UNIVERSITY OF MANCHESTER
DEPARTMENT OF COMPUTER SCIENCE

THE MUPL PROGRAMMING LANGUAGE

A slightly revised version of MUPL will be available in late 1979; see the MU6 MUPL Manual.

SEPTEMBER 1979

CONTENTS

Introduction

MUPL is the programming language used to implement the MUSS portable software system. The language is essentially one of the Algol-like class of languages in having a block structure, procedures with parameters, declared scalar and vector operands, high-level control structures and straightforward but powerful computational statements. These features of the language are machine independent.

There is also a series of low-level statements for each machine on which the language is implemented. Although these statements are broadly similar in different implementations specific machine architectures and instructions are reflected at this level. Thus statements at this level are, in general, machine dependent.

Chapter 1. Basic Elements of the Language

Character set

In the current implementations the language is represented in the basic 64 character card character set. The internal representation of these characters is implementation dependent. There is a set of delimiter words which are represented as $ followed by a letter string. Some operands are predeclared to represent machine registers in the low level form and must not be redeclared.

Operators

The allowed operators (meanings defined later) are:

```
+  -  -:  *  /  /:  <-  <=   !  -/=  =>
+>  ->  -:>  *>  />  /:>  >  !>  -/=>
```

and a number of low level operators

Delimiter words

```
$PROC , $PROCEDURE ,  $BEGIN , $END , $SWITCH , $VS ,
$VD , $VQ , $VV , $L , $IF , $ELSE , $FI , $IS , $ID ,
$IQ , $RS , $RD , $RQ , $LS , $LD , $LQ , $DS , $DD ,
$DQ , $ANDF , $ORF , $AT , $VEC , $F , $SPEC , $RETURN ,
$S , $CYCLE , $WHILE , $REPEAT , $UNTIL , $DATAVEC ,
$DATASTR , $VEC , $MOD , $SMOD , $TOP , $ORIG , $BOUND
```

and a series of low-level delimiters.

Separators

Statements are separated by ; or newline. In the rest of this manual, <NL> will indicate such a separator. When a delimiter word is followed immediately by a name, a space or tab separates the delimiter from the name. Otherwise spaces and tabs between items are ignored.

Continuation

When a statement continues over several lines, £ at the end of a line indicates continuation onto the next line.

Comments

All characters between :- and the following newline are ignored.

Predeclared operands (not all predeclared in all implementation)

AI, AR, AD, AL, NB, XNB, SF, DR, BM, DO, DT, DB, OV, PS, MS, TOS

Names

Names used to represent variables, literals and labels consist of a letter followed by any number of letters and digits possibly separated by fullstops. The latter are for readability only and have no significance in the meaning of names.

Literals

```
<LITERAL> ::= <NAME> ! <CONST> ! $F <NAME>
<CONST>   ::= <DEC.INTEGER>!
              %<HEX>!
              "<CHARS>"!
              %"<CHARS>"!
              !<DESCRIPTOR LITERAL>!
              <FLOATING.CONST>
<DESCRIPTOR LITERAL> ::=  <LITERAL>/<LITERAL>/<LITERAL>
```

Examples

```
123
%1F(6)
"THIS IS QUAD LIT"
 0/100/  %1FF
```

If a literal is a <NAME> the name must have a previously defined value, e.g. as a previously declared named literal, label or literal descriptor (datavec or datastring name). Literals are never scaled so particular care must be taken in the use of label and descriptor literals.

If a literal is $F <NAME>, the value yielded by FINDN for that <NAME> is used.

<HEX> represents a series of hexadecimal digits (0 to F) up to the length required. Hexadecimal constants are right justified and zero filled on the left. If a particular hex digit is repeated it may be followed by a count, in brackets, giving the total number of times the digit is to occur.

Descriptor literals comprise type/bound/origin. Type and bound must be previously defined (single length) literals or constants but origin may be a forward reference to a label. The representation of descriptors is machine dependent but for simple types of descriptors (e.g. 0 and 40) the bound and origin fields will be interpreted appropriately for the target machine.

In character literals <CHARS> is a sequence of characters, not including " (i.e. double quote), but possibly including !<HEX PAIRS>! to denote unrepresentable characters. The HEX PAIRS are given in the character code of the compiling machine and will be converted by the compiler to the character code of the target machine. %"<CHARS>" denotes a string literal, and yields a descriptor to the string of characters.

Integers are always evaluated as single length but <HEX> and <CHAR STR> constants may be to single, or double or quadruple length. Floating constants not yet defined.

Directives

These are of the form:
* <text>
where the possible forms of <text> are given below.

*SEGMENT <CONST>[,<CONST>!<NIL>]

If the program is to be compiled into or for a segment other than the default this directive is used. The first <CONST> is the run-time segment number, the second the compile time if different. Several segments may be created in a single compilation, in which case the compiler remembers the current position in each segment and restores it correctly on a *SEGMENT directive.

*LINE <CONST>
Gives new compiling position in current segment in VB (byte) units.
*END

End compile.

*NL <CONST> , <CONST>
Segment number for library directory if not standard.
*PRINT ON lists compiled code for low-level statements
*PRINT OFF (default)
*MAP ON lists all block and procedure headings.
(default - offline)
*MAP OFF (default - online)
*GLOBALS <CONST> see chapter 3.2
*CONST[Q!D!S] alters the default mode for evaluation of constants.
*CONSTANTS <CONSTS> , <CONST>
Gives start of constant pool (in code segment) in VB units and size in VS units. (Only needed in implementations with a constant pool).

*PRINT NAME LIST Prints out compile time namelist.

