MICRO

MACHINE-INDEPENDENT

MICROASSEMBLER

11 July 1980

by

Edward Fiala
Peter Deutsch
Butler Lampson

Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA. 94304

Filed on: [Maxc1]<AltoDocs>Micro.Press
Sources on: [Ivy]<DoradoSource>MicroMemo.Dm

This manual describes a machine-independent microassembly language originally developed for the Maxc1 computer and since used for the Maxc2, Dorado, and D0 computers as well as for several smaller projects.

TABLE OF CONTENTS

1. Introduction

This document describes MICRO, originally implemented in 1971 for NOVA in Algol to assemble microprograms for the Maxc1 microprocessor. It has since been reimplemented for Alto in Bcpl and is now used to assemble microprograms for Maxc1, Maxc2, Dorado, and D0. Its output format is compatible with the MIDAS loader/debugger, for which there are versions on each of these four machines.

Micro is a rather unspecialized one-pass assembler. It does not know anything specific about the target machine, but instead has a general facility for defining fields and memories, a standard string-oriented macro capability, and a rather unusual parsing algorithm which allows setting fields in memories in a natural way by defining suitable macros and neutrals with properly chosen names.

This document will be of interest primarily to someone who is going to define a new assembly language for some machine. There are a number of complications inside Micro that this person must be aware of when defining the language. However, once the language has been appropriately defined, the interface seen by someone writing programs for a target machine is natural and simple.

In other words, if you were going to write microprograms for Dorado or D0, for example, you would need to read "The Dorado Microassembler" or "The D0 Microassembler", which define languages for those machines, but would probably not require this document.

2. Assembly Procedures

To assemble microprograms on your Alto, you must obtain [Maxc]<Alto>Micro.run or [IVY]<Alto>Micro.run. In addition, you will need to get the definition file(s) for the particular microlanguage that you will be using (see other relevant documentation).

Micro flushes Bravo trailers, so you *can* use Bravo formatting in the preparation of microprograms. However, MCross, a Maxc program that produces cross-reference listings of Micro programs, does not ignore Bravo trailers, so you *may not* use any Bravo formatting features if you are going to use MCross. In addition, error messages produced during assembly have line numbers that will be more difficult to correlate with source statements if automatic Bravo line breaks occur in the source text rather than explicit carriage returns.

We recommend use of GACHA8 (i.e., a relatively small fixed pitch font) for printing hardcopy microprogram listings, and the use of GACHA10.AL for editting microprograms with Bravo. Bravo tab stops should be set at precisely 8 character interals for identical tabulation in Bravo and MCross.

The two relevant lines in USER.CM for Bravo are:

```
FONT:0 GACHA 8 GACHA 10
TABS: Standard tab width = 1795
```

You will probably want to delete the other Font lines for Bravo in USER.CM.

Suppose that you have prepared a language definition file LANG.MC and a number of source files for assembly by Micro. Then a microassembly is accomplished by the following dialog with the

Alto Executive:

```
MICRO/L LANG SRC0 SRC1 ... SRCn
```

This causes the source files "LANG.MC", "SRC0.MC", ... , "SRCn.MC" to be assembled. The binary output and symbol table at the end of assembly are written onto "SRCn.MB" and "SRCn.ST", the error messages onto "SRCn.ER", and an assembly listing onto "SRCn.LS".

In other words, Micro assembles a sequence of source files with default extension ".MC" and outputs four files whose extensions are ".MB", ".ER", ".LS", and ".ST". The default name for these is the name of the *last source file* assembled. Direct output to particular files as follows:

```
MICRO SYS/L/B LANG SRC0 SRC1 ... SRCN
```

This would cause listing output to be put on "SYS.LS" and symbol table and binary output onto "SYS.ST" and "SYS.MB".

A summary of the local and global flags for Micro is as follows:

Global:	/L	produces an expanded listing of the output
	/N	suppress binary output
	/O	suppress symbol table output
	/U	convert text in all source files to upper case
Local:	/R	recover from symbol table file.
	/L	put expanded listing on named file
	/B	puts binary output and symbol table output on named file with extensions .MB and .ST, respectively. Default error listing to named file.
	/E	put error listing on named file
	/S	put symbol table on named file
	/U	convert text in named file (and any file which it INSERT's) to upper case

Local flags override global ones.

```
INSERT[file];
```

statements may be put into source files so you don't have to type as many source names on the command line. This is exactly equivalent to the text of file.mc. INSERT's may be nested to a reasonable depth. However, although INSERT saves typing it is *slower* than putting the file names on the command line because Micro uses a fast file-lookup routine to get handles on every file named in the command line in about 1 second; each INSERT adds an additional 1 second for file name lookup.

Another shortcut is to define a command file MI containing "Micro/O/U LANG" or whatever and then type @MI@ SRC0 ... SRCN", which avoids some typing.

The SETMBEXT[.ext] builtin allows the binary output file extension to be changed from .MB to something else. This declaration has to be assembled before defining any memories (else the output file will have already been opened with extension .MB). The Dorado and D0 microassemblers use this to change the extension to .DIB, as expected by the postprocessor, MicroD.

Micro creates a temporary file Micro.fixups and deletes it at the end of assembly. If you abort assembly with shift-swat, you may delete it yourself.

Micro's binary output is generated in one pass and consists of memory definitions, store directives to memories, forward and external reference fixup directives, and new or changed address symbols for each memory. The block types written on the output file are given in Appendix 3.

Micro assembles declarations at a rate of about 60 statements/second and, with typical microlanguages, assembles microinstructions at about 7 statements/second. On very large assemblies this rate slows slightly as the symbol table grows larger. The assembly time for the Maxc system microcode is about 7 minutes (~2000 72-bit microinstructions, ~500 36-bit words in other memories, ~500 definitions, and ~1400 addresses).

Comments are flushed very quickly by the prescan, so do not worry about a profusion of comments slowing assembly.

Presently, the Micro-Midas system has *no provision for relocating independently assembled source programs.* However, the Micro symbol table is dumped onto a file at the end of the assembly. Later, assembly can be continued at that point onto another binary output file, thereby reducing assembly time. For example, you can build a LANG.ST file as follows:

```
MICRO LANG
```

Then do all further assemblies as follows:

```
MICRO/OU LANG/R SYS/B SRC0 ... SRCN
```

This saves a little assembly time but still does not allow several people to independently maintain sources used in a common system.

To avoid reassembling unchanged files, one would have to partition his program into separate assemblies, each of which used absolute location-counters for the various memories. This would be difficult, probably not as good as reassembling everything. However, if this were done, Midas could link external references between the different modules at load time.

The MicroD program, used to post-process Micro assemblies for Dorado and D0, has limited provisions for relocation. Programmers using the Dorado or D0 microlanguages should read the relevant documentation.

3. Error Messages

During assembly, any error messages are output both to the terminal and to the error file. If an assembly listing is being printed, the error messages are also printed there.

As Micro churns through the source files it prints the name of each on the error file (and terminal), and when INSERT[file] statements appear it outputs "* FILE file ..." and "* RETURN to file" messages. These will pinpoint any error message to a particular source file.

