This memorandum describes an assembler which has been in use on the TX-0 computer at MIT for a year, and has recently been translated for use on the PDP-1. Since the MIDAS language includes most of MACRO, it is hoped that MACRO users will easily be able to switch over to this more powerful assembler.

The MACRO language had been used on the TX-0 for some three years previous to the writing of MIDAS. Hence, MIDAS incorporates most of the features which have been requested by users of MACRO, such as more flexible macro instructions, six character symbols and relocation.

The original MIDAS Assembler was written for MIT primarily by Robert A. Saunders. The PDP-1 translation was done by R. Saunders, now of III; A. Kotok, DEC; W. F. Mann, BBN; D. Gross, MIT; and S. D. Piner, DEC.

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 enter 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 characters (letters, numerals, and/or periods) containing at least one letter. The first six characters of a symbol are used to indentify it if it is more than six characters long.

Syllables may be combined with the following operators:

+	or space means addition, modulo $2^{18}-1$ (ones complement)
-	means addition of the ones complement
V	means logical union (inclusive or)
∧	means logical intersection (logical and)
~	means logical disjunction (exclusive or)
×	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 V, ∧, ~, and × 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 operatior with any other operator. In a string of consecutive additive operations, 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".".)

2	2
2+3	5
2-3	777776
2×3	6
2∨3	3
2∧3	2
2~3	1
-2~3	777776
--1	777776
-+1	1
7-2∨3	4
add 40	400040
cla∨cma	761200

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

More About Symbols. Pseudo-Instructions

MIDAS classifies symbols according to the manner of their definition. The initial vocabulary consists of symbols for the more commonly used PDP-1 instructions, and also a class of symbols called pseudo-instructions, which represent directions to MIDAS on how to proceed with the assembly. Some examples of pseudo-instructions are:

P-I	Action
octal	All integers following (unless specifically denoted as decimal) are interpreted as octal numbers until next appearance of pseudo-instruction decimal.
decimal	All integers following are interpreted as decimal numbers until next appearance of pseudo-instruction octal.
start	Denotes the end of the program.

Additional pseudo-instruction will be discussed at opportune places. A complete list is given in Appendix 1.

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 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 a 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

   clc=claVcma↲

   defines *clc* as 761200, which, as an operate instruction, would clear and complement the AC.
3. As variables. The appearance of overbar within any legal,

3. (Cont'd)

undefined symbol, at any appearance of that symbol, defines that symbol as a <u>variable</u>. For each such symbol defined, one register is allocated in a region of storage reserved by the next appearance of the pseudo-instruction <u>variables</u>. The initial contents of these registers is undefined. This feature facilitates the reserving of temporary storage locations. Example:

```
                  .
                  .
                law i 100
                dac temp
                  .
                  .
                isp temp
                jmp loop
                  .
                  .
                variables
```

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

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

```
equals anysym, defsym
        or
 opsyn anysym, defsym
```

where the symbol <u>anysym</u> is made logically equivalent to <u>defsym</u> if the latter is defined. Previously defined symbols are redefined. <u>Equals</u> and <u>opsyn</u> differ in one

5. respect: opsyn is effective on 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.

The Location Counter

The MIDAS location counter records the assigned location for each word in the object program. It is set to 4 at the beginning of each pass, and counts upward modulo memory size. The location counter may be set to any value by writing:

expr/

where expr is any symbolic expression. This sets the location counter to the value of expr modulo 2^{12}. If expr contains an undefined symbol, on Pass 1 the location becomes indefinite, and the definition of address tags in inhibited until the location again becomes definite by means of a defined location assignment. On Pass 2, an undefined symbol of a defined location assignment.
On Pass 2, an undefined symbol

will result in an error message (usl). 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:

```
sza i
jmp .+3          clf 1
law 1            szf i 1
dac indic        jmp .-1
```

The first example places 1 in register indic if the AC contains any number other than zero, but zero in the AC causes the program to skip this sequence. The second example waits for flag 1 to be set by the typewriter. The third instruction is read "jump point minus one."

