```
.type sloe.lst
01 0001
01 0002          $
01 0003  0       A = 0
01 0004  1       B = 1
01 0005          LIST-OPERATORS

                 OPERATORS

name   prec  unary?  binary?            function

!        20     no      yes     boolean inclusive or
#        20     no      yes     boolean exclusive or
&        20     no      yes     boolean and
(         1    yes       no     left parenthesis
)         1     no      yes     right parenthesis
*        10     no      yes     integer multiplication
+         5    yes      yes     integer addition
,,       30     no      yes     18 bit halfword
-         5    yes      yes     integer subtraction
.AND.     2     no      yes     logical A and B
.OR.      2     no      yes     logical a or b
/        10     no      yes     integer division
/=        2     no      yes     logical not a=b
<         2     no      yes     logical a<b
<=        2     no      yes     logical a<=b
==        2     no      yes     logical a=b
>         2     no      yes     logical a>b
>=        2     no      yes     logical a>=b
DEF    1000    yes       no     boolean, true is symbol known
IDEN   1000    yes       no     boolen, true if next two tokens identical
\        10     no      yes     integer remainder
\\       30    yes       no     boolean not
_        30     no      yes     arithetic shift
__       30     no      yes     logical shift

01 0006
01 0007          list-nextpc
next pc routines
16-WAY           set pc to first of a w word block
4-WAY            set pc to first of a w word block
8-WAY            set pc to first of a w word block
PC-ANY           pc+1 preferred, but anywhere is ok
PC-EVEN          set pc to an even pair
PC-PLUS1         unconditional +1
PC-PLUS20        unconditional pc+20


.dayt
11-FEB-1982 1:46:15 PST

.

.
.
.
.sysn
TYMSHARE C930-P034/K04 2-9-82
.
```

1.0 CONCEPTS

Sloe is designec to be a general purpose assembler for microcode. As such, there are certain underlying assumptions about the architecture of the machines, and the desirable form of the microcode to drive the machines. These assumptions will be stated shortly.

Sloe is syntacticly direct. The only distinctions among characters are between 'separatcr' characters and all others. The only characters with any implied meaning are [ ] ( ) $: ; and ' . This structure allows the widest possible latitude in the definition of symbols.

Sloe is transformed from its initial state into an assembler for some particular machine by making an appropriate set of declarations, corresponding to the architecture of the target machine. There is no separate procedure for declarations. As a design feature, all the capabilities for defining and extending the microcode are always available. In practice, there will be a parameter file for each target machine sloe assembles for, but is always possible to extend the basic parameter in any compatible direction.

Error checking is an intrinsic part of any assembler. Sloe diligently checks for many kinds of errors that are implied by the declarations that have been made. Among these are multiple definitions, missing definitions, syntax errors, missing operands, unbalanced parentheses, and so on. Sloe also checks for errors that are implied by the structure of the declarations that have been made. These include all errors involving attempting to encode impossible or illegal operations. The effectiveness of sloe's strategy for detecting this kind of error is directly related to the precision with which the declarations were made. Loosely structured declarations will result in undetected errors. There is also provision for detecting ad hoc errors, those which are not deduced from the declared structure, but which nonetheless are impossible or undesirable in the actual hardware.

2.0 THE MACHINE MODEL

Here, we refer to the presumed propertied of the target machine which sloe is to assemble for.
|L

2.1 Memories

The target machine has one or more memories. each memory has its own size and its own word format. Memory words may be any width. The total number of bits in all memories is small; Less than a million. There is provision for automatically generating parity bits.

2.2 Parallelism

Micro machines are likely to have a large number of functions, many of which can operate in parallel. This gives the microcode a "do this and do that and do something else" flavor.

2.3 Incompleteness

Not all operations that can be encoded in an instruction can actually be executed by the hardware. For instance, a multiplier and a shifter might both use an internal bus, so both can't be active simultaneously, even if the microcode could code for such an operation.

2.4 Microinstructions

Each microinstruction occupies exactly one word in one of the memories. Microinstruction format is hamming-decodeable, though not necessarily instantaneously. The implied 'next instruction' may be other than .+1, or may not even ~~exits~~ exist. The encoding of the microcode word can be expressed in terms like "if field A has value X and Field B has value Y, then field C can have values C(1) C(2) or C(3)."

