

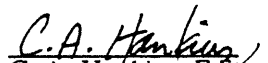
D0 Microprogrammer's Manual

Document: Unassigned
Date: October 1978
Version: 2.0

This manual is intended to provide all necessary documentation for microprogramming a D0. Familiarity with the D0 Functional Specification is assumed. All comments should be addressed to the editor via Laurel.

Release Stage: draft/RELEASED/issued

Approval:


C. A. Hopkins, Editor


W. B. Kennedy, Manager, IOBU

XEROX
BUSINESS SYSTEMS
System Development
Palo Alto, California

TABLE OF CONTENTS

D0 MICROASSEMBLER MANUAL

1.	Introduction	1
2.	Assembly Procedures	1
3.	Error Messages	3
4.	Debugging Microprograms	4
5.	Comments and Conditional Assembly	4
6.	Simplified Parsing Rules	5
7.	Statements Controlling Assembly	6
8.	Integers	7
9.	REPEAT Statements	8
10.	Parameters	9
11.	Constants	9
12.	SETTASK Statements	10
13.	Assembling Data for RM	10
14.	Assembling Data Items in the Instruction Memory	11
15.	RM & STK Clauses	12
16.	ALU Clauses	12
17.	Memory Referencing Instruction Statements	12
18.	Branching	13
	18.1. Branch Clauses	14
	18.2. Dispatch Clauses	15
	18.3. Placement Declarations	15

MICRO: MACHINE-INDEPENDENT MICROASSEMBLER

1.	Introduction	17
2.	Assembly Procedures	17
3.	Error Messages	20
4.	Assembly Listings	21

5.	Cross Reference Listings	22
6.	Comments and Conditional Assembly	23
7.	Statements	24
7.1.	Builtins	25
7.2.	Defining Symbols	25
7.3.	Tokens	27
7.4.	Neutrals and Tails	28
7.5.	Clause Evaluation	30
7.6.	Treatment of Arguments	30
7.7.	Undefined Symbols	31
7.7.1.	Destination Addresses	32
7.7.2.	Octal Numbers	32
7.7.3.	Literals	32
8.	Integers	32
9.	Macros	33
10.	Neutrals	34
11.	Fields, Assignments, and Preassignments	34
12.	Conditionals	35
13.	Memories, Addresses, and Stores	35
13.1.	Target Memory	37
13.2.	Default Statement	37
13.3.	Post Macros	37
14.	Repeat Statement	37
15.	SELECT	38
16.	Bit Tables	38
	Appendix 1. Micro Error Messages	39
	Appendix 2. Limitations of the Language	41
	Appendix 3. Binary Output Format	44
	MICROD MANUAL	46

D0 MICROPROGRAMMER'S GUIDE

1.	Introduction	49
2.	The ALU and Basic Architecture	49
2.1.	Inputs and Outputs	49
2.2.	The Stack	50
3.	The Microinstruction and Branching Conditions	
3.1.	The Microinstruction	51
3.2.	Conditional Branches	51
3.3.	Subroutine Calls	53
3.4.	Dispatch	53
3.5.	Changing Pages	53
3.6.	Notify	54
4.	Special Functions	54
5.	Memory and I/O	
5.1.	General Comments	56
5.2.	Comments on Style	56
5.3.	Quadword Alignment	56
5.4.	Bypassing	57
5.5.	Memory Interlock	57
6.	Getting Started	59
7.	Caveats	60
8.	Suggested Programming Style	60
9.	Sample Programs	62
10.	Common Error Messages	64

MIDAS MANUAL

1.	Midas	65
2.	Starting Midas	65
3.	Midas Display	65
4.	Midas Command Menu	67

5.	Keyboard	69
6.	Command Files	70
7.	Syntax of Command-file Actions	72
8.	Loading Programs	73
9.	Dump and Compare	74
10.	Virtual and Absolute Control Store Interpretation	74
11.	Testing Directly From Midas	75
12.	Scope Loop Actions	77

D0 MIDAS MANUAL

1.	Registers and Memories Known to Midas	79
2.	Task-Specific Registers	80
3.	Complications in the Display of Register Values	80
4.	How Registers Are Read/Written	80
5.	Special Keyboard Input Formats	81
6.	STEP and GO	81
7.	BREAK and UNBREAK	81
8.	BOOT	82
9.	Acquiring Midas	82
10.	Midas Maintenance	82

D0 SIMULATOR MANUAL

1.	Introduction	83
2.	Documentation	83
3.	Getting Started	83
4.	Using DDT	84
5.	Load and Dump	84
6.	Examine and Change	84
7.	Simulator Execution	85
8.	Command Strings	87
9.	DDT Commands	87

10. The Simulator Memories

88

10.1. The C Memory in Detail

88



DO
MICROASSEMBLER
MANUAL

20 October 1978

by

Edward Fiala

edited by

Carol Hankins

Xerox Business Systems
Systems Development Department
3408 Hillview Road
Palo Alto, California 94304



1. Introduction

The D0 microprogramming language, called D0Lang here, is implemented as a set of definitions on top of the machine-independent assembler Micro. The assembly language is based upon the machine description in the 30 July 77 release of "D0 Processor Functional Specification".

If D0Lang were perfect, you would never need to know any details of Micro itself--the language specification described in this document would be complete. I have tried to make D0Lang complete, so if you are forced to modify or augment the D0Lang definition file, please bring the circumstances to my attention.

In the event you are forced to fall back on basic Micro constructs, the documentation on Micro is on-line. It is `Micro.Press` on `Maxc1 <AltoDocs>`. This is supplemented by `<AltoDocs>Micro.Tty`.

The documentation here is also supposed to be complete, so you should not have to study the D0Lang definition file to figure out how anything works. If this proves untrue, please bring it to my attention also.

All numbers in this document (and in D0Lang source files) are in octal.

I personally write microprograms with the upper-case shift-lock key depressed, and the definitions in the microlanguage consist entirely of upper-case characters. However, a Micro switch converts all source file characters to upper-case, so you may follow your own capitalization conventions and use this switch.

Micro flushes Bravo trailers, so you can use Bravo formatting if you want to. However, the cross reference program, `Mcross`, which is expected to produce primary microprogram documentation, does not handle Bravo trailers, so you are advised not to do any special formatting.

2. Assembly Procedures

To assemble microprograms on your Alto, you must obtain from `Maxc <Alto>Micro.run`, `<D1Source>D0lang.mc`, and `<Alto>MicroD.run`. `Micro`, `MicroD`, and `D0Lang` may also be obtained from `Iris <D0>` and `Isis <D0>`.

```
MICRO/L/E d0lang source1 source 2 ... sourceN
```

This causes the source files "d0lang.mc", "source1.mc", ..., "sourceN.mc" to be assembled. The "/L" causes a listing file named "sourceN.LS" to be produced. If "/L" is omitted, no listing file is produced. The assembler also outputs "sourceN.DIB" (intermediate binary and addresses), "sourceN.ER" (error messages--error messages go to the terminal irrespective of whether they are also going to the .ER file), and "sourceN.ST" (the Micro symbol table after assembling source N).

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

```
MICRO SYS/L SYS/B d0lang source 1 ... sourceN
```

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

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

Global:	/L	Produces an expanded listing of the input
	/N	Suppresses binary and symbol table output
	/U	Convert text in all source files to upper-case
	/O	Omit .ST file
Local:	/R	Recover from symbol table file
	/L	Put expanded listing on named file
	/B	Put binary output and symbol table output on named file with extensions .DIB 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 to upper-case

Assemblies are slow--it should take about 3 minutes to assemble a 2048-instruction microprogram.

The symbol table (.ST) file produced by Micro can be used to establish a basis point for further assemblies, thereby reducing assembly time. For example, you can build a DOLANG.ST file as follows:

```
Micro d0lang
```

Then do all further assemblies as follows:

```
Micro/O d0lang/R sys/B source1 ... sourceN
MicroD d0Lang sys
```

Preassembling DOLANG in this way would save about 5 seconds of assembly time. This time savings is so small that I recommend you *do not do it*.

INSERT[file] statements, as described in Section 2.7, can be put in source files so you don't have to type as many source files on the command line.

After obtaining an error-free assembly from Micro, you must postprocess the .DIB file with MicroD to transform it appropriately for loading by Midas. This is accomplished by the following command line syntax to the Alto Executive:

```
MICROD SYS
```

The source files for MicroD (only SYS in the above example) are the output files produced by Micro.

MicroD displays a progress message while it is churning away. I believe that MicroD will require about 3 minutes to process a 2048-instruction file.

The output of MicroD is an ".MB" file, consisting of blocks of data that can be loaded into various D0 memories and of addresses associated with particular locations in memories. The memories are as follows:

IM	40-bit x 4000-word or 10000-word instruction memory (also contains 20 bits/word of placement and other information)
RM	20-bit x 400-word register bank and stack memory

There are at present no facilities provided for microcode overlays. Providing such a facility would require a major addition to MicroD and no such facility will be provided for a long time (maybe never).

3. Error Messages

During assembly, error messages are output to both the display and the error file.

The "TITLE" statement in each source file causes an error message of the form:

```
1...title...ILC = 341
```

This message is not the result of an error. It simply indicates that the assembler has started working on that source file. "ILC=341" indicates that the first IM location assembled in this source file is the 341st in the microprogram. This will be helpful in correlating sources statements with error messages from the postprocessor, MicroD.

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

```
218...error message  
TAG + 39...error message
```

The first example indicates an error on the 218th line of the source file. This form is used for errors that precede the first label in the file. The second form is used afterwards, indicating an error on the 39th line after the label "TAG".

The most common error messages during assembly are due to multiply set fields in instructions and to undefined symbols. I do not believe that you will have any trouble figuring out that these messages mean, so no comments are offered here. The Micro error messages are discussed in Section 3.

MicroD error messages are discussed in Appendix A.

4. Debugging Microprograms

There is a simulator for the D0. See the section on Simulator.

Microprograms can also be debugged on the hardware using facilities provided by Midas. See the section on Midas.

Midas facilities consist of a number of hardware tests, a loader for D0 microprograms, set/clear breakpoints, start, step, or halt the machine, and examine and modify storage. Addresses defined during assembly may be examined on the display.

Midas works with both the imaginary IM addresses defined in your source program and with the absolute IM addresses assigned to instructions by MicroD. The way this works is discussed in the Midas section.

5. Comments and Conditional Assembly

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 one method of producing multi-statement conditional assemblies. This is the COMMENTCHAR feature, used as follows. Suppose you want to have conditional assemblies based on whether the microcode is being assembled for a 2K or 4K D0 configuration. To do this define "~" as the comment character for 2K (i.e., COMMENTCHAR[~];) and "!" as the comment character for 4K. Then in the source files:

```
*! 2K configuration only
...statements for 2K configuration...
*! end of 2K conditional
*~ 4K configuration only
...statements for 4K configuration...
*~
```

In other words, "*" followed by the comment character is equivalent to "%" and is terminated by its next occurrence.

6. Simplified Parsing Rules

After comments, false conditionals, and non-printing characters are stripped out, the rest of the text forms STATEMENTS.

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 the assembly of a statement, an error message is output. If you ever experience a statement buffer overflow error, please tell me. This should be impossible except on multi-statement REPEAT's.

The special characters in statements are:

"[" and "]"	for enclosing built-in, 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 and 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, and tabs are syntactically meaningless (flushed by the prescan), so "P+Q" = "P + Q", each of which is a single symbol.

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

Statements are divided into CLAUSES separated by commas, and the clauses are evaluated right-to-left. 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 referred to as a "source" while FOO←, FOO1←, and FOO2← are "destinations" or "sinks".

```
P←STEMP,  
NAME[N1[N2[ARG]],ARG2]←FOO[X],
```

Further discussion about clause evaluation is postponed until later.

7. Statements Controlling Assembly

Each source file should begin with a TITLE statement as follows:

```
TITLE[SOURCE1];
```

The TITLE statement performs a number of operations.

- a. It prints a message in the .ER file and on the display which will help you correlate subsequent error messages with source statements which caused them.
- b. It puts the assembler in TASK 0 mode and SUBROUTINE mode. These modes will be discussed later.

The final file to be assembled should be terminated with an END statement:

```
END;
```

Currently, the END statement doesn't do anything, but I might find something for it to do later.

You may at any place in the program include an INSERT statement:

```
INSERT[sourceX];
```

This is equivalent to the text of the file sourceX.MC. However, since INSERT is defined by DOLANG, you cannot INSERT DOLANG itself-either DOLANG itself or a /R file which assembled DOLANG must be explicitly mentioned on the command line or an INSERT function must be defined in the file such as:

```
BUILTIN[INSERT,24];  
INSERT[DOLANG];
```

The message printed on the .ER file by TITLE is most helpful in correlating subsequent error messages if any INSERT statements occur either before the TITLE statement or at the end of the file (before the END statement). INSERT works ok anywhere, but it might be harder to figure out which file suffered an error if you deviate from this recommendation.

In the event you request a listing by putting "/L" in the Micro command line, the exact stuff printed is determined by declarations that can be put anywhere in your program.

D0Lang selects verbose listing output. However, unless you are looking for an elusive assembly problem, you will generally NOT want to print this listing. The listing produced by MicroD is the normal listing file you will use during debugging.

If you want to modify the default listing control in D0Lang for any reason, you can do this using the LIST statement, as follows:

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

where the "memory" may be any of the following:

```
IM    4000-word or 10000-word x 40-bit (+20-bit placement) instruction memory
RM    400-word x 20-bit register bank memory
```

and the mode, the "OR" of any of the following:

```
10    alphabetically-ordered list of address symbols
4     numerically-ordered list of address symbols
2     (TAG) FF+3, JCN+4, etc. (list of field stores)
1     (TAG) nnnn nnnn nnnn (octal value printout)
```

NOTE: The listing output will be incorrect in fields affected by forward references (i.e., references to as yet undefined addresses).