COMMENTS

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

```
lac (add 20)
```

The word enclosed in parentheses is stored in a block reserved by the next appearance of the pseudo-instruction constants. Duplicate constants are stored only once. Closing parens will be supplied automatically by MIDAS if the character following is a word terminator

<u>CONSTANTS</u> (Cont'd)

(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)-1io-(30       add a
                     .                  .
                     .                  .
                     .                  .
          constants               a,  add b-1io-c
                                  b,   20
                                  c,   30
```

The pseudo-instruction <u>constants</u> may not be used where the location is indefinite.

<u>Flexo Code Pseudo-Instructions</u>

Three pseudo-instructions are provided to facilitate handling flexowriter characters in programs. These are:

1) <u>character qc</u>, where <u>q</u> is any of the letters <u>l</u>, <u>m</u>, or <u>r</u>, which specifies whether the character <u>c</u> is to be placed in the left (bits 0-5), middle (bits 6-11.) or right (bits 12.-17.) portion of the word. The pseudo-instruction, with its argument, is treated as a single syllable.

2) <u>flexo abc</u>, where <u>a</u>, <u>b</u>, <u>c</u> are any three flexo characters, is equivalent to

   character la+character mb+character rc

3) text qArbitrary string of characters.q, where the arbitrary string of characters is stored three to a word as in flexo until the first character q is encountered again. 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.

character rf	is equivalent to 66
character mm	is equivalent to 4400
flexo thi	is equivalent to 237071
text .this.	is equivalent to 237071 220000

Macro Instructions

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

```
lac a
add b
dac c
lac d
add e
dac f
```

The sequence:

```
lac x
add y
dac 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 mode is used. The appearance of a macro-instruction name in the source program is referred to as a call. The arguments are listed immediately following 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
                  law b
                  jda wr
                  text /a/
        b,        terminate
```

The pseudo-instruction define defines the first legal symbol following it as a macro name. Next follow dummy arguments as required, 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, $\vee$, $\wedge$, ~, $\times$, upper case, lower case, tab, carriage return, equals, comma, slash, overbar, parentheses, brackets and apostrophe. Dummy arguments must be legal symbols; any previous definition of dummy argument symbol 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:

```
             law nextag
             jda wr
             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 typewriter. The argument to be printed, using this format, must not contain the characters comma, tab, carriage return or slash. Comma, tab, or carriage return would end the argument while slash 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 [and] are provided. They might be used as follows:

```
          write [ This, of course, has commas.
          It also has a carriage return],nextag
```

All characters within a pair of brackets are considered to be one argument, and this entire argument, with the brackets 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 brackets. If brackets appear within brackets, the outermost pair is deleted. If an outer bracket 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 slash denoted that the dummy symbol 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 ...a01, ...a02,...,...a09, ...a0a, etc. The created symbol generator is reset to...a01 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 the dummy arguments following a bar suppress the generation of a corresponding created symbol.

A possible problem is, 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, apostrophe, 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 apostrophe is not both preceded and followed by case shifts, only the apostrophe is deleted.

Example:

```
        define  type x464pq
                lio (charac r'x464pq
                tyo
                terminate
                type f    gives: lio (charac rf
                                 tyo
```

How may one cause a created symbol to define a variable? The solution is to place an overbar over the first character of the dummy argument. Note that the overbar may not appear in the middle of the dummy argument.

Example:

```
        define  macro  /abcd
                dac ābcd
                jsp subr
                lac abcd
        terminate
```

The variables would then be of the form $\bar{.}$..a01, $\bar{.}$..a02, 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 defintions 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
              law d                 /called first time
              jda wr
              text /c/
d,            terminate write

              write [a]             /calls new definition
              tra zzxgwq

wr,           0                     /text printing subr
              dap wrzx
lpkh,         lio i wr
              ril 6s
              tyo
              ril 6s
              tyo
              ril 6s
              tyo
              idx wr
              sas wrzx
              jmp lpkh
              jmp i wrzx
wrxz,         0
zzxgwq,       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 supplies 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 (_mnd_) 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 brackets (see example below), contain sub-arguments. A pseudo-instruction _irp_ (indefinite repeat) permits the analysis of such an argument. The pseudo-instruction _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 _irp_ until the next matching _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:

```
          define    sum a,b,c
                    lac a
                    irp b
                    add b
                    endirp
                    dac c
                    terminate

                    sum j,[k,l,m],n
          gives:
                    lac j
                    add k
                    add l
                    add m
                    dac n
```

It is quite permissible to have _irp's_ within an _irp,_ analyzing either the same of different arguments. The pseudo-instructions _irp_ and _endirp_ are defined only within a macro definition. If an _irp_ analyzes a null string, the characters in the range of the _irp_ will not be inserted in the macro expansion.

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 "somethine 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 (*sce*).