*COMPILE IF <CONST>

The statements between this directive and the next occurrence of *COMPILE IF are only compiled if the relevant bits in the compiler mode are set (see MUPL Implementation Manual). Compiler mode is set for the machine on which the compiler is running, unless the user calls the MUPL compiler with a non-zero compiler mode parameter. The significance of the constant is as follows:

0	=	unconditional compilation
1	=	2900
2	=	MU5
4	=	PDP11
8	=	MEMBRAIN MB7700
16	=	MOD 1
32	=	spare
64	=	user defined
128	=	user defined

Combinations of these constants may be used to compile for a selection of machines,

e.g. *COMPILE IF 3

compiles the following statements for MU5 and 2900 only while,

*COMPILE IF 0

resumes unconditional compilation.

*OPT [ON ! OFF] <CONST>

Sets the <CONST>th bit (l.s.e = bit 0) on or off of the compiler mode. (The compiler mode is set automatically to a suitable value for compiling and running jobs, and should only require altering for special system purposes - see MUPL Implementation Manual).

*LIB [<CONST>!<NIL>]

This directive is used to create a private library. The <const> is the number of entries in the private library directory. If absent a default of 48 entries is used. The current compiling segment is used as the library segment, thus the *SEGMENT directive should be used to obtain an empty compiling segment. The procedures to be contained in the library must be enclosed by a $BEGIN and $END statement. They must be full $PROCs not $SUBPROCs. No initialisation of private libraries occurs and so the vector segment should not be used (see 3.3 and 3.4). To update the private library into the system, the library procedure DEFINE should be called at the end of compilation.

Example:

```
MUPL
*SEGMENT -1
*LIB
$BEGIN
    $PROC LISTFILE ($LD NAME, DEVICE)
    $END

$END
*END
DEFINE MYLIB
```

*PROG

This allows a program to be filed as binary by the library procedure DEFINE. As with *LIB, it must be given at the beginning of the program after a *SEGMENT directive.

Chapter 2. Organisation of Procedures and Blocks.

2.1 PROGRAMS.

```
<PROGRAM>   ::=   [<BLOCK>!<PROC>]<NL>*END
```

A MUPL program comprises a block or a procedure declaration, which may in turn contain nested blocks and procedure declarations.

2.2 Blocks.

```
<BLOCK>       ::=   $BEGIN<NL>
                    <STATEMENTS><NL>
                    $END
<STATEMENTS>  ::=   <STATEMENT><NL>[<STATEMENTS>!<NIL>]
<STATEMENT>   ::=   <DECLARATION>!
                    <BLOCK>!
                    <PROC>!
                    <HIGH.LEVEL.STATEMENT>!
                    <LL.STATEMENT>
```

Blocks are used simply to delimit the static scope of names declared within them. Entry to a block is always through its first instruction, and exit is normally via the last instruction, though non-local jumps from blocks are permitted. No instructions are compiled for the $BEGIN or $END of a block except when the block is the whole program. Names declared within a block are appended to the namespace of the immediately enclosing procedure.

<DECLARATION> is defined in Chapter 3, <HIGH.LEVEL.STATEMENT> in Chapter 4 and <LL.STATEMENT> in the relevent machine dependent Chapter.

2.3 Procedures.

```
<PROC>         ::=   <PROC.HEAD><NL>
                     <STATEMENTS><NL>
                     $END
<PROC.HEAD>    ::=   [$PROC!$PROCEDURE!$SUBPROC]<NAME>
                     [(<PARAM.LIST>)!<NIL>][<RESULT>!<NIL>]
<PARAM.LIST>   ::=   <PARAM>[,<PARAM.LIST>!<NIL>]
<PARAM>        ::=   [<TYPE>!$S!<[>[<ELSIZE>!<NIL>]]<NAME>
<RESULT>       ::=   <TYPE>!$S
<TYPE>         ::=   See Chapter 4.
<ELSIZE>       ::=   See Chapter 3.
```

A procedure is entered at its first instruction as a result of a procedure call statement (See Chapter 4), and exit from a procedure is via a $RETURN statement which returns control to the point immediately after the call. Non-local jumps from procedures are allowed, but do not automatically tidy up the stack properly. A $RETURN is planted automatically at the end of a procedure. There are three types of procedure, known as:

SUBPROCs,
PROCs,
and, PROCEDUREs.

2.3.1 SUBPROCS

The sub-proc is the simplest form of procedure, and corresponds to a named block. It may not have parameters, and any locally declared variables are appended statically to the namespace of the textually enclosing PROC or PROCEDURE. (Note: this means that a SUBPROC which declares local variables should only be called from the PROC or PROCEDURE in which it is declared, and not from any enclosed procedures). As a result the sub-proc may use a simpler (and faster?) call/return sequence than the more general forms of procedure.

2.3.2 PROCS.

The PROC is the most commonly-used procedure form. It may have parameters and local variables, and a new namespace is created for these dynamically on entry to the procedure. To minimise the call/return over- heads, the local variables and parameters of a PROC may not be accessed by textually enclosed PROCs or PROCEDURES.

The procedure heading for a PROC specifies the types and names of its formal parameters, and the type of its result if any. Any of the arithmetic types may be specified (See Chapter 4), together with $S meaning a descriptor. Scalar parameters are passed by value, and descriptor parameters must be used explicitly if arrays or references to scalars are to be used.

In a functional procedure, the result is obtained by assignment to the procedure name in the correct mode, immediately followed by $RETURN or the end of the procedure. The result is undefined if the $RETURN does not follow immediately, or the mode is incorrect.