8. Integers

Micro provides a number of built-in operations for manipulating 20-bit assembly-time integers. These have nothing to do with code generation or storage for any memories. Integers are used to implement assembly-time variables and to control REPEAT statements. The operations given in the table below are included here for completeness, but hopefully you will not have to use any of them except SET:

SET[NAME,OCT]	Defines NAME as an integer with value OCT. Changes the value of NAME if already defined.
SELECT[i, C0, ... , Cn]	i must be an integer 0 to n. Evaluates C0 if i = 0, C1 if i = 1, etc.
ADD[01, ... , 08]	Sum of up to 8 integers 01 ... 08.
SUB[01,02]	01-02
IFE[01,02,C1,C2]	Evaluates clause C1 if 01 equals 02, else C2.
IFG[01,02,C1,C2]	Evaluates C1 if 01 greater than 02, else C2.
NOT[01]	Ones complement of 01.
OR[01,02, ... , 08]	Inclusive 'OR' of up to 8 integers.
XOR[01,02, ... , 08]	Exclusive 'OR' of up to 8 integers.
AND[01,02, ... , 08]	'AND' of up to 8 integers.
LSHIFT[01,N]	01 lshift N
RSHIFT[01,N]	01 rshift N

OCT in the SET[NAME,OCT] clause, may be any expression which evaluates to an integer, e.g.:

```
SET[NAME,ADD[NOT[X],AND[Y,Z,3],W]]
```

Where W, X, Y, and Z are integers.

If you want do arithmetic on *addresses*, then the addresses must be converted to integers using the IP operator, e.g.:

```
IP[FOO]           takes the integer part of the address FOO
ADD[3,IP[FOO]]   is legal
ADD[3,FOO]       is illegal
```

Some restrictions on doing arithmetic on IM addresses are discussed later.

9. REPEAT Statements

The assortment of macros and junk in the DOLANG file successfully conceals Micro's complicated macro, neutral, memory, field, and address stuff for ordinary use of the assembler.

However, one special situation that may require you to understand underlying machinery is REPEAT statements--in a diagnostic you might want to assemble a large block of instructions differing only a little bit from each other, and you want to avoid typing the same instruction over and over.

Instructions statements are assembled relative to a location counter called ILC. This is originally set to 0 and is bumped every time an instruction is assembled. To do a REPEAT, you must directly reference ILC as follows:

```
REPEAT[20,ILC[ (... INSTRUCTION STATEMENT ... )]];
```

This would assemble the instruction 20 times. If you want to be bumping some field in the instruction each time, you would proceed as follows:

```
SET[X,0];
REPEAT[20,ILC[(SET[X,ADD[X,1]] ... instruction statement ... )]]
```

where the instruction statement would use X someplace.

For a complicated REPEAT, you may have to know details in DOLANG. For this you will have to delve into it and figure out how things work.

Multi-instruction REPEAT's are also possible. The "ILC[...]" in the above example can be used several times to accomplish this. However, the 500-character size of the statement buffer will limit the complexity of the REPEAT body to only a few instructions.

10. Parameters

Parameters are special assembly-time data objects that you may define as building blocks from which CONSTANTS, RM, or IM data may be constructed. Two macros define parameters:

MP[NAME,OCT];	makes a parameter of NAME with value OCT
SP[NAME,P1,...,P8];	makes NAME a parameter equal to the sum of P1, ..., P8, which are parameters or integers.
NSP[NAME,P1,...,P8];	makes NAME a parameter equal to the ones complement of the sum of P1, ..., P8, which are parameters or integers.

The parameter "NAME" is defined by the integer "NAME!"¹, so it is ok to use the NAME again as an address or constant. However, you cannot use it for more than one of these.

NOTE: The MC and NMC macros discussed in the next sections not only define constants, but also parameters with the same name (i.e., NAME!) and value.

¹The "!" is a symbol constituent added so that a constant or RM address can have an identical NAME.

11. Constants

The hardware allows 10-bit constants to be output in either the left or right halves of ALUB with 0's in the other half of the word. In conjunction with arithmetic ALU operations, the right-half constant is sign-extended.

The assembler permits literal constants to be written as "122C", "177400C", "177600C", "122000C", etc. These can be inserted in microinstructions without previous definition. The assembler error-checks the ALU operation in cases where the selected constant requires or prohibits sign-extension.

Negative constants such as "-1C", "-55C", etc., are presently *illegal*. However, they may be implemented later, if I can figure out how.

Alternatively, constants may be constructed from parameters, integers, or addresses using the following macros:

MC[NAME]P1,...,P8];	defines NAME as a constant whose value is the sum of P1...P8 (integers or parameters).
NMC[NAME,P1,...,P8];	defines NAME as the ones complement of the sum.

NOTE: The two macros above also define NAME as a parameter. You *must not* redefine a parameter with the same name as a constant because the binding of the constant is to the name of its associated parameter, not to its value. In other words, if you redefine a parameter with the same name as a constant, you will redefine the constant also.

Occasionally, you may wish to create a constant whose value is an arithmetic expression or an expression including an address in RM. Here are several examples of ways to do this:

IP[RADDR]C
 ADD[3,LSHIFT[X,4]]C

A constant whose value is an RM address
 A constant whose value is a function of the integer X

12. SETTASK Statements

The hardware OR's various bits of the task number into fields of the microinstruction to determine which RM addresses are referenced. You must tell the assembler what task is going to execute each section of microcode, so that it can perform the proper error checks and set up the fields of microinstructions appropriately.

This is done with a clause of the form:

```
SET TASK[n];
```

where n is the task number, 0 to 17. If you want to refer to task numbers symbolically, you can define integers with values equal to the task numbers. For example:

```
SET[DISPTASK,3];
```

Then use SETTASK[DISPTASK] to refer to the task.

SETTASK controls not only the assembly of instructions, but also the allocation of RM addresses to 100-word sections of RM, as discussed in the next section.

NOTE: The TITLE statement at the beginning of a file does a SETTASK[0].

13. Assembling Data for RM

RM addresses are allocated by RV statements in one of the following ways:

```
RV[name,disp,P1,P2,...,P7];
RV[name,,P1,P2,...,P7];
RV[name,disp];
RV[name];
RV[name,disp,value];
RV[name,,value];
```

The first argument "name" is the name of the RM address which you will subsequently use in instruction statements.

The second argument "disp" is a displacement between 0 and 77. This specifies the low six bits of the RM address. The top two bits are determined by the top two bits of the task number, declared by the last SETTASK statement. If "disp" is omitted, the RM address is allocated at the last location plus 1.

The remaining 7 arguments are parameters summed to determine the value loaded into that location. If all of these are omitted, then the location will be uninitialized.

Avoid assigning useless initial values to variables because this will prevent the "Compare" function in Midas (which compares the microstore image against what you loaded) from reporting fictitious errors. In a system microprogram (as opposed to a diagnostic), any occurrence of a variable with an initial value is probably a programming error since it requires reloading the microcode to restore the initial value. Hence, if you have variables with initial values, you probably should store the initial values elsewhere (in IM, for example), and copy the initial values into the registers during initialization.

The hardware imposes a number of strange constraints upon the placement of RM addresses. For example, addresses used as base registers must be less than 4 mod 8, quadruple fetch/store locations must be 0 mod 4, double fetch/store locations must be even. Also, RM is partitioned so that only locations 0 to 77 are accessible to tasks 0 to 3, 100 to 177 to tasks 4 to 7, 200 to 277 to tasks 10 to 13, and 300 to 377 to tasks 14 to 17. Tasks 1 to 3 in each group of 4 are further limited because the task number is OR'ed into high address bits in various ways. These constraints will be a source of many program bugs.

You must be careful to assign a "disp" that satisfies all the uses of each RM address. If you screw up, the assembler will give you an error message when you subsequently reference the RM location in an instruction.

Sometimes you may want to use several different names to refer to the same RM location. To do this, define the first name with RV, as above; then define the synonyms as follows:

```
RM[FOO,IP[FOO]];
```

This defines the address FOO1 at the same location as the (previously-defined) address FOO.

14. Assembling Data Items in the Instruction Memory

If you do not want to clutter RM with infrequently referenced constants or variables, and if you are willing to cope with the hardware kludges for reading/writing the instruction memory as data, then you can store data items in IM.

To assemble a table of data in the instruction memory:

```
SET[T1LOC,100];
  DATA[(TABLE1:LH[P1,...,P8] RH[P1,...,P8], AT[T1LOC])];
  DATA[(LH[P1,...,P8] RH[P1,...,P8], AT[T1LOC,1])];
  ...
```

where TABLE1 is an IM address symbol equal to the location of the first instruction in the table, P1, ..., P8 are parameters, integers, or addresses. LH stores the sum of up to 8 parameters in the left-half of the IM word and RH, the right-half. "AT" is discussed in Section 2.18.3. Sample sequences for reading and writing IM are given in Section 5.

15. RM & STK Clauses

The hardware complicates references to RM by providing only six bits of RM address in the microinstruction. The remaining two address bits come from the task number. The programmer must declare the task number with SETTASK before referencing any variables or constants.

RM addresses can source ALUA destinations and can be used in ALU expressions. In this case, the RM address has to be enclosed in "()".

RM addresses can be used as destinations for ALU operations and ALU sources (which the assembler routes through the ALU). For these simply write the register name followed by "+".

16. ALU Clauses

The operations performed by the ALU are given below. In these expressions, the "A" component of the ALU expression may be any RM address or one of the other "A" sources. These must be enclosed in "()". The "B" component may be constant, enclosed in "()" or T. "()" are optional around T.

17. Memory Referencing Instruction Statements

Instruction statements that initiate memory references or INPUT have a different form from regular instructions, as discussed in the hardware manual. Branch and placement clauses are identical to those in regular instructions, and the F2 clause, if any, is identical to that in a regular instruction. The rest of the instruction is a single clause in one of the following forms:

```
PFETCH1[rbase,rdest < ,f2 >]
PFETCH2[rbase,rdest < ,f2 >]
PFETCH4[rbase,rdest < ,f2 >]
PSTORE1[rbase,rsource < ,f2 >]
PSTORE2[rbase,rsource < ,f2 >]
PSTORE4[rbase,rsource < ,f2 >]
IOFETCH4[rbase,device < ,f2 >]
IOFETCH20[rbase,device < ,f2 >]
IOSTORE4[rbase,device < ,f2 >]
IOSTORE20[rbase,device < ,f2 >]
WRITEMAP[rbase,rsource < ,f2 >]
READMAP[rbase,rdest < ,f2 >]
INPUT[raddr < ,f2 >]
```

In these clauses, "rbase" is an RM address which must be in the group of 100 accessible to the current task (see "SETTASK") and less than 4 mod 8. The two words of base address are taken from the selected RM address and that location +4. The assembler will give an error if you use an invalid RM address.

The displacement relative to the base register is taken from T, if you omit the optional f2 argument (" $\langle \rangle$ " above denotes an optional argument). If you supply the f2 argument, which must be in integer less than 20, that value is stored in the F2 field of the microinstruction and used instead of T. See the hardware manual for details on how this works.

PFETCHn will then move n words from the memory to the n-word block of RM addresses beginning at "rdest". "rdest" must be even for PFETCH2 and 0 mod 4 for PFETCH4; it must be in the group of 100 (task 0 mod 4), 40 (tasks 1 mod 4 and 2 mod 4), or 20 (task 3 mod 4) RM locations accessible to the task--the assembler will give an error message if "rdest" is illegal.

PSTOREn is like PFETCHn, but moves data from RM to memory.

IOFETCHn moves n words from memory to the selected IO device, where the IO device must be specified by an integer. The hardware OR's the current task number with the 8-bit device in the instruction, and the assembler will give an error message if the device you code is inaccessible to the task.

IOSTOREn is like IOFETCHn, but moves data from the device to memory.

NOTE: The hardware OR's the current task number into the RM address in the microinstruction so that a group of 4 tasks will use different RM locations, while executing a single stretch of microcode. Suppose, for example, that you want tasks 10, 11, 12, and 13 to share a section of microcode but use independent RM locations. Then do a SETTASK[10] before that section of microcode, allocate a block of RM locations in the range 100 to 117 and refer to these locations in the stretch of microcode; also allocate parallel blocks of RM locations in the ranges 120 to 137, 140 to 157, and 160 to 177 for use by tasks 11, 12, and 13. In this way, the program will do what you want. If the stretch of microcode also refers to constants, allocate these in the range 160 to 177, so that they will be accessible to all four tasks.

18. Branching

This section defines branch clauses in instruction statements, declarations which affect instruction placement, and dispatch clauses.

Micro assembles instructions for an imaginary machine identical to D0 but with additional fields assembled for its postprocessor. The imaginary machine is characterized by full-size 12-bit branch addresses in instructions.

A postprocessing program called MicroD places instructions and transforms the .DIB (micro binary) output file for the imaginary machine into a .MB file for D0.

18.1. Branch Clauses

The assembly language defines several constructs of the form:

```
GOTO[branch address, branch condition 1, branch condition 2]
```

where both branch conditions are optional.

The branch addresses for these may be either instruction tags or one of the following special symbols: `-.3` `-.2` `-.1` `..+1` `..+2` `..+3`, where `..` refers to the current instruction and the others are relative to this in-line.

[It is obviously possible to define `-.4`, `..+4`, `..-5`, etc., but my feeling is that it is bad style to jump further than `+/-3` without using a tag. If anyone finds this inconvenient, please let me know.]

When complementary branch conditions are used, the assembler simply reverses the order of the branch tags. Hence, `DBLGOTO[TAG1,TAG2,com C1, com C2] = DBLGOTO[TAG2,TAG1,C1,C2]`. This is provided as a programming convenience.

NOTE: If two branch conditions appear in a statement, they must be both regular or both complementary. When two regular branch conditions are used, the true path takes if *either* is true. However, when two complementary branch conditions are used, the true path takes only when *both* are true. Don't get confused by this.

