000008
00010002
0000000C
00310020
00010002
00000010
00360000


I32,B[I][J][K]

80150802
00000004
002F0002
00000008
00300002
0000000C
00300002
00000010
00310020
00360000
```

I32,B[I][J][K] (alternative form of coding)

```
80150802
00000004
002F0002
00000008
00310020
002F0002
0000000C
00310020
002F0002
00000010
00310020
00360000
```

I32,B[C[D]]

```
80150802
00000004
002F0802
00000008
002F0002
0000000C
00310020
00310020
00360000
```

I32,B[C[D]]+E

```
80150802
00000004
002F0802
00000008
002F0002
0000000C
00310020
00310020
00010002
00000010
00360000
```

I32,5+(-?)+ 7F

```
8015000D
00000005
0001000D
FFFFFFF1
0001000D
0000007F
00360000
```

R64,C1+C2+C3 (64 bit constants)

```
8026000E
00000000
00000001
0001000E
FFFFFFFF
00000000
0001000E
0000FFFF
12345678
00360000
```

I32,C1+C2 (32 bit constants)

```
8015000D
12345678
0001000D
00005432
00360000
```

I32,A[CONST]

```
80150802
00000008
002F000D
00000012
00310020
00360000
```

```
A[DV(B, I, J)]

80150802
00000008
00280002
00000008
00290002
00000010
002A0002
00000018
002B0020
00360000


A[DV(B, I, J)][K]

80150802
00000008
00280002
00000008
00290002
00000010
002A0002
00000018
002B0020
002F0002
00000020
00310020
00360000


A IND B

80150802
00000008
00320002
00000010
00360000
```

```
A[B]INDC

80150802
00000008
002F0002
00000010
00310020
00320002
00000018
00360000
```

```
I32,IF[I32,A=D]THEN B ELSE C

80150023
001A0022
80150002
00000010
001E0002
00000020
001D0001       IF /=
00000005 ──┐
00250002   │   THEN B
00000030   │
00240001   │   SKIP
00000003─┐ │
00250002 <─┼─┘ ELSE C
00000040 <─┘
00360000
```

```
I32,IF B THEN P ELSE Q

80150023
00190002
00000020
00230001
00000005 ──┐
00250002   │   THEN
```

```
00000030            P
00240001
00000003 --+        SKIP
00250002 <-|----    ELSE Q
00000050   |
00360000 <-+
```

I32,IF(IF[A=B]THEN[C=D]ELSE Q)THEN X ELSE Y

```
80150023
00190023            IF
001A0022            IF
80150002            I32 A=B
00000010
001B0002
00000020
001D0001
0000000A ------+
00250023       |
001A0022       |
80150002       |    THEN C=D
00000030       |
001B0002       |
00000040       |
001D0025       |    SET BNIF C=D
00240001       |    SKIP
00000003 --+   |
00250002 <-|---+
00000050   |        ELSE Q
00230001 <-+        IF FALSE SKIP
00000005 ------+
00250002       |    THEN X
00000060       |
00240001       |
00000003 --+   |
00250002 <-|---+
00000080   |        ELSE Y
00360000 <-+
R64,  1+(IF[I32,1=2] THEN   3 ELSE   4)

8026000E
00000000
00000001
00010023
001A0022
8015000D
00000001
001B000D
00000002
001D0001
00000006
0025000E
00000000
00000003
00240001
00000004
0025000E
00000000
00000004
00360000
```

```
R64,  1+(IF[I32,1=2] THEN   3 ELSE   4) -   5

8026000E
00000000
00000001
00010023
001A0022
8015000D
00000001
001B000D
00000002
001D0001
00000006
0025000E
00000000
00000003
00240001
00000004
```

```
0025000E ←
00000000
00000004
0002000E ←
00000000
00000005
00360000
```

R64, 1+(IF[I32,1=2] V [I32,2=1] $ NOT THEN   5 ELSE   4)

```
8026000E
00000000
00000001
00010023
00190023        BOOLEAN CONDITIONAL
001A0022        RELATION
8015000D
00000001
001B000D
00000002
001D0025
000A0023        OR
001A0022        RELATION
8015000D
00000002
001B000D
00000001
001D0025
00220001        IF TRUE
00000006
0025000E
00000000
00000005
00240001
00000004
0025000E ←
00000000
00000004
00360000 ←
```

```
I32,1+IF[2=3] THEN 4 ELSE 5

8015000D
00000001
00010023
001A0022
8015000D
00000002
001B0001
00000003
001D0001
00000005
0025000D
00000004
00240001
00000003
0025000D←
00000005
00360000←
```