Micro error messages are in one of two forms, like the following:

```
statement
218...error message

statement
TAG+39...error message
```

The first example indicates an error in a statement beginning on the 218th line of the source file. This form is used for errors that precede the first label in the source file. The second form is used afterwards, indicating an error on the 39th line after the label "TAG". Micro also prints the source statement causing the error before printing the error message.

Note that the line count measures carriage returns in the source, so if you are using Bravo formatting in the source files, you may have trouble distinguishing carriage returns from line breaks inserted by Bravo's hardcopy command.

ER is the builtin by which a Micro program outputs fatal and non-fatal errors, warnings, and other messages.

```
ER[message,stopcode,value]
```

Blanks are squeezed out of the message argument (a literal string) by the prescan so "-" "." or other printing characters should be used instead of blanks in the message.

Stopcode equal 0 is used for progress messages; 1 for fatal errors; 2 for ordinary errors; 3 for warnings. A nulstring in the stopcode defaults to 0. Assembly continues from the error except when a fatal error (stopcode=1) is evaluated. Ordinary errors are counted together with Micro builtin errors; warnings are counted separately. At the end of assembly the error and warning counts are printed on the Alto display; only when these counts are non-zero do you have to look in the .ER file for details.

ER first prints the source statement and then the message in one of the two forms given above; then, if the value argument is present, it is evaluated (e.g., it may be an IP or other arithmetic expression) and printed in octal; then, if stopcode is 1, assembly is aborted.

When the assembly is not aborted, assembly of the statement in which the error occurred will continue from the point of the error. This may result in more error messages if the assembler gets confused by an undefined symbol or some other condition. The location counter gets incremented iff at least one store is done by the statement, so a statement with an error may still generate an output word, or it may not.

A summary of Micro builtin error messages is given in Appendix 1.

4. Assembly Listings

An expanded listing is produced only when either the global or local /L option is selected. When the listing file is being produced, the information output is controlled independently for each memory by the LIST builtin.

```
LIST[memory,mode]
```

controls assembly listing output for all stores to the selected memory. The value of mode is bit-encoded as follows:

1	enable listing of stores in the memory as octal numbers; by default these are divided into 12-bit groups starting at the right-most bit of the value; the bit of value 20 and the LISTFIELDS builtin modify the form of the octal printout.
2	list stores in the memory as field assignments;
4	produce a numerically-ordered list of symbols at end of assembly;
10	produce an alphabetically-ordered list of symbols at end of assembly;
20	makes the octal printout divide stores into 16-bit groups.

The actions of these bits are or'ed. LIST may be given many times during the assembly, to enable/disable listing output for code sections with difficult bugs. The value of mode at the end of assembly determines whether or not numerically or alphabetically-ordered address lists are printed.

When a statement of the form:

```
ANAME[(TAG: mumble)];
```

is assembled, the listing output would be as follows:

```
302    (TAG)    NNNN NNNN ... NNNN        for mode 1

302    (TAG)    NNNNNN . . . NNNNNN       for mode 21

302    (TAG)    F1←3, F2←34, F3←20;       for mode 2

302    (TAG)    NNNN NNNN ... NNNN        for mode 3
                F1←3,  F2←34,  F3←20
```

Mode equals 0 disables all listing for the specified memory.

F1, F2, and F3 in the above example represent *all* the *fields* to which *explicit* assignments were made during the assembly of (mumble). Fields which have non-zero values due to the action of a DEFAULT statement for the memory are not listed, nor are preassignments listed. Also, fields filled in by forward references will be erroneously listed as containing their default value.

Error messages are printed on the line after the listing of the memory word or between memory words if no field assignments were completed in the statement.

LIST[,mode], where the memory name is null, AND's mode with the listing mode for all memories other than the target, e.g. LIST[,0] suppresses listing of all non-target memories and LIST[,3] restores.

The LISTFIELDS builtin can be used to control the assembly listing more precisely. Micro assembles

```
        LISTFIELDS[MNAME,(clauselist)];
```

as a word for memory MNAME and then notes the positions of all the 1-bits in the result. Thereafter, in the octal listing for that memory, rather than each field being precisely 12 or 16 bits wide, 1-bits in the word given to LISTFIELDS are taken as the rightmost bits of the fields. For example, if the word contains 1-bits only in positions 2, 5, and 6, the octal listing will show a 3-bit field (bits 0:2), another 3-bit field (bits 3:5), a 1-bit field (bit 6), and then the rest of the word chopped up into groups of 12 or 16 bits.

The mode argument to LIST determines whether or not the stores are printed, but LISTFIELDS controls the format of the numerical printout whenever that is turned on by the mode $=1_8$ bit.

5. Cross Reference Listings

A Tenex program called MCross will parse source files according to Micro syntax and produce cross-reference listings. Several simple files must be prepared to tailor MCross for the language file being used. These files eliminate the garbage tokens that would otherwise clutter the cross-refernce listing.

A cross-reference listing is not very useful for small microprograms but becomes increasingly valuable for large systems. Consequently, if you are maintaining a large system, you will probably wish to obtain an account on our Maxc timesharing system. Occasionally, you will dump the sources on your Tenex directory and run MCross over them.

A typical dialog with MCross is given below. The program is more-or-less self-documenting and will give you a list of its commands if you type "?".

```
@MCross
Output file:  LPT:GACHA8
Machine:      D              (selects Dorado syntax--M for Maxc, 0 for D0)
Action:       N              (read def's, no printout)
File:         LANG<cr>
Action:       CL             (read def's, produce cross ref.)
File:         SRC1<cr>
Action:       CL
File:         SRC2<cr>
. . .
Action:       P              (print operation usage statistics)
Action:       G              (print global cross reference)
Action:       E
@
```

6. Comments

Micro ignores all non-printing characters and Bravo trailers. This means that you can freely use spaces, tabs, and carriage returns to format your file for readability without in any way affecting the meaning of the statements.

Comments are handled as follows:

"*" begins a comment terminated by carriage return.

"%" begins a comment terminated by the next "%". This is used for multi-line comments.

";" terminates a statement. Note that if you omit the ";" terminating a statement, and for example, put a "*" to begin a comment, the same statement will be continued on the next line.

Micro has a now-obsolete method of producing conditional comments which is discussed here; at one time this was used for multi-statement conditional assembly, but now conditional assembly builtins discussed later are used instead.

The COMCHAR builtin provides conditional assembly of a large block of instructions by altering the interpretation of comments.

```
COMCHAR[char]
```

makes *char be a comment bracket similar to %. Micro will discard everything from an occurrence of *char through the end-of-line following the next occurrence of *char. Note that this is not quite like % because % stops discarding immediately at its matching occurrence.