2.3.3 Procedures

The PROCEDURE is the most general form of procedure. It is identical to the PROC, except that non-local access to its variables from textually- enclosed PROCS and PROCEDURES is permitted . Note that in general this will result in an overhead both on procedure call/return, and on the non-local variable accesses.

Chapter 3. Declarations.

```
<DECLARATION>   ::=   <PROC.SPEC>!
                      <VAR.DEC>!
                      <VECTOR.DEC>!
                      <LIT.DEC>!
                      <DATAVEC>!
                      <DATASTR>!
                      <SWITCH>!
                      <LABEL>
```

With the exception of labels, all names in a MUPL program must be declared before they are used. The various types of declaration are described below:

3.1 Procedure Specifications

The procedure heading has already been described in Chapter 2, and gives the name of the procedure, the types and names of its formal parameters, and the type of its result. <PROC.SPEC> is an optional duplication of the procedure heading elsewhere in the program, which enables the procedure to be called before it is declared. The syntax is identical to that for a <PROC.HEAD> except that the word $PROC ($SUBPROC,$PROCEDURE) is followed by $SPEC.

Where a procedure spec is used, the procedure itself must be declared at the same textual level . The procedure must also be declared in the same code (execution) segment as the procedure was specified.

3.2 Variable Declarations

```
<VAR.DEC>              ::=  <SIZE><BASE><NAME.DISP.LIST>
<SIZE>                 ::=  $VB!$VS!$VD!$VQ!$VV
<BASE>                 ::=  /0,!/<NAME>,!,!<NIL>!/<BREG>
<NAME.DISP.LIST>       ::=  <NAME.DISP>[,<NAME.DISP.LIST>!<NIL>]
<NAME.DISP>            ::=  <NAME>[:<LITERAL>!<NIL>]
```

$VB, $VS, $VD, $VQ create byte, single, double, and quadruple length variables respectively. $VV creates a V-store name; the size and interpretation of V-store names is machine-dependent.

The <BASE> indicates the addressing environment in which a variable is to be declared. The default is local - i.e. the variable is appended to the local namespace of the immediately enclosing $PROC or $PROCEDURE '/0' indicates a global variable; such variables may be accessed in any environment. '/<NAME>' is used to declare variables equivalenced on to a previously-declared variable. Such variables must not be larger than <NAME>, or extend beyond the variables already declared.

Other, machine-dependent bases may also be specified by using '/<BREG>' Here, <BREG> is a register of the target machine which may be used as a name base.

The displacement of variables from the specified base is by default chosen automatically by the compiler, but the compiler variable controlling the position of global variables may be altered by the directive:

*GLOBALS <CONST>

In addition to this, a displacement ':<LITERAL>' may be given for individual variables, without changing the compiler variable. The <LITERAL> gives the displacement in $VS units, and must be even for double and quadruple length variables.

Examples of Variable Declarations:

```
$VS,   I, J, K                    :-  LOCAL, SINGLE-LENGTH
$VD/0, FILE.NAME                  :-  GLOBAL,DOUBLE-LENGTH
$VS/FILE.NAME, N1, N2             :-  EQUIVALENCED ON
                                      FILENAME
$VV,   V.DRUM.CONTROL             :-  V-STORE
$VD/0, PWW0:0, PWW1:2, PWW2:4     :-  GLOBAL, POSITION
                                      SPECIFIED
$VS/0, PW0:0, PW1:1, PW2:2, PW3:3
```

3.3 Vector Declarations

```
<VECTOR.DEC>      ::=  $VEC <BASE><NAME.DISP.LIST>
                       [<ELEMENT.SPEC>!<NIL>]
<BASE>            ::=  see 3.2 above
<NAME.DISP.LIST>  ::=  see 3.2 above
<ELEMENT.SPEC>    ::=  <[><ELSIZE>[<LITERAL>[$AT<LITERAL>!
                       <NIL>]!<NIL>]<]>
<ELSIZE>          ::=  $VB!$VS!$VQ!$VD!$V1!$S
```

A vector declaration creates and initialises a descriptor, and creates space for the vector elements. The declaration of the descriptor is similar to that for scalars. This may be followed by a specification of the vector itself, giving the element size, number of elements, N, and optionally the starting address. The vector elements are numbered 0 - N-1. In addition to the sizes allowed for variables, vectors may have 1-bit elements ($V1) byte elements ($VB) or descriptor elements ($S). Space for the elements will be allocated statically in a segment created by the compiler unless the $AT <LITERAL> option is used. In this case the elements of the first vector declared start at the byte address specified, and the user must ensure that the segment is created.

Normally only one segment is allocated for a vector segment. When two or more segments are required, the runmode parameter of MUPL should be set to the value of:-

8 + number of segments required.

There is a maximum limit of 7 segments for vector usage.

If the <ELEMENT.SPEC> is omitted or comprises solely <ELSIZE> , an un-initialised descriptor variable suitable for referencing the type of elements specified is created.