Below "`< >`" denote optional args; C1 and C2 either two hardware branch conditions or complements of two hardware branch conditions:

RETURN	To LINK (smashes LINK also).
CORETURN	Like RETURN but LINK <code>..+1</code> and next instruction in-line placed at <code>..+1</code> .
DBLGOTO[TAG1,TAG2,C1<,C2>]	To TAG1 if C1 or C2 true, else to TAG2. Limits TAG2 to the goto addresses.
DBLCALL[TAG1,TAG2,C1<,C2>]	= DBLGOTO[TAG1,TAG2,C1,C2], forces next instruction in-line to be at <code>..+1 mod 100</code> , and limits TAG2 to call addresses.
CALL[TAG<,C1<,C2>>]	= DBLCALL[TAG, <code>..+1</code> ,C1,C2], complementary BC's illegal
GOTO[TAG<,C1<,C2>>]	= DBLGOTO[TAG, <code>..+1</code> ,C1,C2]

A conditional CALL is just barely possible. It requires the next instruction in-line to be simultaneously at the true branch address `..xor 1` and at the address of the caller `..+1`. Since the true branch address must be at a location with three low bits equal 001, these conditions are only met when the address of the caller is the location before the false target address. In other words, complementary BC's are illegal with CALL, and you cannot code two consecutive microinstructions each containing a conditional CALL.

It is also impossible to have a CALL in the instruction after a conditional GOTO because the return of the CALL would be to the true target of the previous conditional branch.

An unconditional RETURN branches to the address of the caller `..+1`. There is no placement constraint on an instruction containing a RETURN.

A conditional RETURN is not defined by the hardware.

If omitted, the branch clause is defaulted to GOTO[.+1].

18.2. Dispatch Clauses

The assembly language defines the following dispatch clauses (or slow branches):

```
DISPATCH[RADDR,POS,SIZE] Dispatch on 1 to 8 bits from RADDR
BBFA[RADDR]
NEXTINST[RADDR]
```

An example using a dispatch clause is given in the next section.

18.3. Placement Declarations

An instruction containing the clause "AT[N]" will be forced by the assembler to appear at absolute location N in the microstore. This will be necessary for instructions in dispatch tables.

"AT[N1,N2]" in an instruction is equivalent to AT[ADD[N1,N2]]. For example, an 8-way DISPATCH might be written as:

```
DISPATCH[RTEMP,0,3];
..., GOTO[SWITCH];

SWITCH: SET[SWLOC,320];
..., AT[SWLOC]; *B[15:17] = 0
..., AT[SWLOC,1]; *B[15:17] = 1
...
..., AT[SWLOC,7]; *B[15:17] = 7
```

where the three instructions in the dispatch need not be consecutive in the assembly source.

NOTE: Because microinstruction addresses are unknown during assembly, it is illegal to create parameters, constants, or R-memory data referring in any way to instruction locations. To do this, you must manually locate the affected instructions with "AT" statements and do arithmetic on integers with the same values as the instruction locations.

Global entries are declared by a "GLOBAL" clause in a statement, e.g.:

```
DONEXT: RETURN, T←377C, GLOBAL;
```

GLOBAL declarations cause placement at one of the 20 global call locations in the microstore.

It would probably be nicer for the assembler to have some way of positioning an instruction at a boundary of 4, 10, 20, etc., without forcing the absolute location to be completely specified. However, I decided this was harder to implement and it will not be provided--you are stuck with "AT" for all dispatch tables.

MICRO
MACHINE-INDEPENDENT
MICROASSEMBLER

29 August 1978

by

Edward Fiala
Peter Deutsch
Butler Lampson

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

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

This manual is the property of Xerox Corporation and is to be used solely for evaluative purposes. No part thereof may be reproduced, stored in a retrieval system transmitted, disseminated, or disclosed to others in any form or by any means without prior written permission of Xerox.



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 editing 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 messages to the error file (and to the terminal).

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

Blanks are squeezed out of the message by the prescan so "-" signs or other printing characters should be used instead.

Stopcode equal 0 continues assembly; non-zero aborts assembly (nulstring in the stopcode defaults to 0).

ER first prints the message (a literal string) on the error file; then, if the value argument is present, evaluates it (c.g., it may be an IP or other arithmetic expression) and prints it in octal on the error file; then, if stopcode is non-zero, aborts the assembly.

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 error message 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-reference 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 and Conditional Assembly

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 one method of producing multi-statement conditional assemblies. This is the COMCHAR builtin, which 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 ← STMP
NAME[N1[N2[ARG]],ARG2]←FOO[X].
```

P+Q+1 is a "source" while FOO, FOO1, and FOO2 are "destinations" or "sinks."

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,I];
```

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

<i>Builtin No.</i>	<i>Name</i>	<i>Discussion</i>
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

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 Maxc1), 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 arbitrary 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

<i>Symbol type</i>	<i>Action</i>
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); 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:

<i>Type of Name</i>	<i>Type of Value</i>	<i>Action</i>
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#];
```

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.

12. 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[i1, i2, (true), (false)]	i1 = i2
IFG[i1, i2, (true), (false)]	i1 > i2
IFDEF[s1, (true), (false)]	s1 in symbol table and not unbound address
IFSE[s1, s2, (true), (false)]	s1 = s2
IFA[field, (true), (false)]	any bit of field previously assigned
IFME[address, s1, (true), (false)]	memory name for address = string

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 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 Statement

```
REPEAT[il,TEXT]
```

assembles TEXT il 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 "H.C[TEXT]" or whatever each time TEXT is repeated. Note that the statement buffer is cleared after each assembly of TEXT.

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 anything at all, 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=17777 (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].

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 EXIST¹

COULD NOT OPEN FILE @ FOR 'INSERT'¹

STORAGE FULL¹

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

$1/2 * \text{Sum} [\text{namelength} + 1]$ for all symbols
 + 6* no. symbols
 + $1/2 * \text{Sum} [\text{length} + 1]$ 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 MEMORIES¹

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 USED¹

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 SET¹

NO TARGET FOR FIELD SET¹

'TARGET' NOT LEGAL INSIDE A STORE¹

@ 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. Conditionals or 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←PI is accomplished by setting the PS field to 50. What we would *like* to do is to define neutrals P+ and PI and then define the macro P←PI as PS[50]. If the hardware is changed so that P←PI is accomplished by PS[20] instead of PS[50], we

would prefer to change only the one macro P+PI. 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+PI 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 conditionals and 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 clause list 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.



MICROD MANUAL

20 October 1978

by

Peter Deutsch

edited by

Carol Hankins

Xerox Business Systems
Systems Development Department
3408 Hillview Road
Palo Alto, California 94304



MicroD takes microprograms for the Dorado or D0, assembled by Micro, and completes the assembly process by assigning absolute locations to the microinstructions. The resulting file can be loaded into a D- machine by Midas and run. MicroD's job is to find a way to assign locations to microinstructions in a way that satisfies both the semantics of the source program and the peculiar addressing restrictions of the hardware.

This document is deliberately somewhat sketchy, since it assumes that its readers have already absorbed the necessary "culture" surrounding D-machine microprogramming and just want to know how to convert Micro output into Midas input. At some future date it may be expanded to be more helpful to people just getting started.

The simplest way to use MicroD is to assemble your entire microprogram at once with Micro, producing a single file xxx.DIB. (DIB stands for "D-machine Intermediate Binary".) Then you invoke MicroD as follows:

```
MicroD xxx
```

to produce a listing file xxx.DLS and a final binary file xxx.MB which can be fed to Midas.

MicroD normally produces a listing with the following parts:

- The name and initial contents of each defined R memory location.
- The initial contents of each IFU and ALUF memory location.
- The label and octal representation of each microinstruction.
- A summary of how much of each page of I (microinstruction) memory was used.

MicroD accepts the following global flags which affect the listing:

- /N (No listing) - only produce the summary
- /C (Concise) - produce everything but the octal contents of I memory

The following global flags produce additional information, not useful to the ordinary user:

- /D (Debug) - print a large amount of debugging information
- /T (Trace) - print a trace of the calls on the storage allocator

Normally MicroD produces its output on xxx.DLS and xxx.MB, where xxx is the name of the last (or only) input file. You can specify a different name with the local /O switch, e.g.

```
MicroD xxx yyy/o
```

to process xxx.DIB but produce yyy.DLS and yyy.MB.

If you wish, you can assemble your microprogram in pieces and let MicroD link the pieces together. (This can save a large amount of assembly time for large programs.) Suppose your program consists of the following parts: some definitions defs1.MC and defs2.MC; one large piece of code this1.MC and this2.MC; another large piece of code that.MC. Then you can proceed as follows:

```
Micro saveit/s defs/b defs1 defs2
```

This assembles the definitions, saves Micro's state on saveit.ST, and produces a file defs.DIB.

```
Micro saveit/r this/b this1 this2
```

This resumes assembly with the definitions saved in saveit, producing this.DIB. Micro will give you a list of "undefined symbols", which are references to symbols not defined in this1 or this2 (presumably defined in that).

Micro saveit/r that

This again resumes assembly with the saved definitions, producing that.DIB. Again, Micro will list the symbols not defined in that (presumably defined in this1 or this2).

MicroD myprog/o defs this that

MicroD will link together any references from this to that (or vice versa) and produce the output files myprog.DLS and myprog.MB.

Note that you do not need to do anything special in your source files to declare labels which are exported (defined here, used elsewhere) or imported (used here, defined elsewhere): Micro assumes that any undefined symbol is meant to be imported (but gives you the list just so you can check), and MicroD assumes that all labels are exported. MicroD also discards all but the last definition of a name (e.g. the name ILC is defined in every file as the address of the last microinstruction).

If you have multiple .DIB files, you can control the listing mode (normal, No listing, or Concise) for each file individually by using /L (List), /N, or /C as a local switch on the file name. The global switch, if any, applies to any input file that lacks a local switch. For example, to get only a concise listing for the second part of the program in the above example, you can use

MicroD/n myprog/o defs this that/c

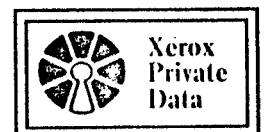
D0 MICROPROGRAMMER'S GUIDE

22 August 1978

by

Carol Hankins

Xerox Business Systems
Systems Development Department
3408 Hillview Road
Palo Alto, California 94304





1. Introduction

You will never get to be a super microcoder until you memorize and understand the architecture of the machine. It is assumed that you have made some attempt to understand the D0 Functional Specification, and that you are now ready to program. This manual breaks the machine into small parts and describes the use of each section in detail. Accompanying each part are some examples of microcode which illustrate the features of the machine.

All numbers in this manual should be considered octal. When decimal is required, the number will be suffixed with a "D". Any number followed by a "B" is octal.

Throughout the manual, a register called "rtemp" will be used.

The D0 Functional Specification alluded to in this manual is dated January 16, 1978.

2. The ALU and Basic Architecture

2.1. Inputs and Outputs

The last page of this section is a diagram of the processor. In the center is a box labelled ALU. There are two inputs, labelled A and B. The A input comes from the R registers through the cyclor/masker. Also on this bus are the special-purpose R registers, such as APC, PCF... The B bus comes from T. Notice the way constants are put on the bus. You can have a constant *or* T - not both. Constants are eight bits - all of which must be entirely contained in the left or right half of the word. There is one output from the ALU which goes into T and R, and also into the memory map.

Notice the signal coming in the top of the ALU, called ALUControl. There are two ways of controlling the ALU's operations: from the ALUF field of the microinstruction, or from a special box called SALUF. The ALU is really a unit with 64 operations, with 14 of the most common mapped into the ALUF field of the microinstruction. All 64 functions may be accessed by loading SALUF. See the D0 Functional Specification for an enumeration of these functions. The ALU operations are:

<u>ALUF</u>	<u>ALUOut =</u>
0	B
1	A
2	A AND B
3	A OR B
4	A XOR B
5	A AND NOT B
6	A OR NOT B
7	A XNOR B
8	A+1
9	A+B
10	A+B+1
11	A-1
12	A-B
13	A-B-1
14	unassigned
15	use SALUF for ALU function



The closest piece of hardware to the input of the ALU along the A bus is the cycler/masker. The cycler/masker is used to manipulate bits from an R register, and provides some standard shifting and masking operations. The following operations are available:

LDF[rtemp, pos, size] - right justify value rtemp of length "size" beginning at bit "pos"
 RSH[rtemp, count] - right shift rtemp by "count"
 LSH[rtemp, count] - left shift rtemp by "count"
 RCY[rtemp, count] - right cycle rtemp by "count"
 LCY[rtemp, count] - left cycle rtemp by "count"
 RHMASK[rtemp] - rtemp AND 377C
 LHMASK[rtemp] - rtemp AND 177400
 ZERO - a way to load a 0 on the bus
 DISPATCH[rtemp, pos, size] - see the section on the jump conditions

The cycler/masker is controlled by a translation of the above instructions into the special function field in the microinstruction. This field will be described in more detail in section 3.0, Special Functions.

Already you know how to write simple microinstructions for manipulating the ALU. Note that Micro requires parentheses around the A input to the ALU. Parentheses around the B input are necessary if it is a constant, optional otherwise. Here are some examples of legal and illegal instructions:

Legal

T ← (rtemp) + T;	* an A input and a B input. Note parentheses
rtemp ← (rtemp) + T;	* can store into rtemp or T
T ←, rtemp ← (rtemp) + T;	* can store into both
T ← (LDF[rtemp, 14, 4]) + T;	* LDF is a cycler/masker function
rtemp ← (RHMASK[rtemp]) XOR T;	* RHMASK is a cycler/masker function
rtemp ← (rtemp) + (377C);	* constant with lower 8 bits
rtemp ← (rtemp) + (177400C);	* constant with upper 8 bits
T ← (zero) + T + 1;	* this is the only way to add 1 to T. zero is an
	* output of the cycler/masker, and A+B+1 is
	* an ALU function..

Illegal

rtemp ← rtemp + T;	* no parentheses around the A source
T ← T + (37C);	* two B bus sources
rtemp ← (rtemp) + (177777C);	* constant is more than 8 bits
rtemp ← (rtemp) + (770C);	* the 8 bits cross a byte-boundary
T ← (LDF[rtemp, 14, 4]) + (37C);	* LDF uses F field and so does constant
T ← (RHMASK[T]);	* T is not on the A bus
rtemp ← T + 1;	* no ALU function of this type

2.2. The Stack

There is a 20B word stack which is loaded from an R register. There are actually two constructs which deal with the stack: STKP and STACK. STKP is the stack pointer which points to an arbitrary location in the R register bank, while STACK contains the contents of the R register pointed to by STKP. Since the stack is on the A bus, STKP gets loaded from this bus. The actual loading of STKP (STKP←) is a special function; this is necessary since there would be two R addresses in the microinstruction otherwise. Reading STKP returns the *complement* of the value; writing STKP is normal. When reading the stack, it is possible to update STKP in the same microinstruction. Several options can be appended onto STACK, such as &+1, &-2... which result



in automatic updating of STKP.

Legal

STKP+rtemp;
t←STKP;
t←STACK;
t←STACK&+1;
t←STACK&-1;
STACK&+1←t;

- * load occurs from A bus
- * t contains *complement* of STKP
- * t will contain contents of R register pointed to by STKP
- * after this, STKP will be incremented by one
- * read and decrement
- * store and increment

Illegal

STKP+t;
STKP+(37C);

- * load can't happen from B bus
- * constant is from B bus

Caution: The stack is operated on modulo 20B, so if STKP points at register 77B, executing STACK&+1 will have STKP equal to 60B.

Note: STKP is the only way of accessing any register in R memory.

3. The Microinstruction and Branching Conditions

3.1. The Microinstruction

Since you would like to do more than arithmetic and logical functions and perhaps more importantly, you would like to maximize the work that you can get from one instruction, the following table shows the fields in a non-memory microinstruction:

NORMAL	0 or 1 - depends on if it's a memory operation or not
RMOD	used for addressing the special R registers (e.g. APC, SB, DB...)
RSEL	used for R addressing. NOTE: only 1 R address per m-i.
ALUF	what the ALU is supposed to do
BSEL	what is supposed to be on the B bus (T or constant)
F1	special function
F2	special function
LR	load R
LT	load T
JC	jump control - call, goto, return, dispatch
JA	where to go next. NOTE: addressing is 8 bits = > page-relative

You know about RSEL, ALUF, BSEL, and the loading of R and T. The remainder of this section will discuss the branching mechanisms and the control logic of the D0. The next section will explain the special function fields and their uses.

Alteration of the flow of control is accomplished by the JC (jump control) and JA (jump address) fields. Each microinstruction must indicate its successor. If you do not instruct otherwise, a microinstruction will be followed by the next instruction in your program. You can modify this in many ways. A simple GOTO[label] will cause the JA field to contain the address of "label".

3.2. Conditional Branches

For the programmer's convenience, several branch conditions exist and alter the flow of control when tested. There is a programming feature called DBL.GOTO which has the form



DBLGOTO[label1, label2, branch-condition]. If branch-condition is true, control will be transferred to label1, if not the next instruction will be label2. The processor requires that these two labels be one bit apart in their address. These are guaranteed to get you into trouble if you do not remember the instruction-placement constraints. The table below describes the placement constraints for "label1". Label1 will occupy an odd location if the condition is listed in the goes-to-odd column below:

JC,,JA	goes-to-odd	goes-to-even	BRANCHSHIFT	time	page
000	ALU#0	ALU=0	0	t3	18
001	CARRY	NOCARRY	0	t3	18
010	ALU<0	ALU>=0	0	t3	18
011	QUADOVF	INQUAD	0	t3	45
100	R<0	R>=0	0	t1	
101	R ODD	R EVEN	0	t1	
110	NOATTEN	IOATTEN	0	t3	62
111	MB	NOMB	0	t1	24
000	INTPENDING	NOINTPENDING	1	t3	
001	NOOVF	OVF	1	t3	18
010	BPCCHK	BPCNOCHK	1	t3	
011	SPAREBRANCH	NOSPAREBRANCH	1		

Note: GOTO[label1, branch-condition] is a degenerate case of DBLGOTO with label2 = current location + 1.

The column labelled "time" refers to the time that this condition is available for testing. If "t3" is listed, you should test this condition in the instruction *following* the instruction which could generate the condition. Conditions listed as "t1" can be tested during the current microinstruction.

The BRANCHSHIFT column deals with special functions (in particular, F1), and will be discussed fully in section 3.0, Special Functions.

The page column refers to the page in the D0 Functional Specification where more information about these conditions can be located.

Note: Don't memorize the above table. In general you won't have to worry about even or odd placement. Micro does it's best to let you do what you want. You will need this table only when MicroD tries to place the instructions in the control store, and cannot succeed because of the above constraints.

Following are examples of instruction placement:

T ← (rtemp) + (377C); DBLGOTO[L1, L2, ALU#0]; regular-instruction; L1: mumble2; L2: mumble3;	* notice test during instruction following operation * at an odd location (L2 OR 1) * at an even location
T ← (rtemp) - T; DBLGOTO[L1, L2, ALU > =0]; mumble; L1: mumble2; L2: mumble3;	* at an even location * at an odd location (L1 OR 1)
DBLGOTO[L1, L2, R < 0], LU ← rtemp; mumble; L1: mumble2; L2: mumble3;	* notice test during current instruction * at an odd location (L2 OR 1) * at an even location



3.3. Subroutine Calls

There is a mechanism for one-level subroutines. These are accomplished by an instruction of the form CALL[label]. When a RETURN is executed, control will be given to the call instruction+1.

Example: Suppose that you wish to begin execution at INIT.

