THE MIDAS ASSEMBLY PROGRAM

November 26, 1962
THE MIDAS ASSEMBLY PROGRAM

Introduction

Programming for a digital computer is writing the precise sequence of instructions and data which is required to perform a given computation. The purpose of an assembly program is to facilitate programming by translating a source language, which is convenient for the programmer to use, into a numerical representation or object program, which is convenient for the computer hardware to deal with. A symbolic assembly program such as MIDAS permits the programmer to use mnemonic symbols to represent instructions, locations, and other quantities with which he may be working. The use of symbolic labels or address tags permits the programmer to refer to instructions or data without actually knowing or caring what specific location in the computer memory they may occupy.

MIDAS is a two pass assembler; that is, it normally processes the source program twice. During the first pass, it enters all symbols definitions encountered into its symbol table, which it then uses on Pass 2 to generate the complete object program.

The MIDAS Source Language

A program consists of a sequence of numbers in memory which may be instructions, data, or both. We shall refer to these numbers as words without specifying whether they are instructions or not. A word is denoted in the source program by one or more syllables separated by suitable combining operators, and terminated by a tab or carriage return. A syllable may be defined as being the smallest element of the programming language which has a numerical or operational value. The following are some different types of syllables:
1. Integers. An integer is a string of digits, which will be interpreted as an octal or decimal number.

2. Symbols. A symbol is a string of alphanumerical characters (lower case numerals and letters, and upper case letters except U, I, S, and X) containing at least one letter. The first six characters of a symbol are used to identify it if it is more than six characters long.

Syllables may be combined with the following operators:

+ or space means addition, modulo $2^{18} - 1$ (one's complement).
- means addition of the one's complement.
U means logical union (inclusive or).
I means logical intersection (logical and).
S means logical disjunction (exclusive or).
X means integer multiplication.

A symbolic expression is one syllable, or more than one syllable combined with these operators. We shall refer to +, -, and space as additive operators, and U, I, S, and X as product operators.

Operations are performed from left to right, except all product operations are performed before additive operations. It is not admissible to precede or follow a product operator with any other operator. In a string of consecutive additive operators, the last one seen applies.

The following examples of symbolic expressions on the left have the value listed on the right. (All numbers in this report are octal unless followed by a decimal point ".")
A complete list is given in Appendix A. Additions to pseudo-instructions will be discussed at opportune places.

Denote the end of the program.

Start

Decrement numbers until next appearance of pseudo-instruction.

All integers following are interpreted as decimal.

Action

Decrement next appearance of pseudo-instruction.

All integers following (unless specified otherwise) are interpreted as decimal.

P-1

Proceed with the assembly. Some examples of pseudo-instructions are:

The assembly need not represent directions to MDAS on how to use pseudo-instructions which represent directions, and also a class of symbols called common symbols. The syntax vocabulary consists of symbols for the more straightforward symbols. A more advanced vocabulary is needed.


Assumed, the location counter is advanced by one.

After each word is determined by a location counter in MDAS, the location in memory to which it is assigned is a storage word. The location in memory to which it is assigned is a symbolic expression terminated by a tab or carriage return.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>700340</td>
<td>7</td>
</tr>
</tbody>
</table>

-3-
Symbols are defined in the following ways:

1. As address tags. A comma following a symbolic expression denotes an address tag. If the tag is a single, undefined symbol, it will be defined with a numerical value equal to the present value of the location counter. If the tag is any other defined symbolic expression, it will have its value compared with the present value of the location counter, and an error comment (mdt) will be made in the event of a disagreement. If the tag is any other symbolic expression which is undefined when encountered on Pass 2, an error comment is made (ust). Use of a defined symbol as an address tag cannot change the value of the symbol.

2. By parameter assignments. A symbol may be assigned a numerical value by the use of a parameter assignment. The form

   symbol=expr

where symbol is any legal symbol and expr is any symbolic expression terminated by a tab or carriage return, defines symbol as having the numerical value of expr. Parameter assignments may be used to set table sizes, define new operation codes, or for other purposes. Thus

   cle=caluxro

defines cle as 700201, which, as an operate instruction, would clear the AC, LR, and XR.

3. As variables. The appearance of a letter or letters in upper case in any legal, undefined symbol, at any up-
pearance of that symbol, defines that symbol as a variable. For each such symbol defined, one register is allocated in a region of storage reserved by the next appearance of the pseudo-instruction variables. The initial contents of these registers is undefined. This feature facilitates the reserving of temporary storage locations. Example:

```plaintext
sto temp
tsx subr
lda Temp
```

variables

Beware: Do NOT try to use upper case letters U, I, S, or X to denote a variable!

4. As macro instructions. A symbol is defined as a macro-instruction name by use of the pseudo-instruction define. Further discussion of macro instructions will be left until later.