You can disable this feature with

```
COMCHAR[]
```

which is Micro's initial state. As an example, suppose you want to assemble one of two code sequences depending upon whether some integer symbol X is zero. You could write the following:

```
IFE[X,0,COMCHAR[#],COMCHAR[=]];
*= here is some code to skip if X neq 0 (assemble if X eq 0)
...
*= end of X eq 0 code
...
*# here is some code to skip if X eq 0 (assemble if X neq 0)
...
*# end of X neq 0 code
COMCHAR[]; *Disable feature
```

7. Statements

After comments and non-printing characters are stripped out, the rest of the text forms STATEMENTS. There is no level of program structure superior to the statement (e.g., conditionals cannot span more than one statement) except for the COMCHAR kludge.

Statements are terminated by ";". You can have as many statements as you want on a text line, and you can spread statements over as many text lines as you want. Statements may be indefinitely long.

However, the size of Micro's statement buffer limits statements to 500-decimal characters at any one time. If this is exceeded at any time during assembly of a statement, an error message is output. Since horrendous macro expansions occur during instruction assembly, overflow is a possibility, and care is required when defining complicated macros.

The special characters in statements are:

"[" and "]"	for enclosing builtin, macro, field, memory, and address argument lists;
"(" and ")"	for causing nested evaluation;
"←"	as the final character of the token to its left;
":"	to put the address to its left into the symbol table with value equal to the current location in the current memory;
","	separates clauses or arguments;
";"	separates statements
"#"	#1, #2, etc. are the formal parameters inside macro definitions;
"01234567"	are number components (all arithmetic in octal)

All other printing characters are ordinary symbol constituents, so it is perfectly ok to have symbols containing "+", "-", "&", etc. which would be syntactically significant in other languages. Also, don't forget that *blanks, carriage returns, line feeds, and tabs are syntactically meaningless* (flushed by the prescan), so "P+Q" = "P + Q", each of which is a single symbol.

Micro handles all code generation by table lookup and minimal use of conditionals. In particular, it does not *evaluate* P+Q+1 but rather looks it up in the symbol table. Since P + Q + 1 is the same for a human, we have chosen to suppress all blanks. Other non-printing characters are suppressed so that control characters don't appear invisibly in print names.

Note that name length is limited only by the size of the statement buffer. However, avoid defining *address* symbols longer than about 13 characters because of problems you will encounter with the debugger Midas.

Statements are divided into *clauses* by commas. An indefinite number of clauses may appear in a statement.

Examples of clauses are:

```
NAME,
NAME[ARG1, ARG2, ..., ARGN],
FOO←FOO1←FOO2←P+Q+1            P+Q+1 is a "source" while FOO, FOO1, and FOO2 are
                               "destinations" or "sinks."
P ← STEMP
NAME[N1[N2[ARG]],ARG2]←FOO[X].
```

7.1 Builtins

All of the predefined operations of Micro are called builtins. With the exception of the BUILTIN and INSERT builtins, none of them have *a priori* names but instead are assigned names by the programmer. Names are assigned to builtin operations by declaration statements of the form:

```
BUILTIN[BUILTIN,1];
```

where the second argument is the intrinsic operation number and the first argument is the name by which it is referred to.

All builtins are called using this same syntax:

```
NAME[ARG1, ARG2, ..., ARG9];
```

The all-inclusive list of builtins is given in Table 1. Note that the *only* print-names assembled into Micro are BUILTIN and INSERT; i.e., the other names in Table 1 are chosen by convention.

7.2 Defining Symbols

The builtins BUILTIN, MACRO, NEUTRAL, MEMORY, FIELD, and SET are used to define symbols of different types, as discussed later. The name of a defined memory can then be used to define addresses in that memory, and addresses are also defined when labels appear in statements being assembled for storage in a memory. Once a symbol has been defined, it is an error to redefine it as any other type of symbol.

It is legal to change the value of a symbol of type integer.

Redefining a macro is legal (but Micro prints a warning message).

When an address is defined by a label, any attempt to change its value is illegal, but when defined by MEMNAME[symbol,value] it is legal to change the integer part of the value (illegal to change the memory part of the value).

Table 1: Builtins

Builtin No.	*Name*	*Discussion*
1	BUILTIN	Section 7.1
2	MACRO	Macro definition (usually the short name "M" is used), section 9
3	NEUTRAL	Neutral definition (usually the short name "N" is used), sections 7.3, 10
4	MEMORY	Memory definition, section 13
5	TARGET	Target memory declaration, section 13.1
6	DEFAULT	Default value of memory bits, section 13.2
7	FIELD	Field definition, section 11
10	PF	Field preassignments, section 11
11	SET	Integer definition and set, section 8
12	ADD	Section 8
13	IP	Integer part of an address, section 8
14	IFSE	If-string-equals conditional, section 12
15	IFA	If-field-assigned conditional, section 12
16	IFE	If-integers-equal conditional, section 12
17	IFG	If-integer-greater conditional, section 12
20	IFDEF	If-symbol-defined conditional, section 12
21	IFME	If-memory-part-of-address-equals-string conditional, section 12
22	ER	Print error message, section 3
23	LIST	Control assembly listing, section 4
24	INSERT	Insert file, section 2
25	NOT	Section 8
26	REPEAT	Repeat evaluation, section 14
27	OR	Section 8
30	XOR	Section 8
31	AND	Section 8
32	COMCHAR	Multi-statement conditionals, section 6
33	BITTABLE	Define bit table, section 16
34	GETBIT	Section 16
35	SETBIT	Section 16
36	FINDBIT	Section 16
37	MEMBT	Section 16
40	LSHIFT	Left-shift integer, section 8
41	RSHIFT	Right-shift integer, section 8
42	FVAL	Get value in field, section 11
43	SELECT	Switchon integer, section 15
44	SETPOST	Define post-evaluation macro for memory, section 13.3
45	--	Deimplemented
46	LISTFIELDS	Control assembly listing, section 4
47	SETMBEXT	Set binary output file extension, section 2
50	SUB	Section 8
52	ASMMODE	Multi-statement conditional assembly, section 17
53	TRACEMODE	Macro expansion tracing, section 18
54	WHILE	Repeat evaluation, section 14

7.3 Tokens

The rules for delimiting clauses into tokens have been carefully chosen to permit the user of Micro to write readable programs. The parsing of statements is strictly right-to-left and the following definitions are required in explanation:

An L-token terminates the token to its left.
An R-token terminates the token to its right.

Then:

(	R	group delimiter
)	L	group delimiter
[	L	builtin argument list delimiter
]		builtin argument list delimiter
,	LR	clause delimiter
:	LR	clause delimiter which takes the preceding token as an address in the current memory at the current address
←	LR	separator which is part of the symbol to its left

Any text with an R-token to its left and an L-token to its right constitutes a token called a *symbol* whose meaning is determined by looking it up in the symbol table. Text enclosed in parentheses is lexically independent of anything outside, and a parenthesized string of text is lexically equivalent to the "tail" which its evaluation produces. The following example clarifies this.