```
DoubleRAndT:  T ← rtemp + (rtemp) + T;
              rtemp ← (rtemp) + T, RETURN;      * next instruction will be "MUMBLE"

INIT:        rtemp ← (4C);
              T ← (37C);
              CALL[DoubleRAndT];
              mumble;
```

3.4. Dispatch

DISPATCH is a cypher/masker function which allows the next instruction to be one of sixteen possible addresses. The lower four bits of APC are selected via DISPATCH[rtemp, pos, size]. You must use the "AT" construct to nail down the targets of the dispatch table. This tells MicroD that you really know where you want this instruction to go. A trivial example of dispatch is as follows:

```
D:  DISPATCH[rtemp, 10, 4];          * dispatch on bits 10:13B of RTMP
    DISP[D0];                       * set up label for dispatch
    SET[DLOC, 20];                  * note dispatch table on 16 word boundary

D0: goto[X], rnext ← (0C), AT[DLOC, 0];
D1: rnext ← (1C), AT[DLOC, 1];
    goto[X], 1 ← LDF[rnext, 3, 1];   * this instr will be at location 20
D2: goto[X], rnext ← (2C), AT[DLOC, 2];
D3: goto[X], rnext ← (3C), AT[DLOC, 3];
D4: goto[X], rnext ← (4C), AT[DLOC, 4];
D5: goto[X], rnext ← (5C), AT[DLOC, 5];
D6: goto[X], rnext ← (6C), AT[DLOC, 6];
D7: goto[X], rnext ← (7C), AT[DLOC, 7];
D10: goto[X], rnext ← (10C), AT[DLOC, 10];
D11: goto[X], rnext ← (11C), AT[DLOC, 11];
D12: goto[X], rnext ← (12C), AT[DLOC, 12];
D13: goto[X], rnext ← (13C), AT[DLOC, 13];
D14: goto[X], rnext ← (14C), AT[DLOC, 14];
D15: goto[X], rnext ← (15C), AT[DLOC, 15];
D16: goto[X], rnext ← (16C), AT[DLOC, 16];
D17: goto[X], rnext ← (17C), AT[DLOC, 17];

X:  T ← (rnext);                    * instructions don't have to be consecutive
```

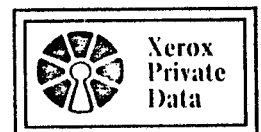
Caution: Hardware execution of a DISPATCHII requires that all four of the low address bits come from APC. This requires all tables to be on 16 word boundaries, no matter how small your dispatch table is. Failure to follow this will result in truly bizarre execution.

3.5. Changing Pages

As noted above, a microinstruction does not know what page it is on, and can only jump to addresses on its current page. There are ways to circumvent this at assembly-time and at run-time.

Assembly-time

ONPAGE[xx] - directs assembler to put this instruction on page xx.



AT[nn] - the assembler assumes that it has been given a 12 bit address, and puts this instruction on the page indicated by the top four bits, and the offset of the last eight bits.

Run-time

LOADPAGE[n] - this is to be done before *every* branch that will be on a different page. This includes GOTO, DBLGOTO, CALL, DISPATCH.

3.6. Notify

When one wants to jump to a specific location in a specific task, APCTASK&APC are loaded with the desired information, and a RETURN is executed.

```
rtemp ← (20C);
rtemp ← (rtemp) OR (160000C);
APCTASK&APC ← (rtemp);
RETURN;
```

* set up the location you want to get to
* OR in the task number = 16

L1 : T ← (0C), AT[20];

After execution of the first block of code, control will be transferred to L1 with task 16 active.

4. Special Functions

The F-field decodes are as follows:

CODE	F1	F2	GROUP B
00	BBFA	REGSHIFT	unused
01	RS232←	STKP←	RESETERRORES
02	LOADTIMER	FREEZERESULT	INCM PANEL
03	ADDTOTIMER	STACKSHIFT	CLEARMPANEL
04	unused	CYCLECONTROL←	GENSRCLOCK
05	LOADPAGE	SB←	RESETWDT
06	unused	DB←	BOOT
07	GROUP B	NEWINST	SETFault
10	no-op	BRANCHSHIFT	APC&APCTASK←
11	WFA	SALUF←	RESTORE
12	BBFB	no-op	RESETFAULT
13	WFB	MNBR←	USECTASK
14	RF	PCF←	WRITECS0&2
15	BBFBX	RESETMEMERRS	WRITECS1
16	NEXTINST	USECOUTASCIN	READCS
17	NEXTOP (NEXTDATA)	PRINTER←	D0OFF

Functions can be best be explained by division into categories. Following each group name will be the pages in the D0 Functional Specification where more information is available. The only function groups expected to be of concern to the programmer are: Useful, ALU, and Sneaky. The rest of the functions should remain unused by most code. In addition, registers used for the Mesa emulator and BitBLT should be avoided.

Useful functions:

APC&APCTASK← - used to directly load this register from ALUA. This is the recommended way to do a *notify* of a different task at a different location. It is usually followed by a RETURN.

USECTASK - forces the next instruction to be taken from the current task. This usually precedes a RETURN, and prohibits task switching.

LOADPAGE - uses F2 for an argument. This statement should precede all CALLs or GOTOs which reference a different page.



ALU functions (p. 17-18):

- FREEZERESULT - inhibits loading of RESULT register. This is used to save the output of the ALU from one instruction to the next.
 USECOUTASCIN - use carry out as carry in.
 SALUF+ - can expand the ALU to its full capabilities.

Sneaky functions: THESE GET SET WITHOUT YOUR EXPLICIT KNOWLEDGE!!!!!!

- REGSHIFT - Set by accessing certain R registers (PRINTER, DB, SB, MNBR) and invoking BBFB.
 STACKSHIFT - Set when STACK&+2, +3, -2, -3 are used
 BRANCHSHIFT - Set by certain branch conditions: MPCARRY, NOOVF, BPCCHK, ALU=<0.

BitBLT (p. 22):

- BBFA - sets up 3 bit dispatch based on SB, DB, MNBR for the next instruction
 BBFB - update of the x level from MW. SETS REGSHIFT!
 BBFBX - update of the main level from MW
 SB+ - loaded from A bus
 DB+ - loaded from A bus
 MNBR+ - loaded from A bus

Mesa (p. 20-22):

- STKP+ - loaded from A bus
 WFA - Mesa Write Field
 WFB - same
 RF - Mesa Read Field controlling the cycler/masker directly from field descriptor.
 NEXTINST - used for Mesa instructions
 NEXTOP, NEXTDATA - used for Mesa instructions
 PCF+ - loaded from A bus
 CYCLECONTROL+ - loaded from A bus. This is a way for controlling the cycler/masker. It also loads SBX[0:5], DBX[0:1].

Auxiliary Registers (p. 27-30):

- RS232+ - from B bus
 PRINTER+ - loaded from A bus

Modification of Control Store (p. 39-40):

- WRITECS0&2 - preceded by APCTASK&APC + x, where x is the address you want to write in. Word 0 is written from A bus, word 2 from B bus.
 WRITECS1 - item on A bus written to word 1 in location in APCTASK&APC

Note: when writing an instruction into the control store, it is the responsibility of the programmer to write proper parity.

- READCS - reads a word from the control store. For the values 0, 1, 3 of T, you get word 0, 1, and 2 from control store into CSData. Word 2 comes back in bits 0:3 of the word, with bits 4:20 coming from word 1

Timer functions (p. 30-33):

- LOADTIMER - loads a timer from ALUA; bits[0:3] state, [4:11] data, [12:15] slot
 ADDTOTIMER - increments a timer from ALUA

Maintenance panel:

- CLEARMPANEL - clears the maintenance panel
 INCMMPANEL - increments the maintenance panel

System functions:

- RESETERRORS (p.38) - clears the freeze on CIA, RESULT and resets PARITY and the fault logic
 RESETMEMERRS - clears the memory error logic
 RESETFAULT - resets the fault logic
 RESTORE (p.38) - loads RESULT register from I12, loads APCTASK&APC from ALUA. This is to restore the machine state after a FAULT.
 GENSRLOCK - clock out bits to the IO controllers
 RESETWDT (p.27) - reset Watchdog timer
 BOOT - initiate a software boot
 SETFAULT - cause a fault to occur



5. Memory and I/O

5.1. General comments

This section is an amplification of the D0 Functional Specifications of January 16, 1978. Its purpose is to provide an interim guide to the proper use of memory and IO operations. You should assume that any topic not covered here is considered correct in the manual. If you follow the guidelines listed, you should not run into any trouble. Ignoring them will get you into funny situations whose symptom is that the data is not where it should be when you think it should be there. The three main topics to be discussed here will be quadword alignment, bypassing, and the memory interlock feature.

Words and phrases in italics are meant to convey a special meaning. If one wanted to avoid trouble, verbs should be read as "must".

5.2. Comments on style

Since the memory operates in parallel with the processor, there are certain hardware features which will prevent you from accessing a location which is an operand in the memory operation which is running concurrently, if these features are used correctly. When an instruction following a memory operation attempts to use data from that operation, the instruction aborts (this means that time freezes until the operation is complete). Efficient microcoders will not write code in this manner, but will use the cycles between a memory operation and use of the data for other necessary code. A forthcoming section will list the maximum execution time for memory and IO instructions.

5.3. Quadword alignment

Memory operations dealing with transference of more than one word *should* adhere to double- or quadword alignment. The memory instruction has two fields involving R registers: the base register field and the SRC/DEST field. The base registers *should* be an even and odd word pair. The even word is the page and displacement, and the odd word contains the upper bits of the virtual address.

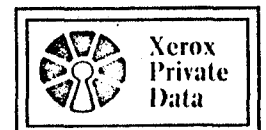
Caution: If the base pointer is denoted BP[0:23], bits 0:7 in the odd register hold BP[0:7], and bits 8:15 hold the BP[0:7]+1. This is incorrect in the manual.

Caution: Section 5.5 hints that bits 0,1 and 8,9 of the base register can get you into trouble if they are not set properly. For safe use, these four bits should be explicitly set to 0 by the programmer.

In general, you *should* always have aligned registers and memory. You can transfer data to a non-quad (or double) aligned R register, but it will defeat the interlock(see below). If the memory is not quad or double aligned, the memory will pick the smallest outer bound matching your request and transfer those words to you. For example, executing a PFetch4 with a memory address of 2 will not give you words 2,3,4, and 5. It will give you 0,1,2, and 3.

Caution: Do not use register 0 in any task block for a SRC/DEST. This forces use of the stack.

Note: The signal QUADOVF is generated only in the following situation: The stack is used for a PFetch2 or PStore2 with memory address equal to 3 mod 4. This is not a general signal which occurs whenever you cross a quadword boundary.



5.4. Bypassing

A non-memory instruction is broken into four cycles: cycle 0 reads the R registers or T, cycles 1 and 2 are taken by the ALU operation, and cycle 3 writes R or T. Since another instruction begins at the beginning of cycle 2, data needed for this instruction will not have been written when the read occurs. The hardware notices this, grabs the needed data for the current instruction, and does the write during cycle 1. The bypass is only good for the following instruction. Bypassing only allows data to be used from one instruction to the next; it does not imply storing.

If the instruction following a store is a memory instruction, the write will be delayed for another two cycles. This means that the store will not take place until cycle 1 of the instruction following the memory operation. As an example, consider a sequence of three instructions, the middle one being the memory operation. A memory instruction reads R registers in cycle 0 for bits 8:23 of the virtual address, and in cycle 1 for the upper bits. Since the R memory cannot be read and written in the same cycle, the second read required by the memory operation forces the write of instruction 1 to occur in cycle 1 of the third instruction. The bypass of data from instruction 1 to instruction 2 will work, and give data to the memory operation for its cycle 0 read, but not its cycle 1 read. This is why you can load an even base register before a memory operation, but not an odd base register.