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*, equal 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 word 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-instructions *1if* and *0if* 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:

```
          repeat 0if vp x+1, macro arg1, arg2↵
          a=1if vzx∧600000→|
      dac p+1if vp-s/×2
```

The value of *1if* is one if the condition tested for is true, and zero otherwise; while the value of *0if* is zero if the condition tested for is true, and one otherwise. There are at present three qualifiers with three 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.
p:	Test is true on Pass 2, false on Pass 1.

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 dac p if s is positive, and dac p+2 if s is negative. It could also be written as:

```
dac p+2×01f vps
```

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 start pseudo-instruction. The title is the first string of characters other than carriage return or stop code and is terminated by a carriage return. Carriage returns 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 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. Start must be preceded by a tab or carriage return. There must be a stop code after the carriage return after start.

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) a center dot. 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 PDP-1 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 causes the input routine to transfer at once to the address specified. The argument of start must have the value of the address to which control is to be transferred.

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 7751-7777.

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 any where in memory. An explanation of this feature will be found in subsequent issues of this memorandum.

Information on Relocatable Programming will be supplied in the next edition.

Format

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.

<u>Format</u> (Cont'd)

2) Backspace, ⊃, <, >, ", ↑, →, ?, $\underline{\cdot}$, _, |, red, black, and unused characters of the flexo code are illegal except in arguments of flexo code pseudo-instructions, titles and comments.

3) Stop codes are ignored except in arguments of flexo code pseudo-instructions. Apostrophes and brackets 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) 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.

3) Forget that you ever learned to count higher than three; let MIDAS count for you. Do not say <u>dac .+6;</u>use an address tag. This will save grief when corrections are required.

Format (Cont'd)

5) Organize the program by pages, separating each page of flexo tape with a stop code and some tape feed. Make page boundaries coincide with logical division 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. Set the test address to 4 and the TW to 0. Load the first source tape into the reader and press continue. MIDAS will read the tape in sections of about one page each, and will stop shortly after reading the stop code at the end of the tape. To process additional tapes after the first, press Start. Now begin Pass 2 by loading the first tape and pressing Continue. For additional tapes, press Start. At the end of Pass 2, press Continue 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:

Condition	AC	IO	Action on Continue	Action on Start
MIDAS or symbol punch read in	0	-0	Begin Pass 1	Begin Pass 2
End of tape, Pass 1	0	0	Begin Pass 2	Continue Pass 1
End of tape, Pass 2	0	0	Punch start block	Continue Pass 2
After start block	0	-0	Restore, begin	Begin Pass 1

Table (Cont'd)

Condition	AC	IO	Action on Continue	Action on Start
Error stop	-0	-0	Continue, suppress punching	Continue Pass

The normal sequence of operations above can be modified by use of the <u>TW</u>. Whenever <u>Start</u> is pressed, bit 0 of the <u>TW</u> is examined. If it is zero, the normal sequence is followed; if it is 1, the next 6 bits of the <u>TW</u> are examined. These control:

Bit 1	Pass 1 if 0, pass 2 if 1.
2	Begin pass if 0, continue pass if 1.
3	If 1, punch if pass 2; if 0, do not punch.
4	If 1, punch input routine if punching; if 0, no input.
5	If 1, punch title if punching; if 0, no title.
6	If 1, restore symbol table to initial symbols and pseudo-instructions.

It is sometimes useful to type in on-line short programs symbol definitions, and the like. This may be done by having sense switch 5 up when <u>Start</u> or <u>Continue</u> is pressed. Instead of reading tape, MIDAS will listen to the typewriter until either a) the buffer is full, in which case the characters will be processed, and control returned to the typewriter, or b) Sense switch 5 is turned off. If you make a typing error, set the test word to 0, press <u>Start</u>, and start typing this buffer load over again.

<u>Error Stops</u>

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 <u>r</u> 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 <u>Continue</u> or <u>Start</u> (with <u>TW</u> 0=0) is

Error Stops (Cont'd)

pressed; the two are equivalent except Continue will discontinue punching on Pass 2 if it was in progress. Turning up TW 17 is equivalent to pressing Continue after an error stop. In either case, if bit 3 of the TW (the punch bit) is on, punching will continue.