In the expression:

```
FOO5(FOO1[FOO2]FOO3[FOO4])FOO6[FOO7]
```

the order in which expansions are recognized assuming that each FOO expansion leaves behind no text is:

```
FOO1[FOO2]
FOO3[FOO4]
FOO5FOO6[FOO7]
```

7.4 Neutrals and Tails

The handling of tails, a distinguishing peculiarity of Micro, works as follows. The tail is initialized to the nulstring at the start of processing a clause. When a *neutral* symbol is recognized using the rules for delimiting tokens (previous section), it is concatenated on the left of a string called the tail thusly:

```
temp ← concatenate (symbol, tail);
if tail eq null do;
   tail ← temp;
else do;
   tail ← null;
   treat temp as a symbol;
end;
```

Parentheses push down the current tail and start a new null one. When the text inside is completely processed, its tail (null or neutral) is treated as though it were a string which had appeared without parentheses.

The use of neutral tails permits complicated machines like Maxc and Dorado to be described by a relatively small number of macros and neutrals. The following example shows how this works.

Maxc has about 30 bus sources and 30 bus destinations, but not all combinations of source and destination are legal (a slow source may not feed a slow destination). An example using the bus is:

```
MDR←X
```

X is a macro that expands to a store into the bus source field of the microinstruction and leaves behind the neutral symbol B. MDR←, the next token recognized, is a macro that expands into a store into the bus destination field and leaves behind the neutral symbol B←. B←B is the next token recognized. Since the connection of a fast bus source to a fast bus destination is legal, B←B has also been entered into the symbol table as a macro equivalent to the neutral symbol B.

If B← could not have been legally connected to B, then the B←B macro would not have been defined, and Micro would have output an error like "B←B undefined" when assembling the statement.

Thus the number of symbols which must be defined for describing bus sources and destinations is roughly 1/source plus 1/destination plus a small number of macros to describe legal connections of a class of sources to a class of destinations. Each class of objects is represented by a neutral symbol.

In other words, the connection concept, which neutral tails implement, decouples sources and destinations inside the language definition file. In conjunction with the peculiar handling of "←", this permits a natural assembly language to be defined in which the programmer thinks of sources flowing over buses to destinations. It is impossible to create a natural language of this type with an ordinary macroassembler.

Here is a more complicated example:

```
        STEMP←MDR←(RTEMP←P) U (X)
```

In this example (from Maxcl), there is an interior routing of data from P (a register) to RTEMP (an address in the RM memory); this routing moves data from P through the ALU and into RTEMP. The ALU data is also routed onto B (a bus) where it is or'ed with data from X (a register). Then the bus data is written into MDR (a register) and into STEMP (an address in the SM memory). A crude outline of the way this is assembled is as follows:

P is a macro that stores the P control in the ALUF field of the microinstruction and leaves the neutral ALU;

RTEMP← is recognized as an RM destination (details later); its address is stored in the RA field leaving the neutral RB←;

RB←ALU is a (connection) macro, leaving the neutral ALU behind;

X is a macro that stores the code for B←X into the BS field of the microinstruction leaving the neutral B;

ALUUB is a (connection) macro that stores the code for B←ALU into the F1 field and leaves the neutral B;

MDR← is a macro that stores the code for MDR←B into the BD field leaving the neutral B←;

B←B is a (connection) macro leaving the neutral B;

STEMP← is recognized as an SM destination (details later); its address is stored in the SA field leaving the neutral SB←;

SB←B is a (connection) macro that stores the code for loading SM into the F2 field leaving the neutral B;

B is the final tail which is thrown away.

This example is as complicated as any we have used in real assemblers thus far. The construction of "(..) U (..)" to represent merging different sources on a bus is used systematically throughout the Maxc microlanguage; sources can be given in arbitary order so, in the above example, (X) U (RTEMP←P) would also assemble. All of these factors contribute to an easily readable, easily rememberable assembly language.

In the above example, the assembler also successfully concealed some complicated alternate encoding issues from the programmer. B←ALU could have been encoded in either the BS or F1 fields; the assembler picked F1 since BS was needed for B←X. SB←B could have been encoded in either BD, F1, or F2; the assembler picked F2 because BD and F1 had already been used. These are some of the issues that the designer of a microlanguage must consider.

7.5 Clause Evaluation

When a clause is broken into *top level* tokens, the possible resulting symbol types and actions are given by the table below:

Table 2: Top Level Evaluation

Symbol type	*Action*
undefined	See section 7.7
integer	Error message and abort clause expansion
address[clauselist]	Carry out a store of the word assembled by the clauselist at the location and memory of the address, and then increment the integer part of the address symbol.
address SYM	Replace by sourcemacro[SYM] (section 13)
address SYM←	Replace by sinkmacro[SYM] (sections 7.7, 13)
unbound address	Error message
MNAME[SYM,integer]	Create an address symbol "SYM" in memory MNAME with value "integer"
FNAME[address]	Store IP[address] in field FNAME (section 11)
FNAME[integer]	Store integer in field FNAME
FNAME[undefined]	Generate forward reference for eventual field assignment at end of assembly or by MIDAS.
macro [args]	Expand it (section 9)
macro	Expand it
neutral	See sections 7.4, 10
neutral [args]	Error message
builtin [args]	Call the builtin function (Table 1) with arguments handled as discussed in section 7.6

Ultimately, the original clause must reduce through macro and neutral expansions to a series of field assignments, preassignments, and builtin calls with a neutral symbol in the "tail." The neutral symbol is then thrown away and the next clause is evaluated.

7.6 Treatment of Arguments

Many symbol types may be followed by argument lists. The only difference among these is that fields, memories, addresses, and most builtins must be followed by an exact number of arguments. Macros, on the other hand, may have surplus arguments (ignored) or deficient arguments (nulstrings supplied). Conditionals may omit arguments (nulstrings supplied).

The nulstring argument is special in the following sense. If it appears where an integer result is wanted, it is equivalent to the value 0 (except for the AND builtin, where it is equivalent to 177777); if it appears where a string is wanted, it is the nulstring; and, if it is looked up, it is undefined. Micro does not allow the programmer to define the nulstring as a symbol.

Each builtin may choose one of three basic ways to receive its arguments: *quoted*, *looked up in the symbol table*, or *evaluated*. Some languages have a step short of evaluation which might be called "macro expansion", but Micro does not make any distinction between macro expansion and complete evaluation of an argument. However, if a string of the form

```
NAME[arguments]:
```

occurs in a clause being evaluated, NAME[arguments] is expanded until a string is left without brackets or parentheses, and then this string is the one affected by the ":". However,

```
IFDEF[NAME[arguments], ...]
```

which looks up its first argument, will look up the entire string including the brackets. This is a limitation of Micro which may someday be repaired. It prevents symbol names from being generated in some situations.

The exact meaning of "look up" and "evaluate" changes with the builtin. Those builtins which "lookup" an argument generally do so for a symbol type check or to decide what action to carry out based upon the symbol type. There is no way for macro definitions to get at symbol types. Only builtins can do this. This is an unfortunate limitation of Micro.

Argument evaluation is slightly different from clause evaluation. For example, evaluating the argument for the field assignment FNAME[VALUE] takes place as follows: evaluate the tokens in the argument right-to-left expanding all macros and neutrals, looking for one of the following:

1) *Address:* Use its integer part to complete the field assignments discussed in section 11.