```
I32,IF [2=3] THEN 4 ELSE 5

80150023
001A0022
8015000D
00000002
001B000D
00000003
001D0001
00000005
0025000D
00000004
00240001
00000003
0025000D←
00000005
00360000←
```

I32,(IF 2=3 THEN 4+7 ELSE 5+8)+9

```
80150023
001A0022
8015000D
00000002
001B000D
00000003
001D0001
00000007 ───┐
0025000D    │
00000004    │
0001000D    │
00000007    │
00240001    │
00000005 ─┐ │
0025000D <┼─┘
00000005  │
0001000D  │
00000008  │
0001000D <┘
00000009
00360000
```

BN,(IF TRUE $ NOT THEN TRUE & TRUE ELSE TRUE V TRUE) V FALSE

```
80080023
0019000D
00000001
00220001
00000007 ───┐
0025000D    │
00000001    │
000B000D    │
00000001    │
00240001    │
00000005 ─┐ │
0025000D <┼─┘
00000001  │
000A000D  │
00000001  │
000A000D <┘
00000000
00360000
```

BN,TRUE & (IF TRUE $ NOT THEN TRUE & TRUE ELSE TRUE V TRUE) V FALSE

```
8008000D
00000001
000B0023
0019000D
00000001
00220001
00000007
0025000D
00000001
000B000D
00000001
00240001
00000005
0025000D
00000001
000A000D
00000001
000A000D
00000000
00360000
```

Chapter 7 Fault Handling

7.1 Compile Time Faults

A library procedure, OUTF(I,S,I), is provided to print compile time fault messages for all compilers. The parameters are as follows:-

P1	The fault number, which should be in the range 1-200. Negative numbers are reserved for faults detected in the organisational procedures. Fault numbers > 200 represent input/output fault messages. Fault numbers > 300 represent run time fault messages.
P2	Descriptor of a message string.
P3	Page (top 16bits) and line number.

The page/line number is first output, if P3 is non-zero (if P3 = 0, spaces are output). '$\alpha\alpha$' is then output followed by the string of characters given by P2, if P2 is non-zero. Finally the fault message, as defined overleaf, is printed, followed by a newline. Note that, if indentation of the fault messages is required, the necessary spaces should be included in the string P2. OUTF also decrements the segment zero location FAULTY. (V32 63/0).

A similar procedure, OUTM(I,S,I), is used to print warning messages. The message is output preceded by '??'. OUTM does not decrement FAULTY.

Compile Time Fault Messages

1. Unusual fault
2. Statement not recognised
3. Illegal expression
4. Facility not implemented
5. Too many ends
6. Too few ends
7. <P> faulty
8. Delimiter not recognised
9. String terminator missing
10. Statement out of context
11. Illegal number
12. Integer too big
13. Illegal LHS of assignment
14. Subscript out of range
15. Exponent out of range
16. Library proc name undefined
17. Subscript/parameter faulty
18. Return required
19. <P> not declared
20. <P> declared twice
21. Literal of wrong mode
22. Operand of wrong size
23. Wrong number of repeats
24. Wrong number of continues
25. Redundant goto
26. <P> Invalid operand
27. Too many elses
28. Datavec too short
29. <P> is undefined
30. <P> of wrong type/size
31. <P> has wrong number of subscripts
32. <P> has wrong number of parameters
33. <P> inconsistently dimensioned
34. <P> wrongly subscripted
35. <P> in wrong context
36. Unsatisfied references
37. Compilation faulty
38. Trap 4 entered. PWA gives reason
39. Illegal FN-OP combination

40. CTL detected fault
41. Statement out of order
42. Operator of wrong type
43. <P> used inconsistently
44. Delimiter comment
45. Cycle across group
46. Wrong no. of ('s or)'s (or modulus)
47. Faulty symbol constant
48. Wrong no. of finishes
49. Parameter inconsistency
50. Illegal loop parameters
51. Illegal use of 'OWN'
52. Dynamic bounds illegal
53. Specification missing
54. Invalid start of program
55. Faulty bound pair list
56. Faulty string
57. Faulty array subscript
58. Inconsistent parameter specification
59. Inconsistent type in expression
60. Comment follows end
61. Hollerith of wrong size
62. Loop stop
63. Statement requires label
64. <P> defined already
65. Do loop overlap
66. Expression of wrong type
67. I/O unit set twice
68. Inconsistent specification
69. <P> common size inconsistent
70. <P> missing do loop terminator
71. Wrong no. of variables in data
72. Not standard Fortran
73. Branch required before end
74. Statement cannot terminate do loop
75. <P> cannot be initialised
76. <P> not format label
77. <P> is do loop variable
78. <P> inconsistently typed
79. <P> cannot be in external

80. <P> not in external
81. <P> cannot be in common
82. <P> cannot be common block name
83. <P> inconsistently equivalenced
84. Comment contined
85. No main program unit
86. Two main program units
87. Label must be on executable statement
88. <P> cannot be implicitly declared
89. Illegal equivalence
90. Area overflow
91. Illegal conversion
92. Faulty area declaration
93. Too many textual levels
94. Faulty prelude
95. Faulty CTL computation vector
96. Illegal type/size in declaration
97. Illegal mode

7.2 CTL detected faults