Examples of Vector Declarations:

```
$VEC,  V1,V2 [$VS 10]                    :- LOCAL,  0..9
$VEC,  S1,S2                             :- LOCAL DESCRIPTORS
$VEC/0, X1,X2,X3[$VS 100]                :- GLOBAL, 0..99
$VEC/0, SST2 [$VS 1024 $AT 1000]         :- STARTING ADDRESS
                                            SPECIFIED
$VEC/0, DESC:32 [$VB 24]                 :- DESCRIPTOR POSITION
                                            SPECIFIED
```

3.4 Table Declarations

```
<TABLE>          ::= $TABLE<NAME><[><LITERAL>[<ADDRESS>
                     !<NULL>]<]><NL><FIELDS>
                     <EXTENDEDTABLE>!$END

<ADDRESS>        ::= $AT [<SEGMENT>/<LINE>!<LITERAL>]
<SEGMENT>        ::= <NAME>!<CONST>
<LINE>           ::= <CONST>

<FIELDS>         ::= <FIELD><FIELDS>!<NULL>
<FIELD>          ::= <FIELDTYPE>[/<NAME>,!
                     ,!<NULL>]<NAMELIST><NL>
<FIELDTYPE>      ::= $VS ! $VD ! $VQ !
                     $VB[<[><CONST><]>!<NULL>]
                     ! $VEC[<[>$VB<]>!<NULL>]


<EXTENDEDTABLE>  ::= $EOR <NL><FIELDS><NL>
                     [$END!<EXTENDEDTABLE>]
```

A table declaration creates a data structure corresponding to the fields declared within it. The data structure may be mapped as an array of records or as a set of vectors. This mapping is dependent upon the target machine - large paged machines, viz. MU5 - 2900, have records - small unpaged machines, viz PDP11, membrane, have sets of vectors.

The address part of the table header specifies where the table is to be allocated. If absent the vector segment is used. The split form of the address allows the user runtime allocation of the segment.

Field Declarations

```
Fieldtypes :          $VB     -        a byte
        $VB[<CONST>]  - <CONST>  no of bytes
        $VS , $VD, $VQ  single, double, quadruple
                        length word size
        $VEC            descriptor size
        $VEC [$VB]      a variable no of bytes
```

Not all field types are implemented on some machines.

MU5/2900 has all types.

Membrane $VS, $VD, $VB, $VB[2], $VB[8], $VEC, $VEC[$VB]

Only one field of type $VEC [$VB] is allowed per record, it must also be the last field of that record. Two names must be given when declaring this field type - the first is the name of the field - the second is a base name for the field. This latter name is used in accessing the field, see below, and in advancing over records, see 3.4.1.

Field names may be equivalenced if the field types are the same. Care should be taken not to overflow the size of the record when equivalencing.

Extended Records

A record may notionally be extended by use of the end of record marker, $EOR. Further declarations may then be equivalenced onto the previous names. The new fields have the effect of being in the next record. Note: Extended records and variable length records are not allowed to be mixed.

Accessing a Field Name

A fixed length field of a record is accessed in the same way as a vector element. - See array accessing 4.1.1.

e.g. FIELD A [POSITION OF RECORD 1]

The subscript is the position of the record being accessed.

For variable length fields - the basename should be mentioned as follows:

VARFIELD [VARBASE : POSITION OF RECORD 1]

The above will access the zeroth element of the varfield of record
See 4.1.3 for syntax description.

Examples of Table Declarations

```
$TABLE  NAMELIST [NOOFNAMES $AT PW1/0]
$VB[2]  NEXTNAME, NAMEPROPS
$VB     NAMECOUNT
$VEC[$VB]  NAMECHRS, NAMEBASE
$END

$TABLE  PROPLIST [NO OF PROPS]
$VB[2]  NEXTPROP, IDENLINK, CBLINK
$VB     PBLOCKNO, PTEXTLEV, PTYPE, PBASESIZE
$VB[2]  PVALUE16, PVALUE162
$EOR
$VB[2]  / NEXTPROP, PVALUE164, PVALUE168
$END
```

3.4.1 The Advance Statement

```
<ADVANCESTAT>   ::=  $ADVANCE <POINTER>*<CONST>,<TABLENAME>
                     [,<VARLENGTH>,<VARBASE>!<NULL>]

<POINTER>       ::=  <NAME>
<VARLENGTH>     ::=  <NAME>!<CONST>
<VARBASE>       ::=  <NAME>
```

This statement must be used to advance the pointer to a record - (notionally allocate position for next record). The <const> specifies no of records to be advanced over.

If a variable length field exists in the record then <varlength> is the no of elements of the field and <varbase> is the base name of the field.

Examples.

```
$ADVANCE NEWPROP*1, PROPLIST
$ADVANCE NEWNAME*1, NAMELIST, NOOFCHARS, NAME.BASE
```

3.5 Literal Declarations

```
<LIT.DEC>    ::=   $L <LIT.LIST>
<LIT.LIST>   ::=   <LIT.ITEM>[,<LIT.LIST>!<NIL>]
<LIT.ITEM>   ::=   <NAME>:<LITERAL>
```

This creates a named literal, which may be used in any context where a <LITERAL> is required. The value of the literal is given by the ':<LITERAL>' part of the <LIT.ITEM>.

Examples of Literal Declarations:

```
$L MASK      :     %7FF, SHIFT:7
$L CR.SEG.L  :     $F CREATESEGMENT
```

3.6 Datavec Declaration

```
<DATAVEC>    ::=   $DATAVEC<BASE><NAME.DISP>
                   (<ELSIZE>)<NL><DV.ELEMS>
                   $END
<ELSIZE>     ::=   see 3.3 above
<DV.ELEMS>   ::=   <DV.LINE>[<[><LITERAL><]>!<NIL>]<NL>
                   [<DV.ELEMS>!<NIL>]
<DV.LINE>    ::=   <LITERAL>[,<DVLINE>!<NIL>]
```

A datavec declaration creates a named descriptor to an initialised vector. If a base is specified the descriptor will be assigned to a variable relative to that base for which a displacement may optionally be given. If no base is specified the implementation may either use a literal descriptor or create a variable as above. If the vector contents are to be altered in execution, it should be declared in a segment other than the code segment (using *SEGMENT - Chapter 1.). Otherwise the contents are normally planted in the code at the point of declaration.