3.0 BASIC ASSEMBLER SYNTAX

SPACES are the characters <space> <tab> <cr> <lf>
SEPARATORS are the characters [ ] ( ) ' $
NUMBERS are 0 1 2 3 4 5 6 7 8 9
LETTERS are A-Z a-z
Lower case is always equivalent to upper case
SPECIALS are all other printing characters
SYLLABLE ANY sequence of LETTERS NUMBERS and SPECIALS
|L

or
one SEPARATOR

Notice that the group called SPECIALS is not really special, only an amalgamation of all the other characters. Syntactically, they are never treated differently than letters or numbers. This is likely to cause some consternation in expression evaluation, where the familiar syntax: A=B*C must be written as A = B * C However, This is a small encumberace in view of the power to define symbols like A>=B without confounding a well meaning syntax analyzer.

The SEPARATORS group are the only characters that get special attention. They constitute a concession to convenience, with the cost that they cannot be incorporated into symbols. Each of the separators has a specific function associated with it

$ terminates multi part declarations and microcode words
( ) group expressions in the usual way
[ ] group operands, values for microcode fields.
' is the concatination character for the macro processor
: defines a label
; begins a comment to the end of line
NOTICE that this is the only place where end of line is not synonymous with a space

4.0 SYMBOLS

Any syllable which is not a separatcr can be defined as a symbol. Except for one case, nothing is implied by the characters in a symbol, that is "A=B" "FOO" "!!" "A-B" are all perfectly valid symbol names, and no arithmetic or logical operations are implied by any of them. The one exception is the single quote character "'", which is given special meaning in some restricted contexts. (see: FIELD VALUES) There are several subclasses of symbols, which are determined by the type of declaration, and which have different useful attributes affecting the assembly proces. Symbols of the same name with different attributes are sometimes permitted, but not encouraged.
|L

SYMBOLS

4.1 PSEUDO OPERATORS

Pseudo ops are the predefined elements of the language. Note that ALL of sloe's predefined language is implemented as pseudo ops, even those like assignment (=) and label declaration (:) which are not normally thought of as such. The pseudo ops will be described in detail later.

```
EXAMPLE:
  DECLARE-MEMORY MAIN
     10 1000
```

"Declare-memory" is a pseudo op, in this case the one that sets the basic memory parameters; an 8-bit by 512-word memory. The rest of the text is interpreted by the particular pseudo op processing routine. Note that end of line does not end the argument list. This is generally true. End of line is almost always equivalent to a space.

4.2 MACROS

Macros provide capability to extend the language processed. The macro processor is rudimentary but functional.

```
EXAMPLE:
.DEFINE FOO [ arg1 arg2 ]
[ arg1 = arg2 ]
```

Note that ARG1 and ARG2 are arbitrary syllables, and that there are no separators between them. The pseudo ops for defining macros and related construtions will be described in detail later.

4.3 FIELDS

A field is a contiguous set of bits within a microcode word that requires a value be specified. Declaration of fields within the microcode of each memory in the target machine is the basic operation that makes sloe an assembler for a particular machine. The stringency of the field declarations you make determines sloe's ability to detect errors in the microcode it assembles.

EXAMPLE: (with the memory definition in the previous example)

```
|L
```

```
OPCODE = FIELD 3 4 $
```

Declares OPCODE as a 4 bit field, whose low order bit is bit 3, counting bit 0=leftmost bit. Internally, this generates a mask of the significant bits in the field within themicrocode word:

```
11110000 (binary)
```

4.4 WORDS

A field, with the required value specified, becomes a word. Words correspond directly to microcode words. The basic assembly operation consists of IOR ing together several word symbols, and Checking that IOR ing the symbols together doesn't assign one part of the microcode word two different values.