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 constant.
s:	In the argument of start.
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.
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 begin defined. First name used.
mdt	Multiply defined tag. Original definition retained.
mdv	Multiply defined variable. A symbol containing an overbar 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 assignment is ignored.
sce	Storage capacity exceeded. Assembly cannot continue.
tmc	Too many constants. The pseudo-instruction constants used too many times in one program, on too many constants words used.
tmp	Too many parameters: the storage reserved for macro instruction arguments has been exceeded.
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 location from that found on Pass 1, meaning all the constants syllables have been assigned the wrong value. Assembly cannot continue.
vld	Variables location disagrees. The pseudo-instruction variables has appeared on Pass 2 in a different location from that found on Pass1. The condition is ignored.
iae	Send the error message and a copy and listing of the source program to the DEC Programming Group so that the trouble may be found.

Troubleshooting

The checking features built into MIDAS will detect simple errors like forgotten tags very simply. Attempting to debug complex macro definitions fromerror messages and binary output is a much more difficult task. Special aids have been provided to simplify this.

1. The pseudo-instructions _print_ and _printx_ take an argument exactly like text, which MIDAS will print out on-line during the assembly process. _Printx_ prints just the argument, while _print_ precedes this with the first three columns of an error listing (with the "error" code _pnt_) and follows it with a carriage return. The argument of _print_ or _printx_ may contain dummy symbols if used in a macro definition.
2. Bit 16 of _TW_ when on, causes MIDAS to print out on-line every character it processes, including all macro expansions. This permits the programmer to let MIDAS do the bokkeeping when testing a complicated macro.

The Symbol Package

A record of symbol definitions may be printed or punched out by use of MIDAS Symbol Package. The MIDAS Symbol Package looks at the Sense Switches to determine its mode of operation.

SS	Function
1	Symbol Punch
2	Alphabetically ordered symbol printout
3	Numerically ordered symbol printout
4	Restore MIDAS to original symbol table

The Sense Switches should be set before pressing Read-In, but if it is desired to eliminate any of the above functions before they complete, just turn the appropriate switch off. If SS1 is up the symbol punch 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, _TW_ 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.

APPENDIX I--MIDAS INITIAL VOCABULARY

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.
endirp	Ends indefinite repeat.
equals	Defines symbol as operationally equivalent to another symbol.
flexo	Inserts numerical value for three flexo characters.
irp	Indefinite repeat. Analyses macro-instruction argument as series of subarguments.
noinput	Suppresses input routine, leaves "readin" status.
null	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.

terminate	Ends macro definition.
text	Inserts words of flexo characters.
variables	Reserves space for arrays and variables.
word	Punches word on object program tape.
0if	Has value of 0 if condition following is true, 1 otherwise.
1if	Has value 1 if condition following is true, 0 otherwise.

APPENDIX II

SOME MACRO-INSTRUCTION EXAMPLES

Following are some examples illustrating some more complex uses of macro-instructions. All of these examples use so-called "information carrying macros." Basically, an information carrying macro is a name assigned to a character string which has provision for using or modifying the string. Three different methods are used for retrieving the information in the following examples.

The first two examples illustrate a method of locating coding at a remote place in the program. It is sometimes convenient, in the middle of a program, to specify flexo text, subroutines, or other material to be inserted at an out-of-the way place. The macro name remote, followed by arbitrary material as an argument, saves up such material for all uses of remote until the macro-instruction here is used, which unloads all the stored information into the program at that point.

In the first example, listname is the information carrying macro. Each call of remote calls in cons to concatenate the new information onto the end of the old. The key to understanding the example is in the definition and use of listname. In order to feed the information in listname into some macro which can make use of it, listname must be called (expanded) and the characters therein fed to the macro to make use of them. This is done by feeding the name of the macro to use the information to listname as its argument. The exapnsion of listname generates the name of the user, followed by two arguments: the name listname itself, followed by the information characters in listname. Thus the user macro can be one which deals with several different information carriers, each of which carries its own label. The point is that in order to generate a function of the information in an i.c.m., first take i.c.m. name of the function name. The i.c.m. flips the function name in front of the information as it expands.

Exercise: Generate the expansion of the following code:

```
remote alfa
remote [add t
        dac t]
here
```

The second example has remote as the i.c.m. The definition of remote is such that remote effectively redefines itself, adding on to its definition anything fed it as an argument. The here macro redefines listname so that when remote next calls it, it unloads itself into the program instead of into a new definition of remote. The definitions as written here are not self-resetting: the appearance of here does not leave either remote or listname in condition to be used again.