5. With equals or opsyn. A symbol may be defined as precisely equivalent to any other symbol by use of the pseudo-instruction equals and opsyn. The usage is:

```plaintext
equals anysym, defsym;
```

or

```plaintext
opsyn anysym, defsym;
```

where the symbol anysym is made logically equivalent to defsym if the latter is defined. Previously defined symbols are redefined. Equals and opsyn differ in one
respect: `opsyn` is effective in Pass 1 only. These may be used to define a logical equivalent for any other defined symbol. Thus abbreviations may be defined for pseudo-instructions if desired. Note that `equals` and `opsyn` are NOT the same as the equals sign used in parameter assignments, and are not in general interchangeable with it. `Equals` and `opsyn` are used to give a symbol a **logical** or **operational** value, while parameter assignments are used to give a symbol a **numerical** value. Beware that if you define synonyms for `start` that either the synonym starts with the letters _s,t,a,r,t_ or the word `start` appears after the use of the synonym at the end of the source program tape. Although the main processor in MIDAS will recognize synonyms for `start`, the part of the program which reads tape will not, and must be fooled into stopping the tape reader independently of the rest of the assembly.

**The Location Counter**

The MIDAS location counter records the assigned location for each word in the object program. It is set to 20 at the beginning of each pass, and counts upward modulo memory size. The location counter may be set to any value by writing

```plaintext
expr
```

This sets the location counter to the value of the symbolic expression `expr` modulo $2^{13}$. If `expr` contains an undefined symbol, on Pass 1 the location becomes **indefinite**, and the definition of address tags is inhibited until the location again becomes definite by means of a defined location assignment. On Pass 2, an undefined symbol
will result in an error message (u31). The undefined symbol is taken as zero, and the location remains definite. The pseudo-instruction variables may not be used when the location is indefinite.

The value of the location counter may be obtained by using the special syllable "." (period). Examples:

```
tze .+2  com
tra nonzero  trn -.1
llr foo
```

The first example transfers to location nonzero if the AC contains any number other than zero, but zero in the AC causes the program to skip to the llr instruction. The second example puts the magnitude of the contents of the AC into the AC by transferring back to the complement instruction until the AC becomes positive. The second instruction is read "trn point minus one."

The character ".", when preceded by an integer, denotes that the integer is to be considered as decimal regardless of the effect of the pseudo-instruction octal or decimal. "Point" means location counter only when it appears as a distinct syllable. Thus,

```
add .  means add this instruction to AC
20.  means 20 decimal.
```

The character ⟨⟩, when not preceded by an expression, denotes the beginning of a comment. Characters following it are ignored until the next tab or carriage return.

**Constants**

Constants required by a program will be reserved automatically by MIDAS when enclosed in parentheses. Thus, if it is required to get the number add 20 into the accumulator, one can write

```
lda (add 20)
```
The word enclosed in parentheses is stored in a block reserved by the
next appearance of the pseudo-instruction constants. Duplicate con-
stants are stored only once. Closing parens will be supplied auto-
matically by MIDAS if the character following is a word terminator
(e.g., tab or carriage return). The constant word and surrounding
parens are treated as a single syllable whose value is the address
of a register containing the constant word. Constants may be used
in constants. The following two program fragments are equivalent:

```
add (add (20)-llr-(30

```

```
add a
```

```

```
constants

```

```

```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Thus my be any character not appearing in the string. Neither appearance of q is considered part of the string. Thus q may be any character not appearing in the string.

The following examples demonstrate their usage.

<table>
<thead>
<tr>
<th>Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf</td>
<td>is equivalent to 11010</td>
</tr>
<tr>
<td>mm</td>
<td>&quot; &quot; 222000</td>
</tr>
<tr>
<td>flexo</td>
<td>&quot; &quot; 100000+202000+004400=306400</td>
</tr>
<tr>
<td>text .this.</td>
<td>&quot; &quot; (306400)</td>
</tr>
</tbody>
</table>

**Macro Instructions**

Often certain character sequences appear several times throughout a program in almost identical form. The following example illustrates such a repeated sequence.

```
lda a
add b
sto c
lda d
add e
sto f
```

The sequence:

```
lda x
add y
sto z
```

is the model upon which the repeated sequence is based. This model can be defined as a macro instruction and given a name. The characters x, y, and z are called dummy arguments, and are identified as such by being listed immediately following the macro name when the macro instruction is defined. Other characters, called arguments, are substituted for the dummy arguments each time the model is used. The appearance of a macro-instruction name in the source program is
referred to as a call. The arguments are listed immediately follow-
ing the macro name when the macro instruction is called. When a
macro instruction is called, MIDAS reads out the characters which
form the macro-instruction definition, substitutes the characters of
the arguments for the dummy arguments, and inserts the resulting
characters into the source program as if typed there originally.

The process of defining a macro is best illustrated with an
example:

```
define write a,b
  tsx wr
  b=-2
  text [a]
  b, terminate
```

The pseudo-instruction define defines the first legal symbol
following it as a macro name. Next follow dummy arguments as re-
quired, separated by commas, terminated by a tab or carriage return.
Next follows the body of the macro definition. Appearances of dummy
arguments are marked, and the character string is stored away. Dummy
arguments are delimited by the following characters: plus, minus,
space, U, I, S, X, upper case, lower case, tab, carriage return,
equals, comma, bar, colon, and upper case 1, 6, and 9. Dummy argu-
ments must be legal symbols; any previous definition of dummy argu-
ment symbols is ignored while in the macro definition.

A macro call consists of the macro name, followed if desired
by a list of arguments separated with commas, and terminated with a
tab or carriage return. The write macro, if called as follows:

```
write 'This gets printed out., nextag
```
generates the following code:

```
tsx wr
nextag=-2
text 'This gets printed out.'
nextag
```
which, with a suitable text-printing subroutine, might comprise the necessary code for printing "This gets printed out." on the flexo-writer. The argument to be printed, using this format, must not contain the characters comma, tab, carriage return or bar. Comma, tab, or carriage return would end the argument while bar would terminate the argument of the text pseudo-instruction. So that comma, tab, and carriage return can be used within arguments, the argument quotation characters upper case and are provided. They might be used as follows:

```
write "This, of course, has commas."
It also has a carriage return, next tag
```

All characters within a pair of argument quotes are considered to be one argument, and this entire argument, with the quotes removed, will be substituted for the dummy argument in the original definition. MIDAS marks the end of an argument only on seeing comma, tab, or carriage return not enclosed within argument quotes. If quotes appear within quotes, the outermost pair is deleted. If an outer argument quote is immediately preceded by an upper case and immediately followed by a lower case, both case shifts are deleted also. A tab or carriage return immediately following a macro name denotes that no arguments are read. Any other separating character will be the first character of the first argument except space: a space used as a separator will be deleted and will not be part of the first argument.

The second argument of the write macro is a symbol which is defined as an address tag each time the macro is called, so a different symbol must be supplied at each call of the macro to avoid multiply defined tags. MIDAS will supply suitable created symbols for
this purpose, guaranteed to be unique to each call of the macro, if we write the first line of the definition thusly:

    define write a|b or define write a,|b

In either case, the vertical bar denotes dummy symbols following it will be supplied from special created symbols if not explicitly supplied when the macro is called. The created symbols are of the form 000a01, 000a02, ... 000a09, 000a0a, etc. The created symbol generator is reset to 000a0: at the beginning of each pass. The number of created symbols may not exceed 33,695. Note that unsupplied arguments corresponding to dummy arguments preceding the bar are plugged in as empty strings. Supplied arguments corresponding to dummy arguments following a bar suppress the generation of a corresponding created symbol.

There remains one problem: How do we plant dummy arguments in the argument of character r, m, or l? Of course, the r, m, or l could be part of the supplied argument, but there is another way. Write, say:

    define macro a
    .
    .
    .
    add (charac r)a |note charac ra does not work as
    .
    .   ra is not a dummy argument
    .
    .

The sequence upper case, l, lower case is deleted during the macro definition, but causes the macro scan to search on each side for dummy arguments. In this case, a is found to be a dummy argument, and is treated accordingly. If the upper case l is not both preceded and followed by case shifts, only the l is deleted.
Example:

```
define type x464pq
   lda (charac r1 x464pq
   pno
   terminate

type f   gives   lda (charac rf
   pno
```

How may one cause a created symbol to define a variable? It will not do to write the dummy argument in upper case, for then the created symbol would be in upper case. Since upper case numerals are not legal symbol constituents, created symbols must not appear in upper case. The solution is to append a suitable upper case letter, say z, to the dummy argument.

Example:

```
define macro [a
   sto aZ        | case shift makes end of
   tax subr     | dummy argument a
   lda aZ
   terminate
```

The variables would then be of the form 000a01Z, 0000a02Z, etc. which are perfectly legal and unique variables.

Created symbols have been introduced to solve the problem of address tags within macro definitions, but they may be used in other ways also. Some examples are given in Appendix 2.

Macro definitions may contain other macro definitions or macro calls. Arguments of the macro being called may be used in the macros it calls or defines with perfect generality. As an example, let us rewrite the write macro so that it inserts a suitable text printing subroutine into the object program at its first
call, and then redefines itself so that later occurrences call the subroutine. This might be done as follows:

define write a
define write c|d |redefines write when called first time
tsx wr
d...-2
text |c|
terminate write

write "as
tra zzxgwq
|calls new definition

wr, llx 0 |text printing subr
ixl
lpkh,
ixl
lax 1
aux .-1
pnt
pnt
prtlxl
ixl lpkh
ixl
tr: 1
zxwq, terminate

Notice that address tags in the text printing subroutine need not be created symbols, as the tags appear only at the first call of write. They must not, of course, conflict with tags used elsewhere in the program, and to insure this, created symbols may be used if desired. Notice that, in this example, the pseudo-instruction terminate has been supplied with an argument: the name of the macro being defined. If terminate is followed by a space, it will expect to find this argument, which it will compare with the name of the macro being defined. Unless they agree, an error comment (mrn) will be made. This permits the programmer to be sure that his defines and terminates count out correctly. An additional aid in this respect is the fact that terminate is undefined outside a macro definition.

Arguments can, by judicious use of argument quotes (see example below), contain sub-arguments. A pseudo-instruction irp
(indefinite repeat) permits the analysis of such an argument. The pseudo-instruction \texttt{irp} in the macro definition takes one argument, namely, the dummy argument corresponding to the argument to be analyzed. When the macro instruction is called, the characters following the argument of the \texttt{irp} until the next matching \texttt{endirp} will be inserted once into the program for each sub-argument in the argument being analyzed, and the sub-arguments will be substituted for the corresponding dummy argument. Example:

\begin{verbatim}
define sum a,b,c
   lda a
   irp b
   add b
   endirp
   sto c
   terminate

sum j, k, l, m, n
\end{verbatim}

\texttt{gives:}

\begin{verbatim}
   lda j
   add k
   add l
   add m
   sto n
\end{verbatim}

It is quite permissible to have \texttt{irp}'s within an \texttt{irp}, analyzing either the same or different arguments. The pseudo-instructions \texttt{irp} and \texttt{endirp} are defined only within a macro definition. If an \texttt{irp} analyzes a null string, the characters in the range of the \texttt{irp} will be inserted once, and null string will be inserted for the subargument.

\textbf{The Garbage Collector}

When MIDAS redefines a macro, the space in the macro instruction table used by the old definition will be recovered, if necessary, by a garbage collector. It is important in a long program to insure that unused macro definitions are abandoned, that is, that their names
are caused to refer to something else other than the original macro definitions. A suitable "something else" is the pseudo-instruction null, which does absolutely nothing. Thus if a macro called foo has been defined, it may be discarded after its last usage by saying:

```
equals foo, null
```

which will make the space used by foo recoverable. The garbage collector is called whenever the combined macro and symbol tables are exhausted. If no space can be recovered, an error comment is made (see).

**Repeat**

The pseudo-instruction repeat expr, anything, where expr is a symbolic expression defined on Pass 1 and anything is any string of characters terminated by a carriage return, causes anything to be inserted into the program a number of times, called the count, equals to the value of expr. The anything, called the range of the repeat, can be storage words, parameter assignments, macro calls (if not containing carriage return in an argument), other repeats, or anything else. If repeat is used in the range of a repeat, both repeats will end on the same carriage return. Repeat may be used in macros, and dummy arguments may appear either in the range or the count of the repeat, or both. If the count of a repeat is zero or negative, the range of the repeat is ignored.

**Dimension**

The pseudo-instruction dimension may be used to allocate space for arrays. The statement

```
dimension name1(size1), name2(size2),...
```
causes space to be reserved in the variables storage for the array names specified. Each name is defined as the location of the first of the block of registers of the length specified. The array names must not have conflicting definitions elsewhere, and the array sizes must be defined at their occurrence on Pass 1.

**Conditional Assembly**

It is often useful, particularly in macro instructions, to be able to test the value of an expression, and to condition part of the assembly on the result of this test. For this purpose the pseudo-instruction `11f` and `01f` are provided. Following the pseudo-instruction name there is a symbol called a **qualifier** that determines the type of test; and then an argument that is tested according to the qualifier. The argument is ended by any of the word terminators tab, carriage return, comma, or slash. All these terminators except slash do what they would have done had the conditional not been present; but slash only marks the end of the conditional, which is treated as a single syllable whose value is one or zero. Examples:

```plaintext
repeat 01f vp x+1, macro arg1,arg2]
a=11f vzxI600000—
sto p+11f vp-s|X2]
```

The value of `11f` is one if the condition tested for is true, and zero otherwise; while the value of `01f` is zero if the condition tested for is true, and one otherwise. There are at present two qualifiers with two corresponding tests:

- **vp**: If the value of the expression following is positive or zero (either plus or minus), the test is true.
- **vz**: If the value of the expression following is zero, the test is true.
The first example calls the macro if \(x \geq -1\). The second example defines \(a\) as one if the two high bits of \(x\) are both zero; otherwise \(a\) is defined as zero. The third example generates \(\text{stop}\) if \(s\) is positive, and \(\text{stop} + 2\) if \(s\) is negative. It could also be written as:

\[
\text{stop} + 2\text{X01f vps0}
\]

Conditionals may be used in or out of macros, but may not contain other conditionals.

The Source and Object Programs

A source program for MIDAS consists of one or more flexo tapes, each with a title, a body, and a \(\text{start}\) pseudo-instruction. The title is the first string of characters and is terminated by a carriage return. Carriage return and stop codes preceding the title are ignored. The body is the storage words, macros, parameter assignments, etc. which make up the substance of the program. It may be void. The \(\text{start}\) pseudo-instruction denotes the end of the source program tape. It takes one argument, which specifies the first instruction to be executed in the object programs. \(\text{Start}\) must be preceded by a tab or carriage return, and followed (after the argument, if supplied) by a carriage return. READ THE LAST SENTENCE AGAIN. In spite of all warnings, the number of people who omit the carriage return after \(\text{start}\) is amazing. Therefore, take heed.

MIDAS will normally punch a binary object program during Pass 2 of an assembly. It will contain a title in readable characters, consisting of the visible characters in the title except those following (and including) an equals sign. Next will be punched an input routine, which is a loader that reads in the rest of the tape, and which is itself read in by the TX-0 \text{read in mode}. The binary
output from the body of the source program is punched in blocks of up to 100 registers. The end of the binary tape is denoted by a **start** block, which is produced by the pseudo-instruction **start**. The start block may be of two types:

1. The **add** start block causes the input routine to stop, and pressing **Restart** transfers to the address specified. It is punched by **start addr**, where **addr** is a symbolic expression whose value specifies the starting address. MIDAS adds **add** to this and punches it on the tape.

2. The **trn** start block causes the input routine to transfer at once to the address specified. In this case the argument of start must have the value of **add addr** where **addr** is the address in question. MIDAS adds **add** (=200000) to this, giving **trn** (=400000) and punches it on the tape.

The format of the output is subject to considerable control by the programmer. The pseudo-instruction **noinput** suppresses punching the input routine. The pseudo-instruction **readin** suppresses the input routine and punches in readin mode until the next encountering of the pseudo-instruction **noinput**, which resumes punching in input routine format. The normal input routine occupies registers 17756-17777, but an input routine occupying registers 0-22 will be supplied by the pseudo-instruction **frontloading**, which, if used, must be the first thing on the English tape (after the title, of course). This pseudo-instruction causes the location counter to start at 30 instead of the usual 20.

For fabricating special tape formats or punching start blocks without stopping the assembly, the pseudo-instruction **word** is provided. Its argument or arguments, separated by commas and ended by
a tab or carriage return, are punched directly on the object program tape, and do not affect the location counter.

The tape formats discussed so far are characterized by having a specific location in core assigned for each word in the object program. MIDAS will also produce relocatable tapes, which, by means of a special loader, may be placed anywhere in memory. Before using this feature, described in the next section, the reader is advised to familiarize himself with Memorandum M-5001-34, which describes the relocatable loader and relocatable system.

**Relocatable Programming**

The pseudo-instruction **relocatable** directs MIDAS to assemble the object program in relocatable format and sets the location counter to relocatable 0. Address tags will be defined as relocatable symbols (relocation count +1) as long as the location is relocatable. Symbols defined by parameter assignment will have a relocation count equal to that of the expression to the right of the equal sign except that no symbol may have a relocation count exceeding one in magnitude. A location assignment puts the location to relocatable or absolute according to whether the relocation count of the location assignment is +1 or 0. **Relocatable** also suppresses punching an input routine, replacing it with a word `trn 17000`, which, when executed in the readin mode, transfers control to the entry of the BRS Loader. Storage words in relocatable mode may have relocation +1, -1, or 0; words in absolute mode may have relocation 0 only.

The pseudo-instruction **exit** is used to define symbols which are external to the program being assembled. The usage is