2) *Unbound address:* Generate a forward reference.

3) *Undefined symbol:* Create an unbound address and generate a forward reference.

4) *Integer:* Complete the assignment as discussed in section 11.

If the argument is the nulstring, put the integer 0 into the field. If the argument is a neutral symbol, if any text is left when the address, integer, or undefined symbol is found, generate an error.

Note that a neutral symbol results in no error for clause evaluation, but an error for a field assignment while an integer results in an error in a clause but no error in an assignment. Other builtins which evaluate their arguments may have different requirements.

For example, the integer builtin ADD (see section 8) accepts only integer arguments. Address [clauselist] evaluates the clauselist exactly as if it had occurred at the top level. In all cases, if part of the argument being evaluated is in parentheses, that part is evaluated exactly as if it had occurred at the top level.

7.7 Undefined Symbols

The print-name of a symbol is a character string by which the symbol can be referred to in the source. However, when the lexical scan finds a string S of characters which is a symbol token (delimited by L or R-tokens), it looks for a symbol with print-name S. If no such symbol exists, an error is indicated except in the following cases:

7.7.1 Destination Addresses

S ends with ←. In this case the ← is stripped off and the resulting string S' is looked up. If S' is an address in memory MEM, S is replaced by MEMSINK[S'] as discussed in section 11.

7.7.2 Octal Numbers

S consists entirely of octal characters with an optional leading "-" sign. In this case it is treated like a symbol of type integer whose value is the octal number. Note that integers may not be larger than 16 bits. Micro does not allow an integer string to be entered into the symbol table, which would usurp the natural use of that integer.

7.7.3 Literals

S starts with an octal character or with a "-" followed by an octal character. In this case the "-" (if any) is stripped off and the rest is split into a head OCT and a tail SYM such that OCT consists entirely of octal characters and SYM does not start with an octal character. Then the macro SYM or -SYM is called as described below.

The first argument of SYM is the four right-most octal characters. The second argument is the next four octal characters, and so on until the octal characters are used up. For example,

```
37436521000V and
-1234567V
```

are replaced by

```
V[1000,3652,374]    and
-V[4567,123].
```

The awkwardness of the 16-bit limitation for integers is clearly pointed out by this kludge. Clearly V[37436521000] would have been much easier to work with and would have been possible if the integer size was greater than or equal to the memory size. Also, going from a three-integer 36-bit result back to a text string is made impractical by the integer size limit.

8. Integers

Micro permits use of integer variables constrained to 16 bits.

```
SET[NAME,VALUE]
```

looks up its first argument and evaluates its second with the following results:

Type of Name	*Type of Value*	*Action*
Undefined	Integer	Enter NAME in the symbol table with type integer and value VALUE.
Integer	Integer	Change the value of NAME to VALUE.

All other combinations are errors.

The following builtins accept integers as arguments and produce an integer as value:

ADD[i0, i1, ... , i7]	Sums i0 ... i7
SUB[i0, i1, ... , i7]	Subtracts the sum of i1 ... i7 from i0
NOT[i0]	1's complement of i0
OR[i0, i1, ... , i7]	Inclusive-or of i0 ... i7
XOR[i0, i1, ... , i7]	Exclusive-or of i0 ... i7
AND[i0, i1, ... , i7]	And of i0 ... i7
LSHIFT[i0, i1]	Logical left-shifts the integer i0 by i1 bits
RSHIFT[i0, i1]	Logical right-shifts the integer i0 by i1 bits

In these, omitted arguments are 0's for every operation except AND, which supplies 177777 (i.e., -1) for omitted arguments. Note that octal strings may begin with an optional "-". However, the negative of an integer-valued symbol cannot be obtained by inserting a leading "-"; -(ISYM) will not work, either.

The value of these integer operations is the unsigned octal string representing the result. Example: ADD[3, 4, 15]S is equivalent to 24S.

IP[ANAME], where ANAME must be an address, is the integer part of the address. This must be done when an address is used in an arithmetic or set expression. (It is not reasonable to automatically take the integer part of an address because of confusion between its use as a source and its use as an integer).

FVAL[FNAME], where FNAME must be a field, is the integer contents of the field FNAME in the word currently being assembled. If nothing has been stored in that field yet, then the contents are whatever value was setup by the DEFAULT statement for the current memory, or are 0, if no DEFAULT statement applies.

9. Macros

A symbol can be given a macro value by the clause

```
M[NAME, body]
```

where the body is an arbitrary balanced string of characters (i.e., parentheses and brackets match up and are nested). Occurrences of the text

```
#digit
```

in the body will be replaced by the corresponding actual parameters (counting left-to-right from 1) when the macro is called. Unsupplied arguments are nulstrings, surplus arguments are ignored, and #0 will be replaced by the number of arguments supplied.

The lexical scan of a statement is done from right to left. Whenever a symbol S is detected, it is looked up. If S turns out to be a macro, then the macro body replaces both S and the bracketed argument list immediately to the right of S, if there is one. Thus after

```
M[FOO, MUMBLE#1];
```

the text FOO[E]D; expands into MUMBLEED; note that D is not a symbol since] is not an R-token. Note that the macro body is quoted and that Micro has no provision for getting any part of it expanded at definition time.

Due to the way in which macro bodies are stored in the Micro symbol table *symbols used in the macro body should be defined before the macro is defined when feasible.* Assembly will be quicker if this rule is followed.

10. Neutrals

A symbol which has been declared neutral by a clause of the form

```
NEUTRAL[SYM]
```

is concatenated with the tail and handled as discussed in section 7.4.

11. Fields, Assignments, and Preassignments

FIELD[FNAME, leftbit, rightbit] causes a symbol of type field to be created. Leftbit and rightbit must evaluate to integers. Also, because of the Alto's 16-bit integer size, the field should not be wider than 16 bits or else some bits of the field could never be set. Finally, leftbit must be in the range [0, 255] and rightbit in the range [leftbit, min(leftbit+15, 255)].

Clauses of the form

```
FNAME[integer];
FNAME[address];
FNAME[unbound address]; or
FNAME[undefined];
```

where FNAME is a field, are used to construct memory words. A field assignment evaluates its argument in the manner discussed in section 7.6.

Field assignments also have the property that attempting more than one assignment to a field in a statement will cause an error unless the new value = old value. (When an error occurs, the value ultimately left in a field is that of the final assignment to it.) Forward references fixup the true value later.

The preassignment

```
PF[FNAME, integer]
```

does nothing if any bits of FNAME have previously been assigned. Otherwise, it is equivalent to

FNAME[integer] except that a later assignment will overrule the preassignment and cause no error. Forward references are illegal in preassignments.

The integer value stored in any field of the memory word currently being assembled may be obtained by using

```
FVAL[FNAME].
```

If the field has not yet been set, FVAL returns the default value.

13. Conditionals

There are a number of builtins which will substitute the text represented by one of their arguments if the other arguments meet some condition. These are called conditionals, summarized in Table 2.