Exercise: Define a macro setup which establishes the correct initial definitions of remote and listname when it is called. Insert calls of setup in appropriate places so that the definitions of remote and listname are properly initialized, and are reset by use of here.

The purpose of the third example is to allow indiscriminate use of the pseudo-instructions octal and decimal in macro definitions without disturbing the current radix outside of macro calls. To this end, the system definitions of octal and decimal are saved in the name roctal (real octal) and rdecml (real decimal). Then octal and decimal are defined as macros which, in addition to setting the current radix, also append the radix to a list of radices called list. To restore the previous radix, the macro oldradix peels the top entry off the list and discards it, then sets the current radix to the top of the entry of the remaining list. The list, after use of decimal and octal would look in part like this:

```
define append newrdx
list newrdx, [roctal], [rdecml, [roctal, [error]]]
terminate
```

The method used for manipulating the list is similar to that of example 2. Note how the third argument of list is added to and deleted from.

Exercises: Determine the definition of oldradix corresponding to the above definition of append. Expand decimal and determine its effect on the list.

The last example illustrates the use of irp, 0if, and 1if. The macro deciprt prints out on-line at assembly time the numerical value of its argument in English words. Zero suppression, sign, and numbers ending in "teen" are all handled correctly. The i.c.m. info contains the text to be printed out, and is handled similarly to listname in the first example. The sequence info redefine appears so often in the original that the macro in has been defined as a shorthand for it. The conversion to decimal is handled by the usual method of depletion of powers of ten. Zero suppression is handled by the indicator sup.

```
/remote macros-method 1, S. R. Russell

define remote a
         listname [cons [a], ]

termin

define listname user
         user listname,

terminate

define cons i2, name, i1/ user
         define name user

         user name, [i1

i2]
terminate name
terminate cons

define here
         listname 2ndarg
         define listname user
         user listname.

terminate listname
terminate here

define 2ndarg a,b
         b

termin

start

/remote macro-method 2, A. Kotok

define remote a
         listname a

termin

define listname i1/i2
         define remote i2
         listname [i1
i2]
terminate remote
terminate listname

define here
         define listname info

info
terminate listname

remote
terminate here

start
```

```
/octal decimal pushdown, S. D. Piner

opsyn roctal, octal
opsyn rdecml, decimal

define octal
         append roctal

termin

define decimal
         append rdecl

termin

define error
         print /Too many oldradix pullups./
         list roctal, error

termin

define list radix, prevrdx, rdxlist
         define append newrdx
         list newrdx, [radix], [prevrdx, [rdxlist]]

newrdx
terminate append
define oldradix
         list prevrdx, rdxlist
         prevrdx
terminate oldradix
terminate list

list roctal, error

start
```

decimal print macros, D. A. Gross

```
define deciprt number
           z=number
           repeat 0if vp z,in minus deci2-z
           repeat 1if vp z/-1if vz z,deci2-z
           repeat 1if vz z/∧1if vz z∧1, in zero
           repeat 1if vz z/∧1if vz zV1, in minus zero
           info write
           redefine
terminate

define deci2 a
           x=a
           sup=0
           deplete 100000.
           teen=0
           integer
           place hundred
           deplete 10000.
           intergy
           deplete 1000.
           integer
           place thousand
           deplete 100.
           teen=0
           sup=0
           integer
           place hundred
           deplete 10.
           intergy
           deplete 1
           integer
terminate

define redefine y
           define info user, data
           user y data
           terminate info
terminate redefine

redefine

define in a
           info redefine, a

terminate

define arg a,b
           sup=1
           repeat teen, in a
           repeat 1-teen, in b
terminate

define place a
           repeat sup, in a
terminate
```

```
define deplete a
           y=0
           repeat 9, repeat 1if vp x-a, x=x-a  y=y+1
terminate

define integer

int1 [[arg eleven, one], [arg twelve, two],[arg thirteen, three]

[arg fourteen, four],[arg fifteen, five],[arg sixteen, six]

[arg seventeen, seven],[arg eighteen, eight],[arg nineteen, nine]]

repeat 1if vz y, repeat teen, in ten
terminate

define intergy

int1 [teen=1, in twenty, in thirty, in forty, in fifty, in sixty
in seventy, in eighty, in ninety]

repeat 0if vz y, sup=1
termin

define int1 k
           j=1
           irp k
           repeat 1if vz y-j,k
           j=j+1
           endirp
terminate

define write b
           printx /b/
terminate

start
```