```
exit s1, s2, s3, ...
```
which enters the symbols s1, s2, ... in the transfer vector and defines them as the addresses they occupy there. Only the first three characters of these symbols are significant to the relocatable loader. These symbols must not be defined with a conflicting definition elsewhere or an error message (mdx) will be produced.

The pseudo-instruction entry is used to denote points in the program to which external programs may transfer control. The usage is:

    entry s1, s2, s3, ...

where the symbols s1, s2, ... must be defined as address tags elsewhere in the program. The symbols so declared are entered in the program card. Again, only the first three characters of such symbols are significant to the relocatable loader. For a program with both primary and secondary entries, the pseudo-instruction entry is used twice consecutively, first listing the primary and then the secondary entries. To the extent that the pseudo-instructions relocatable, entry, and exit are used, they must be used in that order, and no storage words may intervene between them. A program with no entry specified is a main program, and the pseudo-instruction exit will cause a program card to be punched with a name of +0, as required by the BRS loader. If neither entry nor exit is used, no program card will be provided. Since any program to be loaded by the BRS Loader must have a program card, it has been made possible to get a program card with a main program entry by using the pseudo-instruction entry with no arguments. The maximum number of arguments of entry is 37; there is no limit on the number of arguments of exit.

In relocatable programs, the pseudo-instruction noinput will suppress punching the word trn 17000 at the head of the object program tape.
MIDAS has few requirements on format. The user should be aware of the following:

1. Carriage returns and tabs are equivalent except in the title, in the range of a repeat, and after start. Extra tabs or carriage returns are ignored.

2. Backspace, the upper case numerals except 1, 6, and 9, and the unused characters of the flexo code, including blank tape with only the seventh hole punched, are illegal except in arguments of flexo code pseudo-instructions.

3. Stop codes and color shifts are ignored except in arguments of flexo code pseudo-instructions. Upper case 1, 6, and 9 are similarly ignored when not in macro calls or definitions.

4. Deletes are always ignored.

Many programmers have found that adherence to a fairly rigid format is of help in writing and correcting programs. The following suggestions have been found useful in this respect:

1. Place address tags at the left margin, and run instructions vertically down the page indented one tab stop from the left margin.

2. Surround address tags with color shifts. It looks nice.

3. Use only a single carriage return between instructions, except where there is a logical break in the flow of the program. Then put in an extra carriage return.

4. Forget that you ever learned to count higher than three; let MIDAS count for you. Do not say stop; use an ad-
5. Organize the program by pages, separating each page of flexo tape with a stop code and some tape feed. Let the page boundaries coincide with logical divisions of the program if possible. Fixing one bad page and splicing in a new one takes about as much time as reproducing two pages of program, so learn to splice tape.

6. Have the typescript handy when assembling or debugging a program, and note corrections in pencil thereon as soon as you find them.

**Performing an Assembly**

First read in MIDAS. Turn on the on-line flexowriter and press **Start Read**. Set the TBR to trn 20 and the TAC to 0. Load the first source tape into the reader and press **Restart**. MIDAS will read the tape in sections of about two pages each, and will stop shortly after reading **start** at the end of the tape. To process additional tapes after the first, press **Test**. Now begin Pass 2 by loading the first tape and pressing **Restart**. For additional tapes, press **Test**. At the end of Pass 2, press **Restart** again to secure a start block. Tapes should be processed in the same order on both passes.

The normal operation of MIDAS may be summarized by the following table:

<table>
<thead>
<tr>
<th>Condition</th>
<th>AC</th>
<th>IR</th>
<th>MBR</th>
<th>Action on Restart</th>
<th>Action on Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIDAS or symbol punch read in</td>
<td>0</td>
<td>-0</td>
<td>-0</td>
<td>Begin Pass 1</td>
<td>Begin Pass 2</td>
</tr>
<tr>