A conditional and the argument list to its right are equivalent to the "true" string, if the specified condition is met, or the "false" string, if it is not met. Note that any number of arguments may be omitted. The true and false strings may be any *balanced* strings of characters.

Although these conditionals can be used at the top level, they are intended for use inside macro definitions, and the string compare conditional could be used sensibly only inside macro definitions.

Table 2: Conditionals

Form	**Condition**
IFE[il, i2, (true), (false)]	il = i2
IFG[il, i2, (true), (false)]	il > i2
IFDEF[sl, (true), (false)]	sl in symbol table and not unbound address
IFSE[sl, s2, (true), (false)]	sl = s2
IFA[field, (true), (false)]	any bit of field previously assigned
IFME[address, sl, (true), (false)]	memory name for address = string

Note that the text in the selected arm of a conditional is concatenated with the text to the right of the conditional before evaulation, so

```
IFE[2,1,FOO,GLOT]AB
```

will evaulate GLOTAB.

13. Memories, Addresses, and Stores

```
MEMORY[MEM, wordlength, length, sourcemacro, sinkmacro]
```

causes creation of a memory. Micro can manage a reasonable number (15) of these memories, subject to a 255-bit word-length limit and 64K-1 length limit. Once MEM has been defined, symbols can be defined as addresses in MEM and words of MEM can be initialized.

An address ANAME in MEM is created by an expression of the form:

```
MEM[ANAME, integer]
```

or by using

```
ANAME:
```

in a clauselist which is stored in MEM.

Stores into MEM are generated either by selecting an address in MEM as the target (see section 13.1) or by writing

```
ANAME[(clauselist)]
```

which stores the word assembled by the clauselist into MEM at the location of the address ANAME and then increments ANAME. Note that the memory store and incrementing the address are done iff one or more field assignments or preassignments result from the clauselist.

The value stored is generated as follows: It is initialized according to the value assembled by the DEFAULT statement (0 if there has been no DEFAULT statement). Next, the clauselist is evaluated. Then the post macro for the memory, declared by the SETPOST builtin, is evaluated. Finally, if ANAME is out-of-bounds, an error message will occur.

The sourcemacro MSRC and sinkmacro MSINK are applied when the address ANAME appears in a clauselist. If ANAME is evaluated as a token in a clauselist without a following argument list, it is replaced by the string

```
MSRC[ANAME].
```

If ANAME← appears and is undefined, it is replaced by

```
MSINK[ANAME].
```

Note, however, that forward and external references can be generated *only* in the context

```
FNAME[ANAME],
```

not when ANAME is used as a source or sink.

13.1. Target Memory

At any time TARGET[ANAME] will set the target address to ANAME which means that a statement of the form

```
X:    mumble;
```

where mumble must do *at least one* field assignment, is equivalent to

```
ANAME[(X:    mumble)];
```

Otherwise, the target has no effect. Note that the target memory is *not* preserved in the /R file and must be given again for each assembly.

13.2. Default Statement

Before assembly of a clauselist for storage into a memory MEM, the word is initialized to a value which may be overruled by the various assignments in the clauselist. Normally, the initial value is 0, but this may be changed by the statement

```
DEFAULT[MEM, (clauselist)];
```

which assembles clauselist into a value that will subsequently initialize words being assembled for MEM. Note that forward references are not permitted in the clauselist and that any of the default settings may be overruled by explicit assignments in a statement being assembled.

13.3. Post Macros

```
SETPOST[MNAME,POSTMACRO]
```

arranges things so that the macro POSTMACRO will be called just after a word has been assembled for the memory named MNAME but just before the word is output to the binary file. If POSTMACRO is null, SETPOST simply turns off this feature for the memory MNAME.

14. Repeat and While

```
REPEAT[i1,TEXT]
```

assembles TEXT i1 times. This is used primarily for initializing blocks of memory and for replicating nearly-identical instructions in diagnostics.

Since TEXT cannot include ";" stores to the target memory must be put in explicitly. In other words, the program cannot rely on the TARGET directive to insert "ILC[TEXT]" or whatever each time TEXT is repeated. Note that the statement buffer is cleared after each assembly of TEXT.

```
WHILE[i1,TEXT]
```

evaluates the expression i1, which must evaluate to an integer; so long as the result is non-zero, TEXT will be evaluated and the while will repeat.

15. Select

The SELECT builtin corresponds to the Bcpl switchon (case selection) statement. Its form is

```
SELECT[index,text0,text1, ... , textn]
```

and its effect is to replace itself with one of text0, text1, ..., textn depending on whether the value of index is 0, 1, ..., n. Note that although index is evaluated and must produce an integer result, the text arguments may be any balanced strings, just as in the comparison builtins IFE, IFG, etc. If the index does not have a value in the range 0 through n, an error results.

16. Bit Tables

Several builtins manipulate bittables. The rationale for bittables in Micro is the existence of microprocessors (such as the Alto) in which the addressing structure imposes constraints on the locations of certain instructions, and for which the assembler must therefore keep track of precisely which locations have already been used for instructions. The bittable facilities in Micro are adequate for this task in simple cases.

The builtin

```
BITTABLE[table,n]
```

makes table a bittable of size n (the bits are numbered from 0 through n-1). All the bits in the bittable are initially zero.

```
GETBIT[table,i]
```

returns the value of the i'th bit in the table, 0 or 1. Setting bits is a little more complicated.

```
SETBIT[table,i,n,delta,val]
```

sets n bits in table starting with the i'th bit and going up by increments of delta (i.e., bits i, i+delta, ..., i+(n-1)*delta) to val; however, SETBIT may be called with any number of arguments from 2 to 5, with the omitted trailing arguments defaulted as follows: n=1, delta=1, val=1.

There is a builtin similar to SETBIT for locating patterns of 0-bits (available locations) in a table:

```
FINDBIT[table,i,n,delta,hop,count]
```

starts out seeing if bits i, i+delta, ..., i+(n-1)*delta in table are all zero. If so, FINDBIT returns the initial location i. If not, it increments i by hop and tries again, until it has tried a total of count times. If the search fails, FINDBIT returns a null string. As for SETBIT, FINDBIT will supply default values for trailing arguments: n=1, delta=1, hop=1, count=177777 (infinity, i.e., until the size of the bit table is reached). The idea is that, for example, to find a pair of consecutive free locations whose last 3 address bits are 110, 111 respectively, you would use FINDBIT[table,6,2,1,10].

17. Multi-Statement Conditionals

The ASMMODE builtin is used for multi-statement conditional assemblies. ASMMODE[0] is normal and is the initial setting; in this case all statements are assembled normally. ASMMODE[1] disables normal assembly; in this case only statements beginning with the ":" character are evaluated--other statements are flushed.