```

Read          Write - can't take place because of read for memory
/-----/-----/-----/-----/
          Read  Read
          /-----/
                Read Write - from above is done here
                /-----/-----/-----/

```

5.5. Memory Interlock

The memory interlock is provided to protect you from accessing data which may not have been operated on by a preceding memory operation. Use of quad or double word aligned registers will make this work smoothly; nonaligned registers defeat the interlock. The actual R register address is compared (with appropriate low order bits omitted if the operation is double or quad) with R addresses in MC1 and MC2, and the instruction is aborted until the memory is finished. If you like to gamble, you can use non-aligned registers and access those protected by the interlock in the next instruction, but wait until some time later to access the other registers.

Some people have fallen victim to a few bizarre occurrences, and with memory timings, there can be a lot of bizarre occurrences. If you have a problem with the memory, check your code with some of the examples below, particularly the "Gotcha" section. If you find another example of something which doesn't work, please send it to me.

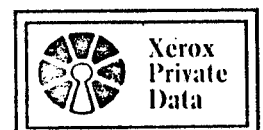
Examples

Proper use of the memory will look like the following:

```

RV[rbaseEven, 10];
RV[rbaseOdd, 11];
RV[rsrc2, 12];
RV[rsrc4, 14];

```



```
RV[rbaseEven2, 20];
RV[rbaseOdd2, 21];
RV[rtmp];
```

1. rbaseOdd ← value1;
 rbaseEven ← value2;
 PFetch2[rbaseEven, rsrc2];
 t ← rsrc2;
 * set up Odd register first
 * set up even register - bypass will get proper
 * value to mem op in cycle 0
 * when memory done, this instr will be executed
2. rbaseOdd ← value1;
 rbaseEven ← value2;
 PFetch4[rbaseEven, rsrc4];
 t ← rsrc4;
3. rbaseOdd ← value1;
 rbaseEven ← value2;
 PFetch2[rbaseEven, rsrc2];
 rbaseOdd2 ← value3;
 rtmp ← t;
 PFetch2[rbaseEven2, rsrc2];
 * this could also be rbaseEven2 ← mumble
 * need this to be sure store is accomplished

Improper use of memory:

1. rbaseEven ← value2;
 rbaseOdd ← value1;
 PFetch2[rbaseEven, rsrc2];
 * mem op needs this in cycle 1, but bypass
 * only works for cycle 0
2. rsrc2 ← value3;
 PStore2[rbaseEven, rsrc2];
 * set up a register to be stored
 * rsrc2 will not have been written when this
 * begins (note 1, page 47)
- 3a. rtmp ← value3;
 MemOp[rbaseEven, rsrc2];
 t ← rtmp;
 * this will not work because of the bypassing
 * mentioned above. Writing of rtmp is in cycle
 * 1 of this instruction
- 3b. t ← value3;
 MemOp[rbaseEven, rsrc2];
 rtmp ← t;
 * same reasons as 3a.

Gotchas:

1. When you do a PFetch or PStore, and you are using a register which is out of your 16 per task allotment, you are likely to be writing into the wrong register. If you use 16 registers, your RSEL field in the microinstruction will contain only 4 bits. As you may recall from R addressing, the top two bits of the 6-bit RSEL field are *conditionally* ORed with your task number. When doing a PFetch or PStore, the task bits are *unconditionally* ORed with your task number, which may or may not change the R address.

2. PFetch1[rbaseEven, rsrc2];

 LOADTIMER[rsrc2];
 * this will defeat the interlock!!!!!!

This bug/feature is very subtle. Unfortunately, Micro decodes this instruction in such a way that the register rsrc2 is not considered a source and therefore the interlock is not checked. This will happen to ALL special functions which load a register from ALUA, and therefore includes APCTASK&APC←. BEWARE!! The way to get around the above, is to say:

```
lu←rsrc2, LOADTIMER;
```

Now rsrc2 will be checked.



6. Getting Started

Most of the information which you will need will be present on Iris, and eventually Isis. We have a directory called D0. This is the first place you should go and look for any programs or documentation that you need. There is also a microcoder's distribution list which is on [maxc]<secretary>d0users.dl. You will receive notification of new programs or updates via this distribution list. Send a message to Jeannette Jenkins in CSL to get on this list.

There are two files on [Iris]<D0> which can be used to create a microcoder's disk. If you have a virgin disk, you should obtain a copy of <alto>newdisk.cm from your local file server. After running this, get [Iris]<D0>newmidasdisk.cm for a Midas disk or <D0>newsimdisk.cm for a simulator disk. Either command file will give you all the files you need to use for microcoding. For a disk already containing an operating system, FTP, Chat, Bravo, and other basic programs, you need to run [Iris]<D0>midasdisk.cm or <D0>simdisk.cm. This will provide you with enough Mesa to run Midas or the simulator and all needed microcode files.

The first document to be read is the D0 Processor Functional Specification, January 16, 1978. This explains the hardware and also gives you pictures of the architecture which are useful to look at while coding. After reading this, you should look at the D0 MicroAssembler manual (which is in this guide) to familiarize yourself with the microcode syntax. After this, you should be able to write a simple program.

Given that you've now written a program, you need to assemble it. Actual assembly is accomplished by two programs: Micro and MicroD ([Iris]<D0>micro.run, microd.run). Micro is the main assembler; MicroD's function is instruction placement in the microstore. Micro is a very general microcode assembler, and it accepts language features from a file called D0lang.mc ([Iris]<D0>D0lang.mc). This file is assembled with each of your microcode files. If your file is named Test, you would assemble it in the following manner:

```
Micro D0lang Test
```

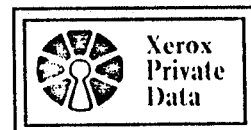
Assuming you got no errors, you would then proceed with

```
MicroD Test
```

At this time you have a file called Test.mb which is ready for loading into the D0 or for use with the simulator.

It is possible to check out your code without a D0. A D0 Simulator ([Iris]<D0>s.bcd) exists, and is very useful for code which does not use and IO routines. There is a very readable document on how to use the simulator in this manual. The simulator closely tracks the D0 and any changes made to it. Once your program runs through the simulator, you can be very confident that it will work on the D0. The simulator has a feature for running in non-overlap mode which is most useful for debugging.

On the D0, microcode programs will be run and debugged with a program called Midas. Midas has its own documentation in this manual. The Midas system is in the form of a "dump" file and is on [Iris]<D0>midasrun.dm. If most of the information in this section is new to you, don't bother getting into Midas yet.



Caution: NEVER get Midas unless it is in a dump file. Midas and its auxiliary files are quite dependent.

7. Caveats

You must execute a TASK function every 12 microinstructions to insure that data from higher priority devices is not lost. A TASK clause in a microinstruction is a cheap trick to execute a CALL and a RETURN, since RETURNS are the only way a higher priority task can gain control.

NEVER use more than sixteen R registers for a given task.

If you are writing microcode which will be incorporated into a release, you must "check out" a prefix from the person in charge of D0 microcode releases (currently Carol Hankins). This prefix will occur before your labels and register names.

Anyone who does not follow the above rules will receive no help from me whatsoever.

8. Suggested Programming Style

It is highly unlikely that you will be the only person reading your code, so below are some suggestions which will make your fellow coder's life easier.

As mentioned in the Caveat section, if this piece of code will ever be in a microcode release, you must check out a prefix from me. This prefix is to be used in front of *all* R register names and labels. Given that they all begin with this prefix, they can still be named something which suggests their function. It is possible to define many names for a particular R register (by executing as many RV's as are necessary), and if your code can be sectioned in a reasonable manner, you may want to try this technique.

If you use names which are a concatenation of two or more syllables, you might consider using lower case letters and having the next syllable begin with upper case. This produces quite readable text. If you use lower case, you must call the micro-assembler, Micro, with the "/u" switch on the command line.

Micro, makes it quite easy to define constants which assign English-like names to arbitrary sets of bits. There are two facilities for accomplishing this. The macro MC[name, number] defines a constant; i.e., every time the assembler finds "name", it substitutes the number appended with a "C". SET[name, number] will give you the number without a "C" which is suitable for use as a parameter.

Examples:

```
MC[bitMask, 200];      * used for expressions like rtemp+(rtemp) OR (bitMask);
MC[sectorMask, 16400]; * T+(DiskAddr) AND (sectorMask);

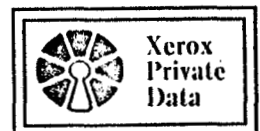
SET[myPage, 3];        * use this as a parameter as in ONPAGE[myPage];
SET[dispLoc, 200];     * DISP[dispLoc];
```

Note: It should be noted that constants formed in this manner must still adhere to the eight bit limit discussed in the section on constants.



It is suggested that you begin each of your modules with a SETTASK and an ONPAGE. Parameterization will make these easy to change later on. You should also begin your modules with a "notify" to get you to the proper task and location for the start of your code.

In addition to writing readable code, you should also try to pack as much in any given microinstruction as is possible.



9. Sample Programs

The following code is a template of what microcode files should look like. It is filed on [iris] < d0 > template.mc, and you are welcome to retrieve it. It is assumed that you will insert your code via Bravo. Micro knows about Bravo trailers, so you may format your code as much as you would like.

```

builtin[sampleInsert, 24];          * you can do this if you want to insert a file
sampleInsert[d0lang];             * like D0lang of a defs module so you won't have
                                   * to type it on the command line to Micro

title[template];                  * give it a reasonable title.

%
% Next put in some comments as to what this code does. If it is a long comment, you can
% enclose it between two percent signs. You might also want to include any assumptions
% that are necessary for this code to work.
%

* your local constants
mc[sectorLate, 4000];              * makes a symbolic constant
mc[resetEverything, 13];           * another
mc[firstLoc, 20];                  * first location you want to go to

* your parameters
set[myTask, 4];                    * see use in settask statement below
set[myPage, 13];
set[dispLoc0, OR@[LSHIFT[myPage, 10], 0]]; * a handy way to parameterize dispatch tables
set[dispLoc1, OR@[LSHIFT[myPage, 10], 20]]; * so that if your page changes you don't have to
                                           * manually change those locations

settask[myTask];                  * used to allocate proper R registers - does NOT make you
                                   * run in that task. You have to do a notify. See below

* register declarations
rv[rtemp, 0];                      * you can force use of a particular register this way
rv[count, 1];                      * actually you don't need "1", regs are assigned in order
rv[baseEven, 2];                   * a good way to set up base register for memory ops
rv[baseOdd, 3];                    * other half
rv[word0];                          * a quad-word buffer beginning at R register 4
rv[word1];
rv[word2];
rv[word3];

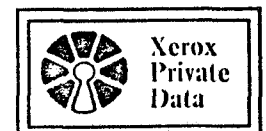
* ready to begin your code
onpage[myPage];                    * directs assembler where to put it

* bootstrap yourself up to your task and load your TPC
init: rtemp ← myTask;
    rtemp ← lsh[rtemp, 14]; * put it in the task field for loading APCTASK&APC
    t ← firstLoc;
    rtemp ← (rtemp) OR t;
    apctask&apc ← rtemp;      * do the notify
    return;

actualStart: t ← rtemp, at[firstLoc]; * you are now running in your task

end.                                * that's all

```



The file, [Iris] < D0 > sample.mc, consists of sample programs, which are each prefaced with what I hope they will illustrate. The sections can each be broken out (code between TITLE and END) and be assembled and run through the Simulator, if you wish.

TITLE[Sample1];

* This code takes the number in R register RNum and multiplies
 * it by 10. This is accomplished by multiplying it first by
 * 8, multiplying a copy of it by 2, and adding the results.

RV[RNum];
 RV[RTemp]; *just a temporary register