<td>End of tape, Pass 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Begin Pass 2</td>
<td>Continue Pass 1</td>
</tr>
<tr>
<td>End of tape, Pass 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Punch start block</td>
<td>Continue Pass 2</td>
</tr>
<tr>
<td>After start block</td>
<td>0</td>
<td>-0</td>
<td>-0</td>
<td>Restore, begin</td>
<td>Begin Pass 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pass 1</td>
<td></td>
</tr>
<tr>
<td>Error stop</td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>Continue, suppress</td>
<td>Continue Pass</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>punching</td>
<td></td>
</tr>
</tbody>
</table>
The normal sequence of operations above can be modified by use of the TAC. Whenever Test is pressed, bit 0 of the TAC is examined. If it is zero, the normal sequence is followed; if it is 1, the next 6 bits of the TAC are examined. These control:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pass 1 if 0, pass 2 if 1.</td>
</tr>
<tr>
<td>2</td>
<td>Begin pass if 0, continue pass if 1.</td>
</tr>
<tr>
<td>3</td>
<td>If 1, punch if pass 2; if 0, do not punch.</td>
</tr>
<tr>
<td>4</td>
<td>If 1, punch input routine if punching; if 0, no input.</td>
</tr>
<tr>
<td>5</td>
<td>If 1, punch title if punching; if 0, no title.</td>
</tr>
<tr>
<td>6</td>
<td>If 1, restore symbol table to initial symbols and pseudo-instructions.</td>
</tr>
</tbody>
</table>

**Error Stops**

MIDAS will complain about various ambiguities and error conditions found in source programs. Some of these have already been mentioned. An error listing has the following format:

Column 1: A three letter code describing the type of error. A number following is the depth of macro calls.

- 2: The octal location in the object program. The symbol \( R \) means relocation.

- 3: The symbolic location, in terms of the last address tag seen.

- 4: The last pseudo- or macro-instruction name seen.

- 5: The offending symbol, if a symbol was in error.

MIDAS will ignore most errors (with exceptions noted below) and will continue the assembly if Restart or Test (with TAC 0=0) is pressed; the two are equivalent except Restart will discontinue punching on Pass 2 if it was in progress. Turning up TAC 17 is equivalent to pressing Restart after an error stop. This bit is independent of the rest of the TAC.
The error conditions are:

**us-** : In general, undefined symbol. Undefined symbols are evaluated as 0. The third letter tells where it was found:

- **w**: In a storage word or argument of pseudo-instruction word.
- **m**: In a storage word generated by a macro call.
- **d**: In the size of a dimension array.
- **p**: In a parameter assignment.
- **c**: In a constant.
- **s**: In the argument of start.
- **e**: In an argument of entry.
- **r**: In the count of a repeat.
- **t**: In an address tag of more than one syllable. This will frequently be the result of an undefined macro instruction.
- **i**: In an argument of `0if` or `1if`.

**ich** Illegal character. The bad character is ignored.

**ilf** Illegal format. Some character or characters were used in an improper manner. Characters are ignored to next tab or carriage return.

**ile** Illegal entry. Argument of entry is improper and will be ignored.

**ilx** Illegal exit. Argument of exit is improper and will be ignored.

**ir-** : Illegal relocation. The relocation is taken as 0. The third letter identifies where it was found, and will be the same as listed under undefined symbols (above).

**mnd** : Macro name disagrees. The argument of terminate disagrees with the name of the macro being defined. First name is used.

**mdt** : Multiply defined tag. Original definition retained.

**mdx** : Multiply defined exit. An argument of exit is previously defined with a conflicting value. Original definition retained.

**mdv** : Multiply defined variable. A symbol containing an upper case letter is previously defined as other than a variable. Original definition retained.
mdd: Multiply defined dimension. An array name in a
dimension statement has a conflicting definition.
Original definition retained.

ipa: Improper parameter assignment. The expression to
the left of an equal sign is improper. The assign-
ment is ignored.

sce: Storage capacity exceeded. Assembly cannot continue.

tmc: Too many constants: the pseudo-instruction constants
used more than 10. times in one program.

tmp: Too many parameters: the storage reserved for macro
instruction arguments has been exceeded.

tme: Too many entries. Maximum number of arguments of an