The initial contents of the vector are specified as a sequence of elements of the required size, separated by commas or newlines. '[<LITERAL>]' at the end of a line indicates that the elements on that line are to be repeated and occur in total <LITERAL> times.

Examples of Datavec Declarations:

```
$DATAVEC  DIGIT  ($V1)        :-   1 - bit elements
   0[48]; 1[10]; 0[70]        :-   00...011...100...0
$END
$DATAVEC  SVEC  ($VS)
0                             :-   0
1, 2,3 [10]                   :-   1,2,3,1,2,3,....1,2,3
0, 0                          :-   0, 0
$END
```

3.7 Datastring Declarations

```
<DATASTR>   ::=  $DATASTR <NAME>   %"<CHARS>" <NL>
<CHARS>     ::=  see chapter 1
```

A datastring declaration creates a named literal descriptor to an initialised character string.

Examples of Datastring Declarations:

```
$DATASTR START.CAP    %"COMPILATION.STARTED.AT."
$DATASTR HEX.CAP      "!22!LPT!22 !"
```

3.8 Switch Declarations

```
<SWITCH>   ::=    $SWITCH <NAME> <[> <LITERAL> <]>
```

The switch declaration declares a switch name. The switch has <LITERAL> elements 0 to <LITERAL> -1. Each switch label is declared using the form S [N]: where S is the switch name and N the literal element number. (Switches are available in high-level form only).

Example of Switch Declaration:

```
$SWITCH CHAR[128]
```

3.9 Label Declarations

```
<LABEL>    ::=    <NAME> : !
                  <NAME> <[> <LITERAL> <]> :
```

The first form declares an ordinary label; the second, a switch label. In the case of a switch label, the switch must already be declared at the current level.

Chapter 4

High Level Statements.

```
<HIGH.LEVEL.STATEMENT>  ::=  <COMP.SEQ>  !
                             <DESC.SEQ>  !
                             <CONTROL.ST>
```

The high level statements are those provided in machine-independent form on all machines. They fall roughly into three categories.

Computational statements
Descriptor manipulation statements
Control statements.

4.1 Computational Statements.

```
<COMP.SEQ>      ::=  <TYPE> <SEQUENCE>
<TYPE>          ::=  $IS!$ID!$IQ!    :- INTEGER
                     $RS!$RD!$RQ!    :- REAL
                     $LS!$LD!$LQ!    :- LOGICAL
                     $DS!$DD!$DQ!    :- DECIMAL
                     <NIL>           :- DEFAULT,$IS
<SEQUENCE>      ::=  <OPERAND><TYPE>[<HL.OP><SEQUENCE>!<NIL>]
<PLAIN.SEQ>     ::=  <OPERAND>[<HL.OP><PLAIN.SEQ>!<NIL>]
<OPERAND>       ::=  <VARIABLE>!<LITERAL>!
                     <ARRAY>!<PROC.CALL>!<VAR.FIELD.ACCESS>
                     (<PLAIN.SEQ>)
<HL.OP>         ::=  +!-!-:!*!/ !/:!
                     <-!<=&!!!-/=!
                     =>!+>!->!-:>!*>!/>!/:>!&>!-/=>!!>
```

A computational sequence specifies a sequence of operations to be performed on a sequence of operands 'load' is implied as the first operator; thereafter, operations are performed from left to right. Thus for example:

```
A + B => C
```

is equivalent to the ALGOL

```
C := A + B
```

The mode of computation may be specified as integer, real, logical (unsigned integer) and decimal, single, double or quadruple length. Not all types are implemented on all machines, but IS, LS and LD is regarded as the minimal subset. If no mode is specified, IS will be assumed.

Type and/or size conversion is achieved by inserting the required <TYPE> at the desired place in the sequence. The type and size remains the same throughout a sequence unless such a <TYPE> is inserted. Note that such conversions are only allowed at the outermost level in a computational sequence.

e.g.

```
CHAR - ''0''  $LD + NAME => NAME
```

The operator meanings are

```
+ add, - subtract, * multiply, / divide
-: reverse subtract, /: reverse divide
<- shift, <= rotate (positive operand - left,
negative operand - right)
& and, ! or, -/= exclusive or
=> store (assignment)
```

Any operator followed by > means operate on the following operand and then assign to that operand. In present implementations the accumulator will contains the assigned value after the operations.

e.g. `1 +> A` is equivalent to `1 + A => A`.

(<PLAIN.SEQ>) invokes stacking of the partial result so far, evaluation of <PLAIN.SEQ>, and a reverse operation on the the stacked operand. Note that this may be expensive on some machines.

4.1.1. Array Accessing.

<ARRAY> :: [<NAME>!<LITERAL>]<[><PLAINSEQ><]>

The subscripted name or literal must yield a descriptor. The subscript expression may not contain any type conversions. The effects of accessing outside the array bound is undefined.

[Note: care should be taken in using arrays to ensure that they are used in a context in which the element size is apparent. For example, when a descriptor is passed as a paremeter to a procedure, or computed as the result of a descriptor manipulation sequence (see 4.2 below) the type of the objects it describes is not known at compile time. In the absense of further information, (for example size information made available at run-time on some machines), the compiler will assume that the size of object referenced by such a descriptor is compatible with the <TYPE> of computation in which it is used].