The following collection of macros shows how conditional assembly of statements nested up to four levels deep can be accomplished:

```
SET[ALEV,0];     *Number of nested :IF's
SET[ASMF,1];     *1 if assembling, 0 if not assembling
SET[ASML,1];     *1 if assembling at this level, 0 if ignoring
SET[L1,0]; SET[L2,0]; SET[L3,0]; SET[G1,0]; SET[G2,0]; SET[G3,0];

M[NOIF,ER[No.:IF.preceding.:#1]];

M[IF,SELECT[ALEV,,
        SET[L1,ASML]  SET[G1,ASMF],
        SET[L2,ASML]  SET[G2,ASMF],
        SET[L3,ASML]  SET[G3,ASMF],
        ER[:IF's.nested.more.than.4.levels,1]]
    SET[ALEV,ADD[ALEV,1]]  SET[ASML,ASMF]
    IFE[ASML,1,
        IFE[#1,0,ASMMODE[1]  SET[ASMF,0],ASMMODE[0]  SET[ASMF,1]]]
];

M[ELSEIF,IFE[ALEV,0,NOIF[ELSEIF],
    IFE[ASML,1,IFE[ASMF,1,SET[ASMF,0]  SET[ASML,0]  ASMMODE[1],
        SET[ASMF,#1]  ASMMODE[IFE[ASMF,0,1,0]]]]]
];

M[ELSE,IFE[ALEV,0,NOIF[ELSE],
    IFE[ASML,1,IFE[ASMF,1,SET[ASMF,0]  SET[ASML,0]  ASMMODE[1],
        ASMMODE[0]  SET[ASMF,1]]]]
];

M[ENDIF,SELECT[ALEV,
        NOIF[ENDIF],
        SET[ASMF,1]  SET[ASML,1],
        SET[ASML,L1]  SET[ASMF,G1],
        SET[ASML,L2]  SET[ASMF,G2],
        SET[ASML,L3]  SET[ASMF,G3]]
    SET[ALEV,SUB[ALEV,1]]
    IFE[ASMF,1,ASMMODE[0]]
];
```

Using these macros, programs can use the following statements for conditional assembly:

```
:IF[IM16K];
    ... statements for IM16K ne 0 ...
:ELSEIF[IM8K];
    ... statements for IM8K ne 0 ...
:ELSE;
```

```
    ... statements for IM16K and IM8K both 0 ...
:ENDIF;
... undonditionally assembled statements ...
```

18. Trace Mode

The TRACEMODE builtin is used to produce a trace of the assembly on the .ER file. This aims at debugging complicated macros and at performance tuning of definition files. The format of the trace output is not particularly pretty but is self-explanatory.

TRACEMODE[n,v] turns on tracing feature n if v is unequal to 0, or turns it off if v is zero. n=0 traces symbol table insertions; n=1 traces all applications of the form name[args].

Appendix 1. Micro Error Messages

Micro error messages are enumerated below, in which the character @ should be replaced by the printname of the token related to the error. Unless marked with a ^{1}, assembly continues from the error with no special action; errors marked with ^{1} terminate assembly.

Program Organization Errors

SOURCE FILE @ DOES NOT EXIST^1

COULD NOT OPEN FILE @ FOR 'INSERT'^{1}

STORAGE FULL^1

Storage required during the assembly is roughly proportional to the following computation:

1/2*Sum [namelength +1] for all symbols
+ 6* no. symbols
+ 1/2*Sum [length +1] of all macro definitions.

When this number is greater than the size of the buffer (approx. ? Alto words), the STORAGE FULL message results.

TOO MANY MEMORIES^1

Limit is currently 15 memories.

Declaration Errors

@ ALREADY DEFINED

The new definition will replace the old and this warning message will be printed.

MACRO @ REDEFINED

Just a warning (doesn't increment error count)

ARG NOT A MEMORY NAME

For DEFAULT, which requires an argument to be of type memory.

UNDEFINED SYMBOL @ IN 'DEFAULT'^{1}

BAD PARAMETERS FOR 'F'

A field may not be larger than 16 bits nor a memory wider than 256 bits, so rightbit > 255 or rightbit-leftbit > 16 are field definition errors.

MEMORY @ ALREADY USED^1

ILLEGAL WIDTH OR SIZE FOR 'MEMORY'^{1}

Limits are 256 bits wide and 64K-1 in size

WRONG NO. ARGS FOR '@'

Only for those builtins which must have correct number of arguments.
Macros may have too many or too few.

ILLEGAL BUILTIN NUMBER FOR '@'^{1}

Statement Assembly Errors

END OF FILE INSIDE COMMENT

Terminates comment and forges ahead

INPUT STATEMENT TOO LONG

Maximum length is 500 characters. Text to the right of the 500th character is truncated.

STATEMENT TOO LONG

During macro expansion of the input statement, the unprocessed text is never permitted to exceed 500 characters. Text to the right is truncated.

MACRO ARGUMENT STORAGE FULL
Truncates characters right-to-left up to matching '[' and proceeds.

SYMBOL @ NOT LEGAL AS TOKEN
Symbol appears without its required argument list.

@ MAY NOT BE FOLLOWED BY [ ]
Only macros, builtins, fields, addresses, and memories may have '[' to their right.

UNPAIRED) OR] IN ARGUMENT LIST

UNPAIRED)

UNPAIRED (

TOO MUCH NESTING OF ( ) AND [ ] IN CLAUSE
Limit is 8 levels

MISSING MACRO NAME OR TAG SYMBOL
No symbol to the left of a : or [.

MACRO '@' NOT DEFINED
Symbol to the left of a "[" wasn't defined

TAG @ ALREADY DEFINED

'TARGET' GIVEN AFTER FIELD SET[1]

NO TARGET FOR FIELD SET[1]

'TARGET' NOT LEGAL INSIDE A STORE[1]

@ UNDEFINED
Not including forward references. Plunges ahead with value 0 and type integer

FIELD @ DOES NOT FIT IN MEMORY @
Right bit of field > right bit of memory

VALUE @ DOES NOT FIT IN FIELD @
Left bits of value truncated before store

ARG IN FIELD STORE NOT INTEGER OR ADDRESS
Doesn't do field assignment and plunges ahead

FIELD @ ALREADY SET
The new value is stored into the field. This message will occur iff new value # old value.

ARG DOES NOT YIELD INTEGER VALUE
Assumes 0 and proceeds. Syntax OK but undefined symbol or address instead of integer.

BAD SYNTAX WHERE VALUE REQUIRED
Something complicated where a simple value expected

FIRST ARG OF 'PF' NOT FIELD
No action

FORWARD REFERENCE NOT LEGAL IN 'PF'
No action

STORE TO @ OUT OF RANGE FOR @

@ BAD FIRST ARG FOR 'SET'
Must be integer or undefined symbol. However, redefinition will take place.

INTEGER '@' TOO LARGE
Integer MOD 2**16 is used.

ARG NOT A FIELD NAME IN 'IFSET'

Appendix 2. Limitations of the Language

Micro lacks some features and possesses certain limitations discussed below:

1. It is impossible to relocate a microprogram at load time.

2. Forward and external references are permitted only on field assignments which means that the occurrence of

```
MDR←STEMP, or STEMP←MDR
```

where STEMP is an address in SM, cannot be assembled if STEMP is a forward or external reference. Forward references to symbols that are not addresses are also impossible.

3. Significant size limits:

   a. Symbol table storage is tight.

   b. Integers are limited to 16 bits.