entry pseudo-instruction is 37 octal.

tmv: Too many variables. The pseudo-instruction variables
has been used more than 8 times in one program.
Assembly cannot continue.

cld: Constants location disagrees. The pseudo-instruction
constants has appeared on Pass 2 in a different lo-
cation from that found on Pass 1, meaning all the
constants syllables have been assigned the wrong
value. Assembly cannot continue.

cld: Variables location disagrees. The pseudo-instruction
variables has appeared on Pass 2 in a different lo-
cation from that found on Pass 1. The condition is
ignored.

iae: Internal assembler error. MIDAS has found that it
has made a mistake in assembling the program. De-
liver the error message and a copy and listing of
the source program to a member of the TX-0 staff so
that the trouble may be found. Assembly cannot con-
tinue. The octal location given is the location in
MIDAS where the error was found.

Troubleshooting

The checking features built into MIDAS will detect simple
efforts like forgotten tags very simply. Attempting to debug complex
macro definitions from error messages and binary output is a much
more difficult proposition. Special aids have been provided to sim-
plify this.
1. The pseudo-instructions \texttt{print} and \texttt{printx} take an argument exactly like \texttt{text}, which MIDAS will print out online during the assembly process. \texttt{Printx} prints just the argument and a following carriage return, while \texttt{print} precedes this with the first three columns of an error listing, with the "error" code \texttt{mnt}. The argument of \texttt{print} or \texttt{printx} may contain dummy symbols if used in a macro definition.

2. Bit 16 of TAC when on, causes MIDAS to print out online every character it processes, including all macro expansions. This permits the programmer to let MIDAS do the bookkeeping when testing a complicated macro.

\textbf{Symbol Punch and Symbol Print}

A record of symbol definitions may be printed out by reading in MIDAS SYMBOL PRINT. An alphabetic or numeric order listing may be secured by reading in the appropriate tapes.

A punched record of symbol and/or macro instruction definitions may be obtained by use of MIDAS SYMBOL PUNCH. When \texttt{SYMBOL PUNCH} is read in, it will feed some blank tape and listen for a title. Type a title on the typewriter. To obtain both symbol and macro-instruction definitions, terminate the title with a carriage return. For symbols only, terminate with a tab, and then type "s" followed by a carriage return. For macro definitions only, terminate the title with a tab, followed by "m" and a carriage return.

The symbol punch so obtained may be used with DOCTOR for symbolic debugging, or read into MIDAS at a later time for assembling patches.
or the like. When a symbol punch is read into MIDAS, TAC 6 is examined. If off, the symbols from the symbol punch are merged with any existing symbol table. If on, the symbol table is restored to the initial vocabulary before merging the symbol punch.

Robert A. Saunders

Approved

J. B. Dennis
Appendix 1. MIDAS Initial Vocabulary.

Part 1. Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>200000</td>
</tr>
<tr>
<td>adi</td>
<td>060000</td>
</tr>
<tr>
<td>adx</td>
<td>220000</td>
</tr>
<tr>
<td>ala</td>
<td>640260</td>
</tr>
<tr>
<td>all</td>
<td>640230</td>
</tr>
<tr>
<td>alp</td>
<td>640220</td>
</tr>
<tr>
<td>alr</td>
<td>640200</td>
</tr>
<tr>
<td>alx</td>
<td>640031</td>
</tr>
<tr>
<td>amx</td>
<td>640050</td>
</tr>
<tr>
<td>ana</td>
<td>740027</td>
</tr>
<tr>
<td>anl</td>
<td>640207</td>
</tr>
<tr>
<td>ano</td>
<td>740207</td>
</tr>
<tr>
<td>arx</td>
<td>640601</td>
</tr>
<tr>
<td>aux</td>
<td>260000</td>
</tr>
<tr>
<td>axc</td>
<td>640061</td>
</tr>
<tr>
<td>axo</td>
<td>640021</td>
</tr>
<tr>
<td>axr</td>
<td>640001</td>
</tr>
<tr>
<td>bsr</td>
<td>604000</td>
</tr>
<tr>
<td>cal</td>
<td>700200</td>
</tr>
<tr>
<td>cax</td>
<td>700001</td>
</tr>
<tr>
<td>cla</td>
<td>700000</td>
</tr>
<tr>
<td>clc</td>
<td>700040</td>
</tr>
<tr>
<td>cll</td>
<td>631000</td>
</tr>
<tr>
<td>clr</td>
<td>632000</td>
</tr>
<tr>
<td>com</td>
<td>600040</td>
</tr>
<tr>
<td>cpr</td>
<td>607000</td>
</tr>
<tr>
<td>cpy</td>
<td>620000</td>
</tr>
<tr>
<td>cry</td>
<td>600012</td>
</tr>
<tr>
<td>cyx</td>
<td>640030</td>
</tr>
<tr>
<td>cyr</td>
<td>600600</td>
</tr>
<tr>
<td>dis</td>
<td>622000</td>
</tr>
<tr>
<td>dsc</td>
<td>662020</td>
</tr>
<tr>
<td>hlt</td>
<td>630000</td>
</tr>
<tr>
<td>iad</td>
<td>640232</td>
</tr>
<tr>
<td>ial</td>
<td>740222</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ix1</td>