4.1.2. Procedure Calls.

```
<PROC.CALL>   ::    <NAME>([<PAR.LIST>!<NIL>])
<PAR.LIST>    ::    <PARAM>[,<PAR.LIST>!<NIL>]
<PARAM>       ::    <COMP.SEQ>!<DESC.SEQ>
```

The default mode of evaluation of parameters will be that given in the procedure heading or specification. Otherwise the normal rules for computational/descriptor sequences apply, If the mode at the end of the sequence differs from the required mode, an appropriate conversion is performed if possible.

A functional procedure call involves stacking the partial result, calling the procedure, and performing a reverse operation on the stacked operand.

Library procedures may be called without prior declaration.

4.1.3. Variable length field Accessing

```
<VARFIELDACCESS>   ::=  <FIELDNAME><[><BASENAME>:<RECORDPOS>
                        [<HL.OP><PLAINSEQ>!<NULL>]<]>

     <FIELDNAME>   ::=  <NAME>
     <BASENAME>    ::=  <NAME>
     <RECORDPOS>   ::=  <NAME>!<CONST>
```

This form should be used for accessing elements of variable length fields of records. The subscript expression may not contain any type conversions. Accessing beyond the field and table bounds is undefined

[Note: See note in 4.1.1 - Array Accessing]

4.2 Descriptor Sequence

```
<DESC.SEQ>     ::=  $S[[$DESC!$VEC]<OPERAND>=><OPERAND>
                    ! <OPERAND><DSEQTAIL>]
<DSEQTAIL>     ::=  [[[$MOD!$SMOD!$BOUND!$ORIG!$TOP][<NAME>!
                    <CONST>]! => <OPERAND>]<DSEQTAIL>!<NIL>]
```

$DESC <OPERAND> generates a descriptor of the operand specified as parameter. If the operand is a name XNB or NB are used as in operand accessing.

$VEC <OPERAND> where the operand is a vector element generates a descriptor vector formed by the remainder of the of the vector starting at the specified element.

e.g. $VEC A[I]

$MOD : modifies the contents of the register by the single length operand in the same manner as
when accessing the store, and subtracts the
operand from the bound field of the register.

$SMOD : similar to $MOD except that neither boundcheck,
scaling nor altering the bound take place.

$BOUND : the bound of the register is replaced by the single length operand.

$ORIG : the origin of the register is replaced by the single length operand.

$TOP : the top half (type, size 8 - bits) of the register
is replaced by the single length operand. Note that
interpretation of type information on some machines makes type and size changes machine dependent.

NOTE: Use of subscripted operands in descriptor sequences will destroy the contents of the Accumulator. No operands in descriptor sequences may be procedure calls.

4.3 Control Statements

```
<CONTROL.ST>::=-><OPERAND>[<[><SEQUENCE><]>!<NIL>]!
              $JUMP <OPERAND>!
              <IF.CLAUSE>!
              $ELSE ! $F1 !
              $CYCLE<CYCLE.CLAUSE>!
              $REPEAT<REPEAT.CLAUSE>!
              $RETURN
```

4.3.1 Unconditional branches

-> <OPERAND> an unconditional branch, in which <OPERAND> is is the label name of a local label. (Also appears in low level forms).

-> <OPERAND>[<SEQUENCE>] is a switch. <OPERAND> is a local switch name and <SEQUENCE> a subscript expression.

$JUMP <OPERAND> is a non-local jump to the outermost level of the program. The stack is reset but no display manipulation takes place.

Examples

```
-> 1
-> s[i]
```

4.3.2 Conditional branches

```
<IF CLAUSE>    ::= <LOGICAL.SEQ>[,-><VARIABLE>!$THEN]
<LOGICAL SEQ>  ::= <LOGICALEXPR>[[$ANDF!$ORF]<LOGICALSEQ>
                        <NIL>]
<LOGICAL EXPR> ::= <COMPSEQ><RELOP><OPERAND>
<RELOP>        ::= <!=<!>!>=!=!/=
<VARIABLE>  is a label name.
```

Examples

```
$IF a < b , -> FRED
$IF a+b < c $ANDF d=c , ->FRED
$IF x = < "9" $ORF x >= "2" , -> FRED2
$IF a < b $THEN
$ELSE
$FI
```

The variable after -> is a label name. Logical sequences are evaluated left to right. The operators $ANDF and $ORF cause the following sequence to be conditionally evaluated. $FI must follow $THEN but the ELSE clause is optional.

4.3.3 Loops

```
<CYCLE CLAUSE>   ::=  $WHILE <LOGICALSEQ> ! <NIL>
<REPEAT CLAUSE>  ::=  $UNTIL <LOGICAL SEQ> ! <NIL>
```

Examples

(1)
```
$CYCLE
$REPEAT
```

(2)
```
$CYCLE $WHILE a/= b
$REPEAT
```

(3)
```
$CYCLE
$REPEAT $UNTIL I + 1 => I = 100
```

(4)
```
$CYCLE $WHILE  I + 1 => I < 100

$REPEAT $UNTIL  ERROR < EPS
```

The $WHILE clause causes a test to be planted at the start of the loop, in the absence of this clause the loop is obeyed at least once. The $UNTIL clause causes a test at the end of the loop, when absent an unconditional loop back occurs. Therefor example (1) loops infinitely, (2) tests at start of loop, (3) at end of loop and (4) both at start and end. Any cycle control variable, such as I, must be initialised outside the loop.

4.3.4 Procedure Return

$RETURN causes a return from the current PROC,PROCEDURE or SUBPROC.

Chapter 5.

MU5 Instructions.

To be added later.