4. It is impossible to check the memory part of an address on forward or external references. Nor is it possible for programs to get at the *type* of a symbol, at the *parameters* of a field or memory, or at the name of the target memory. The 'lookup' capability of builtins is not available through any language constructs.

5. Macros which expand to more than one statement are impossible.

6. It is impossible to pull print names apart or to construct print names except by using neutral subsymbols. In particular, it is impossible to construct constants larger than 16 bits parametrically such that, if several constants contain the same value they can be assigned the same location. This is true because one cannot generate the print name "1420000S" (a literal) either directly from an integer or indirectly from the value assembled by assignments. (Note that if integers were large enough ADD[P1, P2, ... , P7]S *would* generate the literal in S.)

7. There are a number of situations when part of an otherwise quoted argument wants to be expanded and there is no way to do this. For example,

```
IFDEF[FOO[E],(true clause),(false clause)]
```

should lead to expansion of the macro FOO[E] before checking for a defined symbol.

8. Blanks in user-defined error messages are impossible.

9. The REPEAT builtin should supply a ";" after each repetition of the text, so that the ILC[...] in REPEAT[n,(ILC[...])] can be omitted.

10. PF [field, value] was a bad choice because it makes parameterizing the values of a field impractical. For example, suppose that the function P←P1 is accomplished by setting the PS field to 50. What we would *like* to do is to define neutrals P← and P1 and then define the macro P←P1 as PS[50]. If the hardware is changed so that P←P1 is accomplished by PS[20] instead of PS[50], we

would prefer to change only the one macro P←Pl. However, there are also several instances of PF[PS,50] which have to be found and changed and this is the reason why PF[field, value] was a bad choice. Instead, a preset-clauselist operation would have been better because then no other usage than P←Pl would be needed.

To prevent some of the above limitations or to otherwise streamline or augment the language, the following changes should be considered (the ones followed by ? or ?? or ??? are not serious proposals).

1. Make integers at least 36 bits long for MAXC, and consider variable length integers. Currently, considerable inconvenience results from "making do" with 16-bit integers. Also this would make it possible to get the literal equivalent of a constant constructed from parameters, which would allow merging identically-valued constants.

2. Provide a builtin like the one for defining fields except that it takes an additional argument which is a memory name:

```
AFIELD[AFNAME, leftbit, rightbit, memory].
```

AFNAME[address] works like FNAME[integer] except that its argument must expand to an address in "memory" rather than an integer, or if its argument is undefined, a forward address reference is assumed. Forward references to FNAME[undefined] would be illegal and FNAME[address] would be illegal. Unbound addresses would contain the memory type. This would permit memory checking of addresses very conveniently (currently it is cumbersome) and would permit forward references to be checked also (??).

3. Multi-statement macro definitions should be added. Perhaps "{" and "}" could be used syntactically to enclose multi-statement stuff.

4. It should be permissible for an argument list to appear to the right of a neutral symbol because of the following usage:

```
P←LB RSH [1]
```

where LBRSH is a neutral symbol, P← is a neutral symbol, and P←LBRSH is a macro. The argument list [1] should be preserved until P←LB RSH [1] is expanded.

5. In every place where an argument string is "looked up" for a builtin, all macros and neutrals should be expanded. In other words, "looking up" an argument should be identical to evaluating an argument, except that occurrence of any builtin causes an error. Expansion stops when a non-neutral non-macro symbol without brackets, parentheses, ←, or : is left.

6. Currently *address←* is handled by the assembler, but *undefined←* and *macro←* are not handled in any special way. Similarly, an undefined source is not handled. It might be useful to have these cases result in the substitutions UDEST[undefined], MDEST[macro] and USRC[undefined]. This would permit forward or external references to succeed where they don't currently and would permit macros which expand to addresses to be used. MDEST, UDEST, and USRC should be macro names selectable by the programmer.

7. Currently the TARGET directive causes a top level statement to be equivalent to

```
TARGLC[(#1)];
```

where #1 stands for the top level statement. This could be changed to a general macro whose first argument is the clauselist of the statement. However, this would slow assembly.

8. Instead of causing an error, integer results should be treated at the top level as neutral symbols equal to the octal text string for the integer. This would permit arithmetic to be performed and the result concatenated with text to select one of many macros or address symbols.

Appendix 3. Binary Output Format

Micro outputs binary memory images as a series of short blocks of 16-bit words. Each block begins with a word that specifies the type of the block; the number and format of following words depend on the block type. During its pass through the source files, Micro outputs a message to the file Micro.fixups whenever it encounters an assignment

FNAME [NAME]

and NAME is undefined. At the end of processing the source files, Micro reads back Micro.fixups and outputs either a type 3 or type 6 message (see below) to the binary file depending upon whether the symbol was a forward reference or undefined. Finally, it orders new or changed address symbols by memory and outputs them to the binary file.

Midas can link up external address references at load time. Address symbols for Midas to use in linking up external references are output as described below.

Table 4: Micro Binary Output File Format

Type	*Followed by*	*Use*
0	nothing	Indicates the end of the binary file.
1	source line # (1 word); data (N words)	Specifies a data word to go in the current memory at the current location. The current location is to be incremented. N is just large enough to cover the width of the memory, and the value is left-justified, e.g., for a 36-bit memory N=3 and the first word goes in bits 0:15, the second in 16:31, and bits 0:3 of the third in 32:35. The source line # is zero if the word was generated by an INSERT file, and has bit 0 set if the word was generated in the main file by a STORE.
2	memory # (1 word); location (1 word)	Sets the current memory and the current location. Memory numbers are related to memory names by type 4 blocks (see below).
3	memory # (1 word); location (1 word); first bit * 256 + last bit (1 word); value (1 word)	Specifies a forward reference fixup. The value is to be stored into the given bits at the given location in the given memory. (Current memory and location settings are not affected.)
4	memory # (1 word); width of memory in bits (1 word); symbolic name of memory (L words)	Correlates a memory number with a user-supplied name. The name is packed 2 8-bit characters per word terminated by a null (all 0's) character; L=(C+2)/2 where C is the number of characters in the name. The type 4 block defining a memory will appear before any type 2 or 3 blocks storing into that memory.
5	memory # (1 word); value (1 word); address symbol name (L words)	Gives the definition of an address symbol. There is a type 5 block

		for every new or changed address symbol. All type 5 blocks appear together at the end of the binary file.

6	memory # (1 word);
	location (1 word);
	first bit * 256 + last bit (1 word);
	undefined symbol name (L words)
		Specifies a reference to an undefined (external) symbol. The first three words have the same interpretation as for block type 3.

The Midas program accepts any of the block types above. In addition, Midas accepts the following compact block types which are more compact than the ones above and use less storage.

11	block address (1 word);
	word count N (1 word);
	N data words;
		The left-half of the word containing the type is the memory #. The N data words are in the same form as block type 1.

12	address (1 word);
	Bcpl string (L words);
		The left-half of the word containing the type is the memory #. The first word of the Bcpl string contains a character count in the first byte (0:7), followed by the characters of the string.