```
INIT:  RNum ← (4C);       *initialize it
START: T ← RNum;       *need to copy it into T to get it to RTemp
      RTemp ← T;       *RTemp = RNum
      RTemp ← LSH[RTemp, 3];   *RTemp = 8*RNum
      RNum ← LSH[RNum, 1];   *RNum = 2*RNum
      T ← RTemp;       *put in T so we can add them
      RNum ← (RNum) + T;
      GOTO[START];
```

END.

%

Now we try and make the above a bit more efficient.

%

TITLE[Sample2];

RV[RNum];
 RV[RTemp];

```
INIT:  T ← RNum ← (4C);   *loading T is free
TIMES10: RTemp ← T;
      T ← RTemp ← LSH[RTemp, 3];
      RNum ← (LSH[RNum, 1]) + T;   *shifting is on A-bus
      GOTO[TIMES10];
```

END.

%

Moving right along, let's look at branching. The important things to remember about branching are that ALU conditions are available at t3 (after cycle 2) and are saved, while R conditions are available at t1, and are destroyed after this time.

In the next program, we're going to use two subroutines. GETVAL is totally mythical - assume it gets a number from somewhere and puts it in T. TIMES10 is the above code made into a subroutine. The following program reads a count via GETVAL, then calls GETVAL to give it numbers which it makes positive if they aren't, and then multiplies them by 10. When finished with that loop, it goes back up to get another count.

%

TITLE[Sample3];

RV[RNum];
 RV[RCount];
 RV[RTemp];

```
START: CALL[GETVAL];
      RCount ← T;
      GOTO[DONE, R < 0], LU ← RCount;   *way to put something on bus
* could have tested on T above via
* GOTO[DONE, ALU < 0];
```

AGAIN: NOP; *see below for explanation



```

CALL[GETVAL];
RNum ← T;
* again, could have tested on T as above
GOTO[MULR, R >=0], LU ← RNum; *if it's positive, jump
RNum ← (RNum) XOR (100000C); *make it positive
MULR: CALL[TIMES10];
RCount ← (RCount) - (1C); *decrement count    DBLGOTO[AGAIN, DONE, ALU#0];
DONE: GOTO[START];

TIMES10: T ← RNum;
RTemp ← T;
T ← RTemp ← LSH[RTemp, 3];
RETURN, RNum ← (LSH[RNum, 1]) + T;

```

END.

%

Many errors can be avoided by understanding the branching logic. CALL's always have to be at even locations. DBLBRANCH and DBLGOTO go to odd locations if true, and even if false. The DBLGOTO which is right before the label DONE is supposed to go to AGAIN if true, and DONE if false. At AGAIN, we really want to do a CALL[GETVAL], but since the branching logic dictates that AGAIN be placed at an odd location, we have to put in a NOP.

%

10. Common Error Messages

Micro occasionally produces rather baroque error messages. For a complete list, see the Micro documentation. The following are the ones most commonly received when beginning:

RREGISTER+B Undefined - a missing set of parentheses around the "A" field of the the ALU function in the instruction. This comes from a statement like $T \leftarrow RTEMP + (1C)$, where the above message would be $RTEMP+B$ Undefined.

Field RSEL2 already used - this usually results from referring to two R registers in the same statement. There is only space for one in the micro-instruction. $RTEMP \leftarrow (RADDR) + (T)$ is illegal.

Illegal constant - a constant in a microinstruction can only be 8 bits, either the upper or lower 8. If you need a constant which is longer you need to do it in two instructions.

T+B Undefined - you are trying to put two things on the B bus. Look at the diagram of the D0. An instruction of the form $T \leftarrow T + (377C)$ is not possible, since T is on the B bus, and so is the constant.

Field BS already set - Bsel is 0 or 1 for a constant, and 3 for the cyclor/masker. Thus, $RTEMP \leftarrow RSH[RTEMP, 1] \text{ AND } (2C)$ would produce this message. This statement also produces "Fl.uscd.twice".

MicroD is the part of the assembler which places the instructions in their final locations. Any messages received from MicroD are because of placement constraints. The following are the most common:

Attempted to link LabelX with LabelY - you probably have two DBLGOTOs which require LabelX or LabelY to be on an even location for one and an odd location for the other:

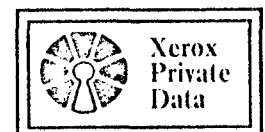
```

e.g. DBLTOGO[LabelX, LabelY, alu#0];
     DBLGOTO[LabelX, LabelZ, alu >=0];

```

Impossible allocation constraints - Most likely there are two CALLs in sequence.

A printout of all locations on two pages - This probably results from doing a GOTO/CALL to a different page not being preceded by a LOADPAGE.



MIDAS MANUAL

29 December 1977

by

Edward R. Fiala

edited by

Brian Rosen

Xerox Business Systems
Systems Development Department
3408 Hillview Road
Palo Alto, California 94304



1. Midas

Midas is a loader/debugger with versions existing for the Maxc2, Dorado, D0, and M68 microprocessors. Midas runs on an Alto, controlling the target machine remotely. It is used for loading/dumping microprograms assembled by Micro, for examining and modifying storage and control signals, and for testing the hardware in an assortment of ways.

Midas is about 90% coded in Bcpl and 10% in assembly language. The Maxc2 version was implemented by E. R. Fiala and H. E. Sturgis. The Dorado, D0, and M68 versions consist of machine-independent modules implemented by E. Fiala (ISF and Overlay packages implemented by L. Deutsch and Alto microcode by E. Taft are also used) and machine-dependent sections implemented by E. Fiala for Dorado; D. Swinehart and P. Baudelaire for M68; D. Charnley, C. Thacker, B. Rosen, and C. Hankins for D0.

An internal description of Midas is available to anyone interested in adapting Midas to a new hardware system (see Maxc1 < D1Docs > MidasInternal.Press).

2. Starting Midas

To start Midas, simply say "Midas" to the executive or, more generally, "Midas com-file". The following ways of starting Midas are of particular interest:

midas/i	initializes (required when any Midas files move or change);
midas	simply fires up Midas;
midas debug	starts Midas and immediately reads commands from the "Debug.Midas" command file

"Midas debug < cr >" to the executive is equivalent to bugging the "debug" item in the submenu put up by the "Run-Prog" command.

Midas command files have the extension ".Midas". Generally, there is one command file for each hardware diagnostic, with the same name as the diagnostic, e.g.:

dgbasic.mb	the diagnostic;
dgbasic.midas	the command file.

A command file following this convention loads the diagnostic into the microprocessor and displays various registers of interest when the microprogram is in use. Assorted command files currently in use are discussed later, in the section about "Run-Prog" and "Read-Cmds".

3. Midas Display

At the top of the Midas display are a number of *name-value menus*. Below these are the name of the last microprogram loaded, two *command comment lines*, the *command menu*, and the *input text line*. When you move the mouse over a name-value menu or the command menu, the menu item selected (if any) turns black.

Note that mouse actions execute when you RELEASE the mouse button, so you can move the mouse with the button depressed without causing damage. If the mouse is no longer over the selected menu item, nothing happens when the button is released.

A name-value menu may contain a *register* or *memory address* in the name area and its contents in the value area. A memory address may be specified as the memory name and word number, or as the name of an address symbol defined in a microprogram you have loaded. The address symbol may be followed by +/- displacement.

Name-value areas are of *different sizes*. Smaller areas on the left are already filled in when you fire-up Midas. You can clear these or replace them with other stuff from your microprogram, if you want to.

Initially, some name-value menus are empty. The largest areas on the right should be used for items with long names or values. If the item you display overflows the selected area, the right-most characters of the name get truncated, then the left-most characters of the value.

To display a new item, type its name (which will appear on the input text line), move the mouse over the name field in a name-value menu, and push-and-release the *left* (or *top*) mouse button. Memory addresses in your microprogram may optionally be followed by a displacement "+n" or "-n". "n" is the same as "+n".

If the command line is empty, the selected menu will be cleared when the button is released.

When you push the *right* (bottom) mouse button over a name field in which an address is displayed, a *subsidiary menu* appears as follows:

A+1 A-1

"A+1" increments the address, displaying the next location. "A-1" decrements the address.

Releasing the *middle button* over an address item shows an *alternate printout* (if any) on the command comment lines. If the input text line is non-empty, it will first display that item.

Releasing the *left button* over a value item, *evaluates* the input text and stores the value (or 0 if no text typed) in the selected register. The input text may consist of octal numbers or memory addresses +/- offsets. For memories and registers whose values are displayed as several fields, the input text must also be divided into fields. Blanks must be inserted where Midas prints blanks in the displayed value of the register.

Midas also provides for *special input evaluation* based upon the register or memory into which the value is to be stored. At the present time, an alternate input routine is implemented for registers and memories that contain microinstructions (MIR, IM, and IMX on Dorado). These are discussed later.

Releasing the middle button over a value item shows an *alternate printout* of the value on the command comment lines. The alternate for registers that normally hold IM addresses is the nearest IM address tag less-equal to the value+offset (The value is also put on the input text line, so you can examine that IM location in some name-value menu, if you want to.). Registers and

memories that contain microinstructions may also be printed out symbolically. These are discussed in more detail later.

Releasing the *right button* over a value item appends the text of the value to the input text line. This is primarily used in command files to move values from one register to another.

Summary:

	<i>Name-field</i>	<i>Value-field</i>
<i>Left Button</i>	Examine	Change value
<i>Middle Button</i>	Alternate printout	Alternate printout
<i>Right Button</i>	A+1, A-1	Append value to input line

4. Midas Command Menu

The command menu contains a list of commands or *actions* that Midas can execute. The basic menu is modified under some conditions. For example, the "Dump" menu item only appears after you have done a "Load". During execution, some commands replace the command menu by sub-menus.

For the command menu, all mouse buttons are presently equivalent (On Dorado, "Ck-DMux" and "No-ck-DMux" actions are exceptions). Many common actions may alternatively be initiated by *keyboard command characters*, as given in the action table below.

General philosophy on mixing keyboard and mouse button control is that, when possible, a command involving some typing is carried out completely at the keyboard, whereas commands involving mouse buttons are carried out completely with the mouse.

For example, to start a microprogram at some address, you normally have to type an address; then you could bug the "Go" item in the command menu, but normally it is more convenient to type "address;G" because you won't have to lift your hand from the keyboard; ";G" are the command characters equivalent to bugging "Go".

Many commands are executed in overlays. When these get executed, the register display will turn off (The code for overlays resides where the display bit buffers would otherwise be.). During loading or execution of command files, the display is turned off to make the machine run faster.

Long-running commands normally display an "Abort" menu item. When this is bugged or when control-C is typed, the action terminates.

Input	Char	Menu Item	Comments
<i>Actions (potentially) available on all implementations of Midas</i>			
[File]		Read-Cmds	Executes command file (def. ext. ".Midas") on input text line or from submenu
File		Show-Cmds	Shows command file text for selected menu items
		Write-Cmds	Write subsequent commands on file
		Run-Prog	Run microprogram selected from submenu (restricted use in command files)
Files	:L	Load	Loads .MB files
Files		LoadSyms	Loads only addresses from .MB files
[File]	:D	Dump	# Dumps compacted .MB file using the .MB file(s) of the previous load to control what's dumped
[File]	:C	Compare	# Compares microprocessor data to data specified in .MB file--compare file must not have fixups for forward references
Addr	=		Prints value of an address (illegal in com-file)
IMaddr	:B	Break	Inserts break point (The breakpoint occurs after the instruction containing the break has been executed.)
[IMaddr]	:K	UnBreak	Removes break at address (last break if nothing typed)
[IMaddr]	:G	Go	* Start at address (continue if nothing typed)
[IMaddr]	:P	Go	* Same as :G (more mnemonic when you mean "proceed")
[IMaddr]	:	Step	Single-step at address (continue-step if nothing typed)
		Reset	Reset or cold-start the machine. Assorted options are controlled by a subsidiary menu.
@		Test	* Test register or memory (see below)
		Test-All	Test everything (see below)
@IMaddr	:R	Rep-Go	* Go at address, repeat endlessly after halts
@IMaddr	:S	Rep-SS	* Repeatedly step at address
@		PEscan	Scans local memories (IM, IFUM, RM, T on Dorado) for parity errors
@		Field-loop	* For scoping (see below)
@LDRaddr		LDR-loop	* (see below)
		Virtual	Changes IM address interpretation to be virtual
		Absolute	Changes IM address interpretation to be absolute

@Not available in D0 Midas as of this date

Actions available only on the Dorado Midas

		Set-Clock	Set the clock speed to the value selected from a submenu
		t1	Clocks MIR through t1, reads the DMux, then clocks through t2 and restores MIR (so display shows DMux values read after t1).
		t2	Clocks MIR through t2, reads the DMux, and restores MIR (so display shows DMux values read after t2).
		t3	Clocks MIR through t3, reads the DMux, then clocks through t4 and restores MIR (so display shows DMux values read after t3).
		Rep-t2	Repeatedly does t2
		Ck-DMux	Left-button causes the DMux consistency checker to be called after Go, Step, t1, t2, and t3. Middle and right mouse buttons modify display of DMux items as discussed later.
[IMaddr]		SimGo	* Like "Go" invoking the DMux checker after each step.
		SimTest	* Random instruction test using MIR and the DMux checker.
		Passive	Prevents Midas from disturbing the hardware while running
		Active	Puts Midas into normal mode
		Update	Read registers and display new values (used while passive)

Actions available only on the D0 Midas

[File]		Boot	Boot D0 and load [File] as kernel. KERNEL.MB used if [File] is defaulted
--------	--	------	---

* = requires preceding "TimeOut" command in com-file
 # = requires confirmation with <cr>, "Y", or "." (or by preceding "Confirm" command in com-file)
 [...] = optional input text

Some actions in the preceding table are replaced with complementary actions after execution. These are Show-Cmds by Conceal-Cmds, Write-Cmds by Stop-Write-Cmds, Ck-DMux by No-ck-DMux, Passive by Active.

5. Keyboard

Some characters which are symbol constituents in microprograms will cause trouble for Midas if they appear in address symbols.

Lower case typein is converted to upper case by Midas, so avoid lower case characters in microprogram addresses. You should write microprograms with the shift-lock key depressed or assemble them with the convert-to-upper-case assembly switch.

Avoid "=".

"+" and "-" are ok so long as the following character (if any) is a letter, but you should avoid these generally.

Typing ahead is legal until the character you type would cause execution of an action. After that, Midas will flush input and blink at you until the current action finishes.

At the end of an action, input text typed for that action is displayed on the input text line. This text remains valid and can be used as the arg for another mouse action. However, if you type any character (except control-A or backspace), the old input will be flushed before inserting the new character.

Keyboard editing characters are as follows:

control-A	delete last character
backspace	delete last character
control-Q	clear text line
del	clear text line

Other special keyboard characters are as follows:

control-C	abort the current action--equivalent to bugging the "Abort" command
control-Z	abort a command file
escape	repeat previous action (special for "Test" and "TestAll")
control-D	turns on the display (used during command files)
control-O	turns off the display (used during command files)

The interrupt characters above are ineffective during loading, dumping, or comparing, which typically take between 2 and 20 seconds. Indefinite duration commands, such as "Go", "Test", etc. always monitor the keyboard, so control-C can be used to terminate them. Although control-

C and "Abort" are equivalent, "Abort" is only effective if the Midas main loop sees the mouse button go down then up; in testing big memories or registers that take a long time to read-write, Midas doesn't monitor mouse buttons often--you may have to depress a button 5 or 10 seconds before Midas sees that it is down. For this reason, normally use control-C to abort--even with control-C the abort may be delayed for a few seconds.

Control-Z, control-D, and control-O are intended for use during command files. However, these characters do not take effect until the command file executes a command such as "Go" which monitors the keyboard. There is no way to abort a command file and give control back to Midas safely except during a "Go" or other long-running command. This is not expected to be a problem because commands are executed quickly.

After interrupting a "Go" with control-C or control-Z, proceeding with ";P" or ";G" will succeed except when you have smashed the machine state by doing a "test", "reset", etc. action or have displayed a register that Midas cannot examine non-destructively (e.g., IFUM on Dorado can only be examined destructively).

Although command menu items "Step", "Go", "Break", "UnBreak", "Rep-SS", and "Rep-Go" are provided, the keyboard character equivalent to these is usually more convenient.

6. Command Files

Command files (default extension ".Midas") are normally executed either by typing "Midas filename" to the Executive or by bugging a file name in the subsidiary menus put up by "Run-Prog" or "Read-Cmds". Alternatively, you may type a file name first, then bug "Read-Cmds". ("Run-Prog" does not permit you to type a file name--you are limited to files appearing in the sub-menu.)

"Run-Prog" *resets Midas*, while "Read-Cmds" does not; resetting Midas consists of clearing the symbol table and restoring the display to its initial arrangement.

"Run-Prog" is used to completely change contexts--to run a new microprogram, for example. Selecting a command file from the "Run-Prog" submenu is equivalent to exiting to the Executive and typing "Midas comfilename".

"Read-Cmds" is frequently used to modify the display in various ways.

The file names that appear in the sub-menus for these are contained in the Midas.Programs file.

The command-file facility is actually an (awkward and limited) programming language. The collection of actions discussed below is being developed so that command files can monitor diagnostic microprograms, collect and report error information on an output file, or direct the sequence of diagnostic microprograms according to hardware failures that are observed.

For system microcode, command-files can be used to control auto-restart and failure diagnosis.

Command files can be nested several levels (limited by the size of sysZone which must be big enough to accommodate OpenFile and buffers for the command files already open). However,

there are the following RESTRICTIONS:

- (1) [Maxc2 only] "AltIO" terminates command files (i.e., upon return to Midas from AltIO the command file will not be continued).
- (2) "Run-Prog" is illegal except in the *top level* command file. ("Run-Prog" resets Midas, then calls "Read-Cmds". This reset operation smashes the symbol table, the display, and the stack back to their initial state. Hence, if you were to execute "Run-Prog" from a subsidiary command file, that command file would be continued, but the higher level ones would not, and the sysZone buffers for the higher level command files would not be released.)

Since Midas builds a table of file FP's during its initialization, when you edit a command file or .MB file, you should reinitialize Midas by typing "Midas/I". When you add new command files or .MB files you should update the "Midas.Programs" file appropriately and do "Midas/I". The form of "Midas.Programs" is discussed later.

A number of commands that can never occur when Midas is run interactively are useful in command files. These, not given in the table earlier, are as follows:

Text Arg	Action	Comments
Value	SkipVEql	Skip the next command if the input text evaluates equal to the contents of the register or memory word displayed. The input text is evaluated exactly as though it were to be stored into the register displayed in that name-value menu, so if the value displayed has several fields, the input text must also have several fields.
Value	SkipVGr	Skip the next command if input text evaluates greater than the contents of the item in the name-value menu.
Value	SkipVLs	Skip the next command if input text evaluates less than the contents of the item in the name-value menu.
Octal no.	Skip	Skip N following commands, where N is the value of the input text.
Octal no.	BackSkip	Reset to byte 1 of the command file, then skip.
Octal no.	Return	Return out of current command file, then skip ("Tag" form is presently illegal for this one.).
.Tag	Skip	Skip following commands until one is encountered with the label ".Tag". Command labels are distinguished by beginning with ".".
.Tag	BackSkip	Reset to byte 1 of the command file, then skip.
	DisplayOn	Turn on the display, so that effects of subsequent commands can be observed. The display is initially off for a command file.
	DisplayOff	Turns off the display.
Octal no.	TimeOut	Input text is evaluated to a 32-bit octal number of msec at which to abort the immediately following command, if it has not finished by then. This is intended for use before "Go" and other commands which might hang indefinitely. If the timeout occurs, Midas will skip the command after the "Go". TimeOut also turns on the display, necessary because the machinery which checks for timeout is only active with the display on.
	Confirm	Supplies confirmation for the command which follows (which should be one of the commands requiring confirmation).
File name	OpenOutput	Opens an output file (default extension ".Report") on which text can be written.
	CloseOutput	Closes the output file.
[text]	WriteMessage	Writes the contents of the input text buffer on the output file. Note that if any text follows the WriteMessage, that text up to but not including the <cr> is what gets written. However, if <cr> immediately follows WriteMessage, then the contents of the input text buffer left by the previous command get written. "~" is translated into <cr>.

text ShowError Displays the text arg on the command line, turns on the display if it was off, and queries with "Abort" and "Continue" menu items.

7. Syntax of Command-file Actions

The syntax of a command-file action is as follows:

```
[ "." <tag> <$ " "> ] <buttons> <$ " "> <menu> <" "> [ <$ " "> <text> ] [ ";" <comment> ] <cr>
```

where the "[]" denote that the ".tag", input text, and ";comment" are optional. <\$ " "> denotes a sequence of blanks.

If the first character on the line is a ".", then the characters after that are a label or tag which may be used as the argument for the "Skip" or "BackSkip" actions given in the table earlier.

<buttons> may be any combination of the letters "L" (left-button), "M" (middle-button), and "R" (right-button); these are the buttons released to execute the action. These may appear in any order.

<menu> is the menu name in which the action is executed ("X" for the command menu, "A0"..."A19", "B0"..."B19", and "C0"..."C19" for name-value menus).

<text> is the text typed on the command line, which may be anything except a ";".

Note that if a *single blank* terminates <menu> and if no input text argument is given, then input text left-over from the preceding action will be used. This allows text from a right-button action over a value to be used in a following action (e.g., in WriteMessage or to store the value into another register). However, one or more extra blanks will reset the input text, so the action is executed with null input text.

For registers/memories that contain addresses, the pretty-print procedures (middle-button over value), also print the result on the input text line; this can also be used in subsequent actions.

";" begins a comment, which may be omitted.

<cr> (carriage-return) terminates the action.

To find out what text should be put in command files, you can bug "Show-Cmds" in the command menu. This will cause the command file text for each command to be displayed on the command comment line as the mouse selects it (You don't have to execute the command to see the equivalent text.). This text is complete except that the mouse button which executes the command isn't shown unless you depress the mouse button. To terminate "Show-Cmds", bug "Conceal-Cmds" (which appears only when "Show-Cmds" is in progress.).

You can prepare a command file (default extension ".Midas") by typing a file name and bugging "Write-Cmds". This causes text for subsequent commands to be put on the file. When you are done with this, bug "Stop-Write-Cmds" to close the file. ("Stop-Write-Cmds" is in the command menu only when a command file is being written.).

You will probably want to edit out your goofs with Bravo after the command file is written.

In addition, you will have to insert "Confirm" and "TimeOut" commands into the command file before those actions which require confirmation or which might hang indefinitely (See the table given earlier for the actions that require this.).

Here is a sample command file:

L X Load dgl;	Equivalent to typing "dgl" and bugging "Load" in the command menu
L A0 Addr TASK;	Examine the "TASK" register in name-value menu A0
L A0 Val 0;	Change the value in TASK to 0
L A1 Addr RTEMP;	Examine the address "RTEMP" in menu A1
L A1 SkipVEql FOO+3;	Skip the next command if RTEMP contains the value FOO+3
L X ShowError RTEMP not loaded correctly	
L A2 TLINK 0;	Examine the Link register for task 0 in menu A2
L X TimeOut 2000;	Abort the following command if it hasn't finished in 1.024 sec.
L X Go START;	Begin microprogram execution at address "START"
L X Skip 1;	Skip the next command if "Go" halts before timeout
L X ShowError START;G failed;	Show an error message

8. Loading Programs

Programs are loaded by typing a file name (default extension ".mb") and bugging "Load" in the command menu. However, direct use of "Load" should be rare if you add appropriate command files to Midas.Programs. "Load" loads the entire .mb file--symbols into the Midas symbol table and data into the hardware.

"LoadSyms" loads only the address symbols and virtual memory mapping table from the .mb file. This may be useful when reentering Midas from the Executive without smashing the program stored in the microprocessor.

"LoadData", (in command files but not available interactively), loads only the data blocks from the .mb file. "LoadData" is provided so that, when necessary, a microprogram can be loaded without cluttering the Midas symbol table.

On Dorado, the DMUX and DCHK memories are exceptions--symbols for these are loaded anyway.

The Midas symbol table consists of resident storage for about 700 symbols (i.e., 6 symbol blocks of 2000₈ words each). If your program exceeds this, symbol buffers swap off the disk. (The primary penalty for exceeding resident symbol storage is that breakpoint response will be .15 seconds slower per symbol block on the disk.)

To avoid this problem, don't load one microprogram on top of another--use "Run-prog" to reset Midas, or, if the program you want to load does not exist in the "Run-prog" submenu, do a "Run-Prog" and bug "Loader" in the submenu to reinitialize Midas, then do a "Load".

It is also a good idea to assemble microprograms as a single .MB file. Although Midas can load multiple .MB files (typed as a list separated by commas), this will fragment the symbol table and cause extra thrashing.

These recommendations follow because Midas takes advantage of alphabetical address ordering in .MB files to pack its symbol buffers nearly full. But when subsequent files are loaded the symbol buffers will fragment to about half-full, symbol buffer swapping will result, and symbol searches will be longer.

9. Dump and Compare

Both "Dump" and "Compare" require confirmation by <cr>, Y, or "." They accept the name of a microprogram (default extension ".mb") on the input text line. If the input text line is empty, then the file name is defaulted to the name of the program last loaded.

"Dump" deletes forward reference fixups left by Micro (which never occur on Dorado or D0 because MicroD does these) and compacts both data and addresses to use less disk space and load more quickly later.

Also, if undumped .MB files contain forward references, they cannot be used with "Compare" (no problem on Dorado).

Note that *only memory words loaded by Load are dumped*--you cannot patch unused locations, dump the program, and expect the patches to survive. (Suggestion: assemble extra locations as a patch area with your microprogram, so that you can patch and dump during debugging.)

"Compare" compares data currently in storage against data in the file and reports differences on the Midas.Errors file.

10. Virtual and Absolute Control Store Interpretation

Because the placement transformations performed by MicroD make it difficult to correlate IM locations with positions in microprogram sources, the Dorado and D0 implementations of Midas contain a map to transform addresses produced by Micro into absolute control store locations produced by MicroD.

The general idea is that, if you suspect a hardware problem in the control section, you will work in absolute mode, but in all other situations you will work in virtual mode.

When you fire up Midas, the display is in absolute mode and the "Virtual" command appears in the command menu; when you load a microprogram, the display switches to virtual mode and the "Absolute" command appears in the command menu. You can always tell which mode the display is in because the *opposite* mode appears in the command menu. You can always switch from one mode to the other by bugging "Virtual" or "Absolute", but if you have not loaded any microprogram, then switching to virtual mode will not be useful.

In virtual mode, values in all registers that normally contain control store addresses are translated by Midas into virtual addresses, and the virtual addresses are displayed on the screen.

On Dorado the registers affected by this are CIA, CIAD, TNIA, BNPC, TPC, TLINK, and OLINK.

On the D0, the affected registers are CIA, TPC and CALLER.

When a memory or register containing a control store has a value outside the VM, it prints as 7777. To find the absolute value in this case, you have to switch to "Absolute" mode.

Midas defines two memory names for the control store, IM and IMX. IM is addressed by virtual addresses, and only locations assembled by your microprogram have meaning in the virtual memory.

In other words, if your microprogram is 10 words long, the meaningful part of virtual memory is only 10 words long. In this case, if you examine virtual addresses greater than 7, the printout will show an absolute address of 7777 and a meaningless number for the rest of the value.

If you wish, one of these meaningless virtual locations can be added to the virtual memory (i.e., made meaningful) by storing a value into it. However, be careful to assign an absolute location not used elsewhere--note that *the absolute location is part of the value*. If you screw up, you can wind up with several virtual addresses mapping to the same absolute location.

Also, remember that any patched locations not part of the original "Load" cannot be "Dump"ed.

When you modify the contents of a virtual IM location with Midas by typing fields of octal numbers, you *must supply the absolute address as part of the value*. Midas neither defaults this to the old absolute location nor warns you when you smash an absolute location already in use elsewhere. Consequently, it is possible to modify a different absolute location than the one you originally examined. *This is grounds for caution*. Normally, use the symbolic method for patching IM (discussed later), which does not have this problem.

To examine a memory location on the display, you usually type memory name and location or memory address and displacement. If you omit the name and simply type a number, then Midas defaults the memory name to either "IM" in virtual mode or "IMX" in absolute mode.

11. Testing Directly From Midas

As of this date, testing was not implemented on D0 Midas.

"Test", "LDR-loop", and "Test-All" allow the microprocessor to be tested from the Alto. Data patterns for the test are determined from the first subsidiary menu, as follows:

ZEROES	All-zeroes data
ONES	All-ones data
SHOULD-BE	Constant test pattern equal to value in SHOULD-BE
CYC1	Vector of the same size as the register containing zeroes with a single one-bit cycled left one position each iteration
CYC0	Cycled zero in vector of ones
RANDOM	Random numbers
SEQUENTIAL	0, 1, sequential numbers
ALTZO	Alternating all-ones and all-zeroes patterns
ALT-SHOULD-BE	Alternating contents of SHOULD-BE with its ones-complement

Testing is controlled/described by eight addresses on the display as follows:

LOW-ADDR

HIGH-ADDR	
CURRENT-ADDR	
ADDR-INC	(For memory tests only) These words all contain double-precision numbers. CURRENT-ADDR contains the last address tested. If ADDR-INC (normally 1) is positive, the test starts at LOW-ADDR and advances through the memory in steps of ADDR-INC until CURRENT-ADDR is greater than HIGH-ADDR. If ADDR-INC is negative, the test starts at HIGH-ADDR and goes by steps of ADDR-INC until CURRENT-ADDR is below LOW-ADDR.
LOOP-COUNT	The number of successful iterations of the test prior to failure or prior to aborting from the keyboard or with the mouse.
SHOULD-BE	What the data should have been.
DATA-WAS	What the data actually was.
BITS-CHECKED	Mask of bits checked (see below).

These addresses are in the fake DLDR memory (i.e., the values are stored in a table in the Alto's memory, not in the hardware).

When the value initially in LOW-ADDR is greater than HIGH-ADDR or greater than the largest legal memory address, it is reset to 0 before testing. Similarly, when HIGH-ADDR is initially greater than the largest legal address in the memory, it is reset to memlength-1 prior to testing.

"Test" AND's BITS-CHECKED with the maximum-sized mask for the register or memory being tested to determine a comparison mask for the test. If you previously tested a small register, then you must load BITS-CHECKED with a full-sized mask before testing a big register. If you don't want to check all the bits in a register, then clear the bits you don't want to check in BITS-CHECKED.

"Test", after showing the data-pattern menu, shows a menu of register and memory names and other test names, and executes a test of the one you select until the test fails or you halt the test from the keyboard.