Chapter 6

2900 Instructions

There are five types of indtructions:

1)	Computational	<REG><OPR><OPND>
2)	Organisational	<ORG REG><ORG OPR><OPND>
3)	Store to store	$<STS.NAME><OPND.2>
4)	Branch	$IF<COND>, -> <OPND.3>! [-!<NIL>] -> <OPND>
5)	Special functions	$<FN NAME>[<OPND>!<NIL>]

The most common form of operand <OPND>, is described first.

6.1 Operands

The syntax for <OPND> is

```
<OPND>      ::=   BM ! TOS !
                  [DR!<NAME>!<LITERAL>!<NIL>]
                  [ <[> [BM!TOS!<NAME>!<LITERAL>]<]>!<NIL>]
```

<NAME> As defined in chapter 1. The names defined as <REG> or <ORG REG>, are predeclared operands and may not be used for other purposes.

```
<LITERAL>   ::=   <DECIMAL NUMBER>!
                  <HEX NUMBER>!
                  "<CHARACTER STRING>"!
                  !<DESCRIPTOR LITERAL>!
                    <NAMED LITERAL>
```

The operand forms <NIL>[<NAME>] etc is the means by which access to image store is achieved. Note, it does not access main store.

Examples of operands

```
FRED [5]
JOE  [BM]
JACK [JILL]
DR   [1]
DR   [BM]
TOM
01CAB
BM
[3]   Image store accesses
[BM]
```

Not all of the operand forms exist in the hardware, hence some will result in extra instructions as follows:

```
<NAME1>[<NAME2>]     => DR  =   <NAME1>
                                DR[<NAME2>]
<NAME1>[<LITERAL>]   => DR  =   <NAME1>
                                DR[<LITERAL>]
                        (When the literal is non zero)
```

A similar form exists when <NAME1> is replaced by a <LITERAL>

```
[<NAME>]   =>   BM   =   <NAME>[BM]
```

DR itself cannot be accessed directly as an operand, and so this operand form is faulted.

6.2 Computational instructions

```
        <REG><OPR><OPND> [<REST>!<NIL>]
Where   <REG>       ::=   AL!AD!AI!AR!BM
        <OPR>       ::=  <LOAD OPR>!
                          +!-!-:!*!**!/!//!/:!
                          <>!<-!<=!&!'.!-/=

        <LOAD OPR   ::=  [*!<NIL>][=!:=!::=!:::=]

        <REST>      ::=  <OPR><OPND> [<REST>!<NIL>]
```

The only load operators allowed for BM are = and *=. The load operation = does not involve a change in accumulator size, whereas :=, ::= and :::= set the accumulator size to single, double or quadruple length respectively Not all of the operators are available in the hardware for all types of accumulator. The following table summaries those available.

	AL	AD	AI	AR	BM	+
	-					
	-:					x
	*	x				
	**	x				
	/	x				x
	//	x				x
	/:	x				x
	<>					
Identical	<-					
operation	<=					x
for all	&					x
types of	!					x
accumulator	-/=					x
						x

Operators available for computational orders

Examples of computational orders:-

```
BM = COUNT
AI := TAB[BM]
AI + 1
AI => DR[BM]
BM + 1

or  AI := TAB[BM]+1 =>DR[BM]
BM <> 255
or  BM + 1 <> 255
```

6.3 Organisational instructions

```
        <ORG REG><ORG OPR><OPND>

Where   <ORG REG>  ::=  NB$F!XNB!DR!DT!DB!DO!PS

        <ORG OPR>  ::=  *=!=!=>!+!-
```

The following table indicates the vadility of the <ORGREG> / <ORG OPR> combinations.

	NB	SF	XNB	DR	DT	DB	DO	PS
*=	x	x	x		x	x	x	x
=								
=>			x		x	x	x	
+			x		x	x		x
-			x	x	x	x	x	x

Note that NB- actually sets NB = SF -

Examples OF ORGANISATIONAL ORDERS:-

```
NB => TOPSTACK
SF + 4
NB - 5
DR = TABLE
DO + 1
DR => TABLE
```

6.4 Store to store instructions

```
        $<STS NAME><OPND.2>
Where   <STS NAME>          ::=   TCH!AND$ORS !NEQS!
                                   FK !INS !SUPK!EXP !
                                  SWEQ !SWNE!CPS !TTR !
                                  MVL  !MV  !CHOV!COM !

        <OPND.2>            ::=   [DR,!<NIL>]<LITERAL>
                                  [,<LITERAL>,<LITERAL>!<NIL>]
```

If DR is specified in <OPND2>, then the length of the operation is taken from the descriptor in DR, otherwise the first literal specified provides the length.

The two optional literals give the mask and filler for the operation. On some orders, if these are not specified, the value in B is used as a mask and filler.

Examples of store to store orders:-

```
MVL DR, 0
TTR 255
SWNE DR, "*",  FF,0
```

6.5 Branch instructions

```
$IF <COND>,-> <OPND3>! [-!<NIL>]-> <OPND>
```

Where <COND> ::= <COND REG><COMP> 0!
<COMP>!
<LITERAL>

<COND REG>::=<REG>!DB!OV
<COMP>::==!/=!>!>=!<!=<
<OPND3>::=[DR!<NAME>!<LITERAL>]
[<[>[BM!<LITERAL>]<]>!<NIL>]

The restrictions applying to DR and <NAME>[ ] operands are described in 6.1.

<COMP> provides the more usual forms of relational operators. However, this does not cover all the available hardware conditions. These are made available by the <LITERAL> type of condition.

The hardware does not cater for all possible <COND REG><COMP> combinations. The restrictions are indicated in the following table.