EXAMPLE: (extending the previous example)

```
ADD = OPCODE[5] $
SUB = OPCODE[6] $
```

Defines a ADD as a word that has the following mask and value

```
11110000 OPCODE field (binary mask)
01010000 ADD (value bits)
01100000 SUB (value bits)
```

Any attempt to change the value of a bit which has already been specified is an error. For instance:

```
ADD SUB $
```

Generates an error message in the above example, because some bits of the OPCODE field are required to have two different values. This is the basis for most of SLOE's error checking.

There are several variations on the basic FIELD and WORD types, which are for convenience and syntactic clarity, but are not essential to the concept. These embellishments will be described in detail later.
|L

4.5 VALUES

Value symbols associate a name with an integer. The syntax to declare a value is

valuename = expression

Note that there must be spaces separating the = from the name and expression. The limits on expressions will be discussed in detail later, but briefly; an expression involves only integer numbers, numeric pseudo operators, labels, and previously defined values.

Values, unlike fields words and labels, can be redefined.

4.6 LABELS

Labels are values with additional attribute 'label' attached. Labels are declared by the ":" pseudo op.

for EXAMPLE:

FOO:

Declares the label "FOO" at the current memory location. Unlike values, labels cannot be redefined, but unlike everything else, a label can be forward referenced. Fields can have the attribute of requiring a label to fill their vacancy. Any undefined symbols encountered in expressions are assumed to be forward references to labels. When the forward references are eventually resolved, the value is ADDED to the field. This means that expressions like:

```
FOO + N ;Where FOO is a forward reference
```

Will work, but expressions like:

```
FOO ! N ;Where FOO is a forward reference
```

Will not work.

? CHECKED ?

4.7 NUMBERS

A syllable which consists of only digits, possibly termininated by a decimal point, can be interpreted as a number; if all else fails. You CAN define such syllables as symbols, so caution is advised. Even numbers are not necessarily what they seem!

|L

4.8 PCROUTS

PCROUTS are a special class of pseudo operator, which specify an algoritm for selecting the location for the next microinstruction. For all syntactic purposes, PCROUTS are identical to PSEUDOs.
|L

5.0 BLOCKS

A BLOCK is a group of syllables, delimited by correctly nested left and right markers. A left marker is either a "[" or "BEGIN XXX", where XXX is the name of the block. A right marker is either "]" or "END XXX", where XXX is the name of the block. The name of the block serves to enforce the matched closure of blocks.

EXAMPLE:

```
[ ;beginning of ablock
 A = 1
BEGIN FOO ;begin a names block
 B = 1
END FOO  ;end of the named block
]  ;end of the unnamed block
```

All of the constructs that expect BLOCKs as arguments allow blocks to be nested, and insist that named blocks be matched.

Within this document, blocks will be referred to as XXX-BLOCK, where XXX is somewhat descriptive of the function of the block. Two caveats are in order: First, those descriptions that explicity use [and] mean it. BEGIN FOO ... END FOO is not acceptable (for instance) in supplying field values. Second, do not confuse the use of the word BLOCK with the .BLOCK pseudo op, which is totally unrelated.
|L

6.0 ASSEMBLER DIRECTIVES

The following section describes each pseudo op in the current release of SLOE, in alphabetical order. Most pseudo ops are recognised in any context, but a few are recognised only in restricted contexts. These exceptions to the rule will be noted.

SLOE pseudo ops are implemented by ad-hoc subroutines. Those which require arguments usually use the standard parser, so their input will look much like other SLOE constructions. Those which take numeric arguments are not restricted to simple numbers, but will accept an arbitrary expresion.

6.1 $

The "$" pseudo op is used as a terminator, which returns the assembler to it's top level state from whatever sub context you are in. The two most important functions are:

$ terminates the scope of a FIELD pseudo op.

$ terminates assembly of a microcode word, and initiates the process of supplying default values for unspecified fields and for selecting the next microcode PC.

$'s encountered when neither a FIELD declaration is in progress or a microcode word has been built will not cause any microcode to be generated or any error messages to be generated. So you can sprinkle $'s around to suit your taste, without fear of trashing anything.

6.2 .BLOCK [lb Ub] <0X1> #,#,#:# ...