<td>600303</td>
</tr>
<tr>
<td>lac</td>
<td>700022</td>
</tr>
<tr>
<td>lad</td>
<td>600032</td>
</tr>
<tr>
<td>lar</td>
<td>700622</td>
</tr>
<tr>
<td>lax</td>
<td>360000</td>
</tr>
<tr>
<td>laz</td>
<td>700072</td>
</tr>
<tr>
<td>lcc</td>
<td>700062</td>
</tr>
<tr>
<td>lcd</td>
<td>600072</td>
</tr>
<tr>
<td>lda</td>
<td>340000</td>
</tr>
<tr>
<td>ldx</td>
<td>240000</td>
</tr>
<tr>
<td>lir</td>
<td>300000</td>
</tr>
<tr>
<td>llx</td>
<td>320000</td>
</tr>
<tr>
<td>lpd</td>
<td>600022</td>
</tr>
<tr>
<td>lro</td>
<td>600200</td>
</tr>
<tr>
<td>lxr</td>
<td>600003</td>
</tr>
<tr>
<td>opr</td>
<td>600000</td>
</tr>
<tr>
<td>ora</td>
<td>740025</td>
</tr>
<tr>
<td>orl</td>
<td>640205</td>
</tr>
<tr>
<td>oro</td>
<td>740205</td>
</tr>
<tr>
<td>p61</td>
<td>766020</td>
</tr>
<tr>
<td>p6h</td>
<td>626600</td>
</tr>
<tr>
<td>p6o</td>
<td>666020</td>
</tr>
<tr>
<td>p6e</td>
<td>726000</td>
</tr>
<tr>
<td>p7n</td>
<td>627600</td>
</tr>
<tr>
<td>p7o</td>
<td>667020</td>
</tr>
<tr>
<td>pen</td>
<td>603000</td>
</tr>
<tr>
<td>pnc</td>
<td>664060</td>
</tr>
<tr>
<td>pno</td>
<td>664020</td>
</tr>
<tr>
<td>pnt</td>
<td>624600</td>
</tr>
<tr>
<td>prt</td>
<td>624000</td>
</tr>
<tr>
<td>r1c</td>
<td>721000</td>
</tr>
<tr>
<td>r1r</td>
<td>721600</td>
</tr>
<tr>
<td>r3c</td>
<td>723000</td>
</tr>
<tr>
<td>rax</td>
<td>640203</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rds</td>
<td>604004</td>
</tr>
<tr>
<td>rew</td>
<td>604010</td>
</tr>
<tr>
<td>rpf</td>
<td>706020</td>
</tr>
<tr>
<td>rtb</td>
<td>604004</td>
</tr>
<tr>
<td>rtd</td>
<td>604024</td>
</tr>
<tr>
<td>rxa</td>
<td>700322</td>
</tr>
<tr>
<td>shr</td>
<td>600400</td>
</tr>
<tr>
<td>slr</td>
<td>100000</td>
</tr>
<tr>
<td>sx</td>
<td>120000</td>
</tr>
<tr>
<td>spf</td>
<td>647000</td>
</tr>
<tr>
<td>stc</td>
<td>0</td>
</tr>
<tr>
<td>stx</td>
<td>020000</td>
</tr>
<tr>
<td>stz</td>
<td>140000</td>
</tr>
<tr>
<td>sxa</td>
<td>040000</td>
</tr>
<tr>
<td>tac</td>
<td>701000</td>
</tr>
<tr>
<td>tbr</td>
<td>702020</td>
</tr>
<tr>
<td>tix</td>
<td>450000</td>
</tr>
<tr>
<td>tlv</td>
<td>540000</td>
</tr>
<tr>
<td>tpl</td>
<td>560000</td>
</tr>
<tr>
<td>tra</td>
<td>500000</td>
</tr>
<tr>
<td>trn</td>
<td>400000</td>
</tr>
<tr>
<td>trx</td>
<td>520000</td>
</tr>
<tr>
<td>tsx</td>
<td>440000</td>
</tr>
<tr>
<td>typ</td>
<td>625000</td>
</tr>
<tr>
<td>tze</td>
<td>420000</td>
</tr>
<tr>
<td>wrs</td>
<td>604014</td>
</tr>
<tr>
<td>wtb</td>
<td>604014</td>
</tr>
<tr>
<td>2td</td>
<td>604034</td>
</tr>
<tr>
<td>xac</td>
<td>700120</td>
</tr>
<tr>
<td>xad</td>
<td>600130</td>
</tr>
<tr>
<td>xal</td>
<td>700110</td>
</tr>
<tr>
<td>xcd</td>
<td>600170</td>
</tr>
<tr>
<td>xlr</td>
<td>600300</td>
</tr>
<tr>
<td>xro</td>
<td>600001</td>
</tr>
</tbody>
</table>
Part 2--Pseudo-Instructions

character  Inserts numerical value of a flexo character.
constants  Denotes location of stored constants words.
decimal   Interpret integers as decimal numbers.
define    Define macro-instructions.
dimension Allocates space for arrays.
derip      Ends indefinite repeat.
entry     In relocatable programs, puts symbol definitions into program card for use by BRS loader.
equals    Defines symbol as operationally equivalent to another symbol.
exit      In relocatable programs, names subroutines to be called by entering names in transfer vector.
flexo     Inserts numerical value for three flexo characters.
frontloading Calls for front input routine.
irp       Indefinite repeat. Analyses macro-instruction argument as series of subarguments.
noinput   Suppresses input routine, leaves "readin" status.
nul       No-operation, ignored.
octal     Interpret integers as octal numbers.
opsyn     Defines symbol; same as equals but effective on Pass 1 only.
print     Generates symbolic location printout and prints comment during assembly.
printx    Prints comment during assembly.
readin    Punch in readin mode format.
relocatable Punch in relocatable format.
repeat    Repeats character string.
start     Denotes end of program and specifies (in absolute program) starting address.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminate</td>
<td>Ends macro definition.</td>
</tr>
<tr>
<td>text</td>
<td>Inserts words of flexo characters.</td>
</tr>
<tr>
<td>variables</td>
<td>Reserves space for arrays and variables.</td>
</tr>
<tr>
<td>word</td>
<td>Punches word on object program tape.</td>
</tr>
<tr>
<td>0if</td>
<td>Has value 0 if condition following is true, 1 otherwise.</td>
</tr>
<tr>
<td>1if</td>
<td>Has value 1 if condition following is true, 0 otherwise.</td>
</tr>
</tbody>
</table>