	AL	AD	AI	AR	BMDBOV
=					
/=					
>	x				xx
>=	x				xx
<	x				xx
=<	x				xx

Examples of conditional instructions:

```
$IF  AI < 0 ,  -> LB1901
$IF    >=   ,  ->  LAB2
$IF     3   ,  ->   RM
```

6.6 Special functions

```
$<FN NAME>[<OPND>!<NIL>]
```

```
Where  <FN NAME    ::=   ACT!CALL!CBIN!CDEC!COMA!
                         CPIB!CYD !DEBJ!DIAG!ESEX!
                         EXIT!EXPA!FIX !FLT !IDLE!
                         INCT!JLK !LDRL!LUH !OUT !
                         RRTC!SIG !SHS !SHZ !STUH!
                         TDEC!VAL
```

Chapter 7.

PDP11 Instructions.

To be added later.

Chapter 8.

MUPL ON MB7700.

A subset of MUPL is used for system programming on the MB7700. The restrictions are outlined below, related to the descriptions in the main part of the manual.

Chapter 1 :		No floating constants *LIB, *PROG, *PNL, *CONSTANTS not implemented
Chapter 2 :	2.3.3	$PROCEDURE not implemented
Chapter 3 :	3.2	$VS, $VD, only. At most 256 variables or globals. No xnb names.
	3.3	$VB, $VD, $VEC sizes only.
	3.4	No table declarations
Chapter 4 :	4.1	IS, LS, LD. Only = , => for LD modes
	4.2	Only = , => implemented

Further notes on MB7700 MUPL.

(1) Multiple-length working

The MB7700 has only single-length (16-bit) operations defined in its basic order code, and it is felt to be undesirable to implement multiple length operations in software. However $LD is implemented for load and store operations, and is represented by 4 single length values.

Descriptors in the MB7700 consist of three 16-bit words. Because of the difficulty of manipulating these objects, no explicit descriptor literal facility is provided. Similarly the $AT facility for vectors is not implemented. Library procedures which explicitly use addresses (e.g. SET.TRAP, OUT.STACK) will use descriptors.

(2) Operators
The operators

```
+ , -, *, /
<-, <=, &, '., -/=
=>
```

are implemented efficiently (1 order) on the MB7700. The reverse operators,

```
-:, /:
```

are implemented by stacking the accumulator and loading the second operand (1 order), then operating on the top of stack (1 order). The 'assignment operator'

```
+>,->,*>,/>, etc...
```

Are implemented as two independent operations, 'operate' and 'assign'.

Appendix 1.

Example Program Run On MU5.

```
***A JOB MUCS SCUM MUPLEX
MUPL

:- THIS PROGRAM SEARCHES FOR A LINE STARTING WITH
:- *COMPILE . IT OUTPUTS THE LINE, WITH SPACES
:- REMOVED, IF SUCH A LINE IS FOUND.

$BEGIN
  $VS CHR,T
  $VEC BUFFER [$VS 160]

*COMPILE IF 2 :- MU5  - ISO CODES
  $L NP : 12, NL : 10, EOF : 4
*COMPILE IF 1 :- 2900 - EBCDIC CODES
  $L NP : %C, NL : %25, EOF : 3
*COMPILE IF 0

  $PROC FIND.ENTRY $IS
    $DATAVEC HEADING ($V8)

      "*","C","O","M","P","I","L","E"
    $END
    $VS CHR.T

  LOOP:
    $CYCLE $WHILE INSYM() => CHR /= NL $ANDF CHR /= NP
      $IF CHR = EOF, -> EXIT
    $REPEAT
    $IF INSYM() => CHR /= "*" ,-> LOOP
    CHR => BUFFER[]; 0 => T
    $CYCLE $WHILE INSYM() => BUFFER[ 1+> T] => CHR /= NL #
                             $ANDF CHR /= NP
      $IF CHR = " " $THEN
        1 -;> T
      $FI
    $REPEAT

    :- TEST FOR *COMPILE

    0 => T
    $CYCLE $WHILE 1+> T <8
      $IF BUFFER[T] /= HEADING[T], -> LOOP
    $REPEAT
    1 => FINDENTRY
    $RETURN
  EXIT:
    -1 => FINDENTRY
  $END
  SELECTINPUT(1)
  $IF FIND.ENTRY() > 0 $THEN
    -1 => T
    $CYCLE $WHILE BUFFER[1+> T] => CHR /= NL $ANDF CHR /= NP
      OUTSYM(CHR)
    $REPEAT
    NEWLINES(1)
  $ELSE
    CAPTION(%"****NO SUCH LINE"); NEWLINES(1)
  $FI
  SELECTINPUT(0)
$END
*END
CO MUCS SCUM
DI 1 MPL103
RUN
STOP
***Z
```

```
    MICS              MUPLEX        STREAM 0 SECTION 0
STREAM  0          SECTION  0                TIME 14.31.12              DATE 27.09
START TIME  14.31.06.  ON  27.09.79.
MUPL
14.31.06. "MUPL" COMPILER, ON 27.09.79
        1.  10    BEGIN BLOCK 1 AT 5:00000007
        1.  21      PROC FINDENTRY AT 5:00000034
        1.  22        DATAVEC HEADING AT 5:0000003A
        1.  24        END DATAVEC AT 5:0000003E
        1.  49      END FINDENTRY AT 5:00000097
        1.  62    END BLOCK 1 AT 5:000000E7

14.31.07. COMPILATION SUCCESSFUL
OD MUCS SCUM
DT 1 MPL103
RUN
*COMP.LEI:NOT64
STOP
STOP REASON      0    COST      3   TIME  14.31.07.  27.09.79
```