The .BLOCK pseudo op finds a block of memory meeting constraints specified by the arguments. The next microinstruction will be assembled at location 0 of the block. The .BLOCK pseudo op does not actually allocate the memory locations, but only assures that they are available. Use the ":" pseudo op to determine which location within an allocated block to use,, or rely on the default sequencing, if that does the right thing.
|L

You can use .MARKOUT in conjunction with .BLOCK, to mark memory locations used.

For convenience and appearance' sake, .BLOCK uses a slightly nonstandard syntax in its scan. A .BLOCK pseudo op is terminated by end of line, unless the last character on the line is a comma or a colon; and within the "#,#:#" part of the argument list, COMMA is considered to be a separator. This will cause confusion if you try to use expressions involving symbols with commas in their names.

It is also Ill advised to use the .BLOCK psuedo op in the middle of assembling a micro instrucion; a warning message is generated if you do.

[lb ub]
: specifies the bounds within memory what will be searched for a location that meets the other constraints imposed. LB and UB default to 0 and max-value respectively. LB and UB can be an expression rather than a constant. The whole of [lb ub] is optional

<0x1>
: is entirely optional. if specified, it is a picture of the low order adress bits of the block to be found, x specifys a "don't care" bit. If omitted, <xxxxx> is assumed (with enough x's so any address will match).

#
: is a number or expression

#:#
: is a range of numbers

EXAMPLE:

.BLOCK [. 1000] <100> 0:3,10

Finds a block of memory between current micro PC and 1000, whose low order bits are 100 (binary), and for which the 0'th through 3'rd and 10'th following locations are unused.

.BLOCK [.] <0 1

Finds an even pair of memory locations, at some higher address

|L

6.3 .DEFINE MACRONAME ARG-BLOCK BODY-BLOCK

The .DEFINE pseudo op declares a macro. It must be followed by a syllable, the name of the macro being defined, and blocks. Every syllable in ARG-BLOCK becomes a dummy argument for the macro, and all occurences of dummy arguments are replaced by the actual arguments supplied when the macro is invoked This is the usual macro capability, without some of the flourishes that are available in MACRO-10 or FAIL. Every syllable in the argument list is an argument, so there should be no extraneous syllables (such as commas) separating the dummy arguments
EXAMPLE:

```
.DEFINE FOO [ A B ] [ A = B ] ; define the macro "FOO"
 FOO [ XYZ  0 ]               ; evaluates to " XYZ = 0 "
```

6.4 .FOR FORMAL ARG-BLOCK BLOCK

The .FOR construct provices an immediate execution macro capability, similar to the FOR construct in FAIL. The above example is equivalent to the following:

```
.DEFINE xxx [ FORMAL ] BLOCK
      XXX [ arg1 ]
      XXX [ arg2 ]
      ... and so on for each syllable in arg-block
```

Were XXX is a temporary macro. .FOR statements can be nested to any reasonable level. For example:

```
.FOR A [ B C ] [ .FOR D [ E F ] [ A^D = 0 ] ]
```

evaluates as: BE = 0 BF = 0 CE = 0 CF = 0

6.5 .IF EXPRESSION TRUE-BLOCK ELSE FALSE-BLOCK

The .IF pseudo op provides a mechanism for conditionally assembling code into a program. EXPRESSION can be any boolean expression. If the expression evaluates to TRUE, then TRUE-BLOCK is assembled, otherwise it is skipped. If the next syllable after termination of TRUE-BLOCK is "ELSE", then FALSE-BLOCK is scanned for and executed (or not) appropriately. .IFs can be nested to any reasonable level. See the LIST-OPERATORS pseudo ops for a list of the conditions available.
EXAMPLE:

```
|L
```

```
.IF A 0 [ B = B / A ] ELSE [ B = 0 ]
```

6.6 .INSERT Filespec

Causes the named file to be inserted into the assembly. The default extension is .SLO. .INSERTs can be nested to any reasonable level.

6.7 .MARKOUT #,#:#

.MARKOUT marks the microcode memory locations listed as USED. The format of the argument list is identical to the corresponding part of the .BLOCK pseudo op. Error messages are generated if any marked out loctions are already used, or if any subsequent attempt is made to use a marked out location.