If CTL detects a fault the user defined procedure given in CTL.INITIALISE is called. This has two parameters. The first is an I32 fault number. The second is a 64 bit quantity which gives further information for some faults, (the name, <P>, as required for CUT.F).

Most fault calls in CTL are followed by a RETURN to the user procedure which called CTL. (Thus the user fault procedure could set some fault indicator and return to CTL which returns to the user.) After some faults CTL does not return immediately but continues with the action specified under that fault.

Note that the numbers of CTL detected faults correspond to the numbers of the compile time fault messages given previously.

List of faults indicated by CTL

0. CTL fault
1. Unusual fault
3. Illegal expression
4. Facility not implemented
5. Too many ends
6. Too few ends
10. Statement out of context [Return not in a procedure]
14. Subscript out of range [in switch label declaration]
16. Library Proc name undefined
19. <P> not declared
20. <P> declared twice
23. Wrong no. of repeats
24. Wrong no. of continues
26. <P> Invalid operand [<P> is not given when fault detected at low level]
27. Too many elses
29. Datavec is undefined [<P> is 'Datavec']
30. <P> of wrong type/size [<P> may be 'area']
32. <P> has wrong no. of parameters
35. R/ in wrong context [<P> is 'R/']
36. Unsatisfied reference
42. Operator of wrong type
43. <P> used inconsistently
49. Parameter inconsistency
50. Illegal loop parameters
52. Dynamic bounds illegal
53. <P> Specification missing [<P> is not given when fault detected at low level]
59. Inconsistent type in expression [Faulty result mode]
64. <P> defined already [<P> may be 'label', 'switch label', 'proc']
68. Inconsistent specification
89. Illegal equivalence
90. Area overflow
91. Illegal conversion
92. Faulty area declaration
93. Too many textual levels
94. Faulty prelude
95. Faulty CTL computation vector
96. Illegal type/size in declaration
97. Illegal mode

7.3 Run time fault monitoring

Run time fault monitoring will be done by a library procedure for all the basic languages (AA, Algol, Fortran, MU5 Autocode). This procedure will use the CTL standard compile time property lists (or a condensed version, at the discretion of the system). It also uses a line table constructed at compile time. At the start of each line the CTL procedure:-

CTL.LINE (I32)

should be called. The parameter is the page (most significant 16 bits) and line number to be associated with the following instructions up to the next call of CTL.LINE. The current page/line number can be found by calling I.LINE ()I32.

Any untrapped run-time fault will cause an entry to trap 6, the default of which is to call CTL.FAULT.MONITOR (I32). This will print, on output zero the fault reason and the line/page number (if known) by calling OUT.F, after adding 300 to the fault number. (If no line table has been set, the control address is printed).

CTL.STACK.PRINT will then be called. This will start at the current block printing,

IN BLOCK <NUMBER>
or IN PROCEDURE <NAME>

The variables will then be printed in the form <NAME> = <value>. If a mode has been given in the declaration the value will be printed in that mode otherwise it will be printed in hexadecimal (and possibly fixed decimal). Descriptors will be printed in hexadecimal.

The printing of any variable is inhibited by setting the top of the (8) mode bits in the CTL properties to 1 (i.e. adding %80 to the mode when the properties are created).

If the properties are for a descriptor whose printing is inhibited but the element mode is given (in the first additional property) without printing inhibited then element 0 of the vector is printed in the mode specified.

For certain parameters a mode specification enables the parameter itself to be printed, see page 98.

If no properties are known for a procedure/block its namespace is printed in hexadecimal.

Runtime Fault Numbers (Reasons for Trap 6)

1. Call to an undefined procedure
2. Jump to an undefined switch element
3. Jump to an undefined label
4. Call to an undefined parametic procedure (or procedure variable)
5. Exponentiation of zero by zero or by a negative integer
6. Negative exponent in integer arithmetic
7. Undefined Software Function
8. Non-local jump out of a procedure with no postlude set
   (Could be caused by no RETURN in a procedure)
9. Non-local jump to a label in an inactive block
10. Assigned goto label not in list
11. Do loop parameter fault
12. Array format faulty
13. Unit no. not specified
14. Unit mode inconsistent
15. Strange char. in input/char. out of order
16. FD wrong type
17. End of record reached
18. No field descriptor
19. No T or F in logical read
20. Strange CCC char
21. Value too large for field
22. Exponent too large
23. Scale factor wrong size
24. Divide not implomented