The testable registers and memories appear in the second sub-menu for the "Test" action. This menu also includes several other machine-dependent test programs.

On Dorado, the additional tests are as follows:

STACK←	Tests writing RM with address in StkP
B←STACK	Tests reading RM with address in StkP
StkP+1	Tests Stkp←Stkp+1 (does not test RM read/write)
StkP-1	Tests Stkp←Stkp-1
StkP-2	Tests Stkp←Stkp-2
Shmv	Tests the output of the shift-control ROM's on the ProcH and ProcL boards against correct values.
WF	Tests loading ShC via WF←
RF	Tests loading ShC via RF←
IF	Tests loading ShC via "insert field"
EF	Tests loading ShC via "extract field"

< esc > will continue a register or memory test that has halted; it restarts an OtherTest that has

halted.

"Test-All" automatically loads BITS-CHECKED with a full-sized comparison mask prior to testing each item; memories are tested with LOW-ADDR = 0, HIGH-ADDR = memory length-1, and ADDR-INC = 1. It tests each register 200 times and makes 4 passes through each memory and each OtherTest. It is a good idea to run "Test-All" whenever the hardware is in a suspicious state.

On Dorado, the "LDR-loop" action should only be used when the "debug" command file has been executed. This requires a sophisticated understanding of the hardware and of the innards of Midas and is not recommended for novices.

Dorado Midas stores many microinstructions in a fake memory called LDR (see LOADER.MC). These are used by various actions to operate the hardware. "LDR-loop" allows these to be executed in non-standard sequences to beat on particular hardware problems.

"LDR-loop" accepts a list of LDR addresses separated by commas as input text. If only one LDR address is typed, the ABMUX register is loaded once with the selected data pattern, then the LDR instruction is repeatedly executed with UseABMux true for a scope loop.

When two, three, etc., up to ten LDR addresses are typed, a test loop occurs whereby ABMUX is loaded with the next data pattern, the first instruction is executed with UseABMux true, then the rest of the instructions are executed, and then the BMux is read back and compared against the original data under control of BITS-CHECKED. The loop stops when (data-read-back xor data-sent-out) & BITS-CHECKED is non-zero.

12. Scope Loop Actions

"Field-Loop" exercises signal decoding for particular fields of the microinstruction for scope loops. A microinstruction is fabricated from a no-op microinstruction in which the field selected from the first subsidiary menu is replaced by various values. The second subsidiary menu allows the value in the selected field to be incremented, decremented, and shifted.

"Rep-Go" starts the microprocessor at the address typed on the command line, waits for it to halt at a breakpoint or parity error, then restarts it at the original address.

On Dorado, the task for the original Go is taken from the TASK register; subsequent restarts do not reselect the task. The control section's NOTIFY register is reset before the first Go, but is not reset each time through the loop.

"Rep-SS" single-steps the microprocessor at the address typed on the command line endlessly.

On Dorado, "Rep-t2" endlessly executes the instruction in MIR and reloads that value into MIR. Unlike "Rep-SS", "Rep-t2" doesn't issue extraneous clocks while looping, so it is ordinarily more convenient for scoping.



DO MIDAS MANUAL

30 December 1977

by

Brian Rosen

Xerox Business Systems
Systems Development Department
3408 Hillview Road
Palo Alto, California 94304



1. Registers and Memories Known to Midas

The registers and memories known to Midas are as follows (numbers in octal):

Register	Width	Memory	Length	Width
APCTASK	4	TPC ^{2,5}	20	14
APC	20	IM	10000	100
CTASK	4	IMX	10000	44
CIA ^{4,5}	20	RM	400	20
CSDATA	20	T ²	20	20
PAGE	4	MAIN	??	20
PARITY ¹	4	VMAIN	??	20
BOOTREASON ¹	10	MAP	40000	20
RS232	20	DLDR ³	40	42
PCXREG	4	BP3	100	40
PCFREG	4			
DBREG	6			
SBREG	6			
MNBR	20			
ALURESULT	4			
SALUF	10			
SSTKP	10			
STKP	10			
MEMERROR	20			
MEMSYNDROME	10			
TIMER	20			
T(CTASK) ³	20			
TPC(CTASK) ³				
CALLER ^{1,3}	20			

1. Read-only to Midas
2. Task-specific registers
3. Fake memories and registers, artifacts of stuff inside Midas
4. Derived from NCIA
5. Virtual/absolute stuff applies

Most registers and memories listed above correspond to ones discussed in the D0 Functional Specification (January 16, 1978). The others are discussed below.

CIA is the complement of the hardware's NCIA.

T(CTASK) and TPC(CTASK) show the current task's T register and TPC. Changing CTASK will change T(CTASK) and TPC(CTASK).

The CALLER register shows the absolute value in TPC(CTASK) with the least significant bit forced to be a zero. When control store addresses are displayed in absolute mode, this is useless. However, in virtual mode CALLER will usually show the location that last did a CALL.

IM and IMX are virtually and absolutely addressed versions of the control store, discussed later.

VMAIN is the same memory as MAIN but is addressed by the current contents of MAP rather

than absolutely. In other words, to reference MAIN, Midas first loads a Map location with the absolute page location, then makes the reference. This is not done in referencing VMAIN. As of this date, the D0 did not have a memory system, and so this is not available.

2. Task-Specific Registers

Midas treats all task-specific registers (T and TPC) as 20-word memories. In other words, "T 6" is the T-register for task 6.

In addition, a special kludge allows you to display the 21st word (i.e., "T 20", "TPC 20", etc.) and have that be interpreted as the register for the *currently selected task*. The currently selected task is the value in CTASK.

3. Complications in the Display of Register Values

IMX and IM contain microinstructions, and the 44_g bits which are the value of the instruction are displayed the same way for all of these. A middle-button action over the value will print this value symbolically on the comment lines.

4. How Registers are Read/Written

The D0 contains no special hardware for MIDAS to enable reading or writing any state information without affecting the microprocessor. To enable MIDAS to control the D0, a special microprogram (the kernel) is loaded into the last page of the control store by midas when it does a Boot. The kernel uses the D0's printer interface in conjunction with the Blue Box to connect to an ALTO's printer interface. The MIDAS (running in the ALTO) communicates with the kernel through this hardware, passing commands and data back and forth between the two programs. When MIDAS wants to know the contents of a register, it asks the kernel to supply it; similarly, when MIDAS wants to change the contents of a register, it sends a message to the kernel with the address and the contents of the affected register. For registers which are part of the machine state which the kernel itself modifies (CIA, CTASK, APC, APCTASK, CSDATA, etc), the kernel maintains copies of the hardware registers in the R file. The copies are updated when the kernel is entered (via Boot, breakpoint or other fault). MIDAS manipulates the copies, examining and changing them as necessary. When GO or STEP is needed, the kernel loads the hardware state from the copies.

Breakpoints are done by replacing the instruction with another instruction containing the BREAKPOINT "F". The JA field of this instruction has the breakpoint number in it. MIDAS saves the original instruction and replaces it when the breakpoint is reached. MIDAS can analyze an instruction to find its successor(s), it will breakpoint all successors of an instruction when STEPping, or proceeding from a breakpoint.

5. Special Keyboard Input Formats

Registers and memories that contain microinstructions (IM and IMX) evaluate a special form of input as follows: The first character on the input text line should be "(" to change the values of several fields in the instruction without clobbering other fields, or "[" to reconstruct the value beginning with a no-op microinstruction. This is followed by a number of clauses of the form "Field+integer" separated by blanks and/or commas. The legal field names are MEMINST, RMOD, RSEL, ALUF, BSEL, LR, LT, F1, F2, JC, JA, CSpar and AT.

AT is defined only of IM, it sets the absolute address.

In addition to "field+value" clauses, Midas interprets RETURN (= JC+6) and the following control clauses: GOTO[n], GOTO[ntrue,nfalse,cond], CALL[n], and DISP[n]. The parameters n, ntrue and nfalse can be an IM (virtual mode) or IMX (absolute mode) address and modify JC and JA to contain a goto/call/dispatch to the target location. Arguments may be expressions such as FOO+3, if you like. The address evaluator assumes you are causing PAGE to be loaded correctly, it only worries about setting up the JA field.

6. STEP and GO

When the microprocessor halts, the values of CTASK and PC are remembered and used later, if you continue (i.e., execute a "GO" or "STEP" without specifying a starting address).

When you execute a "GO" or "STEP" at a new address, the value in CTASK is the task activated.

Although "GO" and "STEP" appear in the command menu, you will probably discover that it is faster to type "address;G" to Midas, an alternative to "GO", or "address;S", an alternative to "STEP". Similarly, ";S" is equivalent to a continue-"STEP" and ";G" to a continue-"GO". ":" is a synonym for ";S", and ";P" (Proceed) is a synonym for ";G".

7. BREAK and UNBREAK

The "BREAK" command inserts a breakpoint in the IM or IMX address typed on the input text line. The original contents of the instruction are saved by MIDAS and replaced with a special Breakpoint instruction just before MIDAS starts the processor. The BP memory shows you the status, address and contents of the breakpoints.

The address must be typed--there is no default break address. You will normally find it faster to type "address;B" to insert a breakpoint.

"UNBREAK" clears the breakpoint entry in the BP table. If no text is typed, the address defaults to the breakpoint that caused the last program halt. You will normally find it faster to type "address;K" or ";K" to remove a breakpoint.

8. BOOT

BOOT cause a complete hardware restart on the D0. MIDAS causes the D0 to go through a boot procedure, and then downloads the kernel program. The BOOT command without a file name loads KERNEL.MB, if a file name is supplied, it is used instead of KERNEL.MB. When midas is initially started, it does a BOOT of KERNEL.MB.

9. Acquiring Midas

To acquire Midas, use Ftp to retrieve [Iris] <D0> newmidasdisk.cm or midasdisk.cm (see section in D0 Microprogrammer's manual entitled "Getting Started". After loading, you must do Midas/I to initialize Midas on your disk. The total size of the files retrieved by these command files and those created by Midas/I is about 400 pages--be sure your Alto disk has enough space before plunging ahead.

10. Midas Maintenance

The current sources for Midas are kept on the "D0 Midas" disk (maintained by Chamley).

The various files in <eod>d0midasrun.dm are used as follows:

Midas.run	~240 pages	
Midas.syms	~38 pages	
Midas.Programs	~2 pages	(see below)
*.Midas	~2 pages each	Command files for Run-Prog and Read-Cmds
*.mb		Assorted micro-binary files loaded by command files

Midas.Programs contains a list of file names separated by blanks, commas, or carriage-returns. The names must be UPPER-CASE. This list serves two purposes. First, file FP's are built for all of the names to speedup OpenFile. Next, the list of names for the "Run-Prog" command menu is built. If the file name contains no extension, then hint FP's will be built for both name.MB and name.MIDAS and name will be put in the "Run-Prog" menu. (However, the hint FP's are not built unless the file exists, and the file name will not be put in the "Run-Prog" menu unless name.MIDAS exists). If the file name contains an extension, then it will be put in the quick OpenFile table, but won't appear in the "Run-Prog" menu. If the name ends in "*", a quick OpenFile table entry is made for name.midas and the name will appear in the "Read-Cmds" menu.

Midas creates and uses the following files (+ Swatee):

Midas.State	~29 pages	Built by Midas/I
Midas.FixUps	2 pages	Built when external fixups occur in .MB files being loaded (Current microcode never uses this.)
Midas.Errors	2 pages