Markout is intended to allow you to interdict memory locations that are nonexistant or dedicated to other use, so they will not be selected by the pc sequencing mechanisms.
|L

7.0 CONTEXTS

The grammar recognised by SLOE is context sensitive and syntax driven. Therefore, the kind of entity expected depends on the current context. Symbol tables are searched for in a predefined order in each context.

7.1 GLOBAL Context

Which is where you start, and where you return each time a $ is encountered. Here, one makes declarations, gives assembler directives, or starts assembling microcode words. In order of scan,

MACROS — Macros have highest priority. A macro with the same name as a pseudo op effectively replaces it.

PSEUDO OPS — pseudo ops are the means of directing the assembly. The pseudo ops will be described in detail later.

PCROUTS — Which are syntacticly equivalent to pseudos

FIELDS — microcode definitions

Note that numbers and numeric expressions are NOT acceptable in this context.

7.2 FIELDS

Whenever the name of a field is encountered, a special local context is entered, where a special set of field value names are recognised. In order of search:

FIELD VALUES associated with the field immediately following the FIELD pseudo op, or after a MODIFY-FIELD pseudo op

FIELD VALUES of other fields, allowed in this context by the ALLOW pseudo op

NUMERIC EXPRESSIONS

|L

7.3 DECLARATIONS

Declaration context is entered by the "=" pseudo op. Within declaration context, several additional symbol types are searched, and numeric expressions are allowed.

PCROUTS	a special set of symbols, corresponding to the algorithm for assigning the location of the next micro instruction to be assembled.
DEFOPS	a few special words like DEFAULT.

7.4 EXPRESSIONS

Numeric expressions are legal within declaration syntax, and secondarily, when no subfield is found to complete a field reference.

OPERATORS	operator symbols become available, and have first scan priority within expressions.
LABELS	labels defined with ":"
VALUES	integer symbols defined with =
NUMBERS	syllables that are logically numbers.
FIELDS	

7.4.1 "short" And "long" Word Names -

In one context, a shorter than complete name of a word is recognised. This provides a limited facility to have apparently the same word have different values in different contexts. The single quote character: "'" is treated as a terminator for a symbol name within the restricted context of a field requiring a value.

Example:

```
OPCODE = FIELD 5 5
 ADD'OP = 1
 SUB = 2 $
ADD = OPCODE 3 $
OPCODE[ADD] $
ADD $
```

In the example, OPCODE[ADD] is equivalent to OPCODE[ADD'OP], whereas ADD $ is equivalent to OPCODE[3] $

|L

ADD*OP is recognised "anywhere", but "ADD" is recognised as equivalent to "ADD*OP" only when the enclosing field "OPCODE" has been explicitly mentioned.

It is intended that names without imbedded "*"s be given only to symbols that are to be referencable from top level, without explicitly mentioning the enclosing field.

7.4.2 Optional Forms For Field Values -

There are three optional forms for specifying a field value word. From the above example:

```
SUB $

OPCODE SUB $

OPCODE[SUB] $


ADD*OP $

OPCODE ADD $

OPCODE ADD $
```

All three forms in each group are equivalent. Which form is used is a matter of preference, except when the field ALLOWS (see ALLOW Pseudo op writeup) more than one actual fields values to be specified. In that case, the bracketed form must be used is more than one value is actually specified.

|L

CONTEXTS

|L

Memory specifications

Sloe can simultaniously assemble code into any number of different memories, with different sizes and field properties. AT LEAST ONE memory must be declared before any significant assembling can be done.

DEFINE-MEMORY memory-name [subclass list] bit-size words-long

SUBCLASS LIST is a list of "extra" 1 bit fields (more later)
BIT-SIZE is the width of the memory, excluding the class list
WORDS-LONG is the length of the memory

declares a new memory and selects it to be current.

SELECT memory-name

selects a declared memory as current

LIST-MEMORIES

lists the properties of declared memories.

DEFAULT-PC xxx

selects mode xxx as the default next pc selector. See the PC CONTROL section of this document.

in addition to BITSIZE and WORDSIZE, each memory has other properties preserved over memory switches. These are:

LOCATION COUNTER
ALLOCATION MAP
FIELDS LIST
NEXT-PC SELECTOR
SUBCLASS LIST
PARITY GENERATION SPECIFICATION

NOT included in the per memory stuff is
VARIABLES LIST
MACROS LIST
LABELS LIST

Self Documentation Features

There are a number of psudo ops that dump documentation into the listing file. The documentation so dumped is gaurenteed to be current, since it is created at compile time and is an integral part of the assembler.

LIST-PSUDOS

.. is the most important, since it lists the list of lists. others that exist at the time this file was last updated are:

LIST-CORMAP	print memory allocation map of your assembled program
LIST-FIELDS	list all the field definitions very useful to see if everything is as you expect!
LIST-OPERATORS	list the arithmetic expression operators
LIST-MEMORIES	list name and sizes of defined memories
LIST-NEXTPC	list the available next-pc operators
LIST-LABELS	list label-to-address symbol table
LIST-SYMBOLS	list the integer symbols defined
LIST-PARITY	list the parity generation specification

SYMBOLS

SYMBOL TOKENS are bounded by separator characters.
the separator set is:
$: ; [ ] ( ) space tab line feed

the separator set does NOT include
< = > ! & / \ *

SYMBOL TOKENS can include any characters not in the separator set. as FIELD VALUES, tokens are scanned onlu as far as the first single quote (') character. This makes it convenient to define local names that possibly conflict with global names.
EXAMPLE:

```
                FIELD FOO s p $
                FIELD ABC s p
                  FOO'abc  = 3 $
```

now FOO references field FOO
ABC FOO references field ABC
FOO'ABC references field ABC

```
        EXAMPLE:
                ALU[A+B] $      is parsed as ALU [ A+B ] $
                ALU[A + B] $    is parsed as ALU [ A + B ] $

SYMBOL TYPES
        User defined symbols can be of several types

A:              defines label A.  Also, an undefined reference in a
                LABEL type field implicitly creates a label, whose
                value will be defined later.
A = expr        defines a simple variable.  NOTE that the space between
                the A and = is necessary!

A = FIELD s p $
                defines a FIELD within a microcode word.
                S is the size of the field in bits
                P is the number of the low order bit of the field,
                  with  bit zero at the left.
                Syntactically, a field defined this way requires
                a value specification to follow it.
A = FIELD s p
        VALUE1 = n
        VALUE2 = k
          ...
        VALUEN = z $

                defines a field which has named values.  The values
                may be used as value specification for the field, or,
                if unique, may be used to imply the field.

        FOR EXAMPLE:

        FOOBAR = FIELD 5 5              ;define a 5 it field ending 5 bits fro
                FOO-1  = 0              ;end of microcode word
                FOO-2  = 2 $            ;FOO-1 and FOO-2 are possible values
                FOOBAR 4 $
                FOOBAR FOO-1 $
                FOO-2 $                 ;all specify the FOOBAR field

        ****** COMPLEX FIELD DEFINITIONS *****


        special modifiers for field definitions.

LABEL           before the $ ending the field definition
                specifies that the value of the field is a label

                FOOBAR = FIELD 5 5 LABEL $
```

DEFAULT after a subfield specification specified that this is to be the field's default value.

```
FOOBAR = FIELD 5 5
        FOO-1 = 0
        FOO-2 = 1 DEFAULT
        FOO-3 = 2 $
```

NOVALUE after a field value definition, specifies that this cefinition does not supply a value for the field. This allows unrelated specifications to be included in the field for syntactic clarity

```
FOOBAR = FIELD 5 5
        FOO-1 = 0
        FOO-2 = 1
        STROBE = XYZ 3 NOVALUE
        FOO-3 = 3 $

FOOBAR [ FOO-2 STROBE ] $
```

restrictions and implied fields can be added to field specifications and field value specifications.

```
FOOBAR = XXX 3 YYY 4 FIELD 5 5
        FOO-1 = ZZZ[3] 0
        FOO-2 = 2 DEFAULT
        FOO-3 = XYZ 4 NOVALUE $
```

This specifies that FOOBAR (or subfields) set XXX field to 3 and YYY field to 4, as well as requiring a value for the field.
FOO-1 also sets ZZZ field to 3
FOO-3 also sets XZY field to 4, and specifies no value for FOOBAR

The "subclass list" of each memory is a list of names of 1 Bit fields, which are not part of the "real" microcode word. Unlike all fields which are part of the microcode word, class list fields do not have to be specified (or defaulted) in every microcode word. All the usual operations can be done with the class list fields. This permits you to set up arbitrary restrictions in microcode.

EXAMPLE:

```
Declare-memory main [ SPECIAL ] 10 1000

FOO = FIELD 4 3
   BAR = SPECIAL 1 1   ;special on
   BAZ = SPECIAL 0 1 $ ;special off
FOO2 = FIELD 4 7
   FOOZ = SPECIAL 0 1 $
```

```
;       NOW, the micorcode word
                FOO [BAR] FOO2[FOOZ] $
;       will cause an error message 'FOO2 fcauses a field conflict'
;       because FOO2[FOOZ] requires SPECIAL[0] and FOO[BAR] requires
;       SPECIAL[1]
```

Finally, fields can require another field's value be specified.

```
        @ = FIELD 12 30 LABEL $
        JCODE = FIELD 3 20
                JUMP = @ [] 7
                NEVER = 3 DEFAULT
                PUSHJ = @ [] 4
                POPJ = 5 $

                defines JCODE[NEVER] as the default specification,
                but if JUMP or PUSHJ is used, an address must be
                specified as well.

        JUMP FOO $
         and
        JCODE 4 @ FOO $
         .. are now equivalent.


A = BIT P V $
                is equivalent to:
                A-FIELD = FIELD P 1
                        A = V DEFAULT $
                if V is omitted, 1 is assumed
                This is a convenient mechanism for defining control
                bit fields.  All the other elaborations of FIELD
                definitions can be used too.
```

****

Memory allocation etc.

****

Before anything, you must declare the microcode wordsize

DECLARE-MEMORY memory-name bitsize wordsize

memory-name	is the name of the memory
bitsize	is the number of bits wide it is
wordsize	is the number of words long it is

any number of memories can be declared, and you can switch assembly into any of them with

SELECT memory-name

: nnn where nnn is a number, expression, or simple variable
sets the location counter to nnn
A: defines a new label at the current location counter

: A + nn sets the location counter to a relative location

.BLOCK [lb ub] <0x1> #,#,#:# ...
finds a block of memory locations, meeting constraints imposed

[lb ub]
specifies the bounds within memory what will be searched for a location that meets the other constraints imposed. LB and UB default to 0 and max-value respectively. LB and UB can also be an expression rather than a constant. The whole of [lb ub] is optional

<0x1> is entirely optional. if specified,
it is a picture of the low order adress bits of the block to be found, x specifys a "don't care" bit.
is a number or evpression
#:# is a range of numbers

.MARKOUT #,#:# ...
marks specified memory locations, relative to the current location counter, as used.

****

PARITY GENERATION

****

Each memory has a parity generation specification, which can be declared anytime before final output is written.

GEN-PARITY type mask-field data-field

TYPE	must be the word "ODD" or "EVEN"
MASK-FIELD	can be any field. The bits masked by the field are counted for the calculation
DATA-FIELD	can be any 1-BIT field. Data-field will be stuffed with the desired parity

There can be any number of different parity specifications; The calculations are performed in the order originally specified. It is possible, therefore, to autoatically caculate separate parity for different parts of the word, or even to generate error correcting codes!

The LIST-PARITY pseudo-op lists the parity generation sequence for the current memory.

****

Next PC control

****

The vagarities of PC control are handled through a special mechanism. There are a number 'next pc' selectors (PCCs) available, more will be generated on request. Currently available:

PC+1	unconditionally the next memory location
PC+20	unconditionally PC+20
PC-EVEN	the even location of a pair of free locations
PC-ANY	PC+1 if available, but any if not.

In genral, any set of constraints to the .BLOCK psudo op could be used. Any zero-argument macro which manipulates the location counter can be added to the list with the DECLARE-NEW-PC psudo-op

DECLARE-NEW-PC macroname newpc-name comment-to-end-of-line

macroname must be a zero-arg macro
newpc-name will be it's alias as a new-pc generator

the rest of the line is a comment, which will be printed by the LIST-NEXTPC psudo op

EXAMPLE:
DECLARE-NEW-PC FUBAR PC-MYWAY do it the right way!
will set up the macro FUBAR as PC-MYWAY in the PCROUTS list

At any moment, one of the PCCs is the default, set with the DEFAULT-PC psudo op.

```
        DEFAULT-PC  xxxx
```

sets the pc selector. The default cefault pc-selector is PC+1.

Now, any FIELD can imply a non-default pc selector. for instance:

```
FIELD PC-MOD 2 2
        NO-SKIP         = 0 default
        SKIP-USER       = PC-EVEN 1
        SKIP-EXEC       = PC-EVEN 2 $
```

now, asserting SKIP-USER in a microcode word also causes the sequencer to select an appropriately constrained location as the default next.

In adcition to selecting a default next pc, it is sometimes necessary to assure that that value is used in some label field. For instance, on FOONLY, the Next address field nust be filled. LABEL type fields are allowed to have a default value, and two useful special labels are available.

```
        .       is the current location counter
        NAF     is the default next location counter

FIELD NEXT-ADR 10 10
        NAF* = 0 DEFAULT $      ;default pc to NAFm whatever
                                ;that may evaluate to be.

FIELD NEXT-ADR-2 10 20
        .    = 1 DEFAULT $      ;default to .+1
```

****

Conditional compilation

****

DOESN'T WORK

.IF <boolean expression> <block> else <block>

Is the basic syntax for conditionals. the ELSE and the second block are optional.

Block can be either

```
BEGIN any-token
 ...
END any-token
```

or

```
[ ... ]
```

EXAMPLE:

```
.IF A > 0 [ B = 1 ] ELSE
        BEGIN FOO
        B = B + 1
        C = B * A
        END FOO
```

****

Repeat, For, Macros etc.

****

.REPEAT <expression> <block>
repeats BLOCK however many times, possibly zero.

EXAMPLE:

```
.REPEAT A + 3 [ B = B * 2 ]

.REPEAT 0 BEGIN FOO
        this is a random comment
          END FOO
```

.DEFINE <macroname> <arg-block> <body-block>

defines a macro.
MACRONAME is any valid token
ARG-BLOCK is a block. ALL the tokens within the block
are formal arguments, max of 127.
BODY-BLOCK is a block to substitute the formals in
the usual way.
` is the concatination character

EXAMPLE:

```
.DEFINE FOO [ A B ]
BEGIN XX
 A`B = A + B
END XX

.DEFINE FOO BEGIN ARGS A B C END ARGS
BEGIN BODY
        A B C $
END BODY
```

.FOR <formalname> <actuals> <body>

defines a repetitive replacememt of FORMALNAME
with each of the actials. Much like Fail.

EXAMPLE:

```
.FOR A [ B C D ]
 [ A = 0 ]
```

evaluates to

```
B = 0
C = 0
D = 0
```

****

Labels and expressions

****

Labels, unlike fieldnames, may be forward referenced; both in defined words and code words

```
        ERROR = JUMP BUGHLT $
```

is a valud defintion, even if BUGHLT is currently undefined

CAVEAT arithmetic performed with forwarc referenced lables is valid for simple + and - operations only!

```
        JUMP [FOO + 1]          is always valid
        JUMP [FOO ! 1]          is valid only if FOO is defined
```

!L

Special Notes for the export version

The ususal commands begin with a :

|D (control D) calls DDT, return with |E

!foo a b c d (up to four arguments) calls trip function FOO

TAPE output format is 8 bit bytes, (tape frames)
2000. frames in a record, changable by setting BUFSIZ

<adr-high><adr-low> <8bits data><8bits cata> ...<8bits data>
enough 8bit data bytes to contain the whole word, however long
that may be

microcode word groups are output in the order encountered in the
assembly

file is terminated by an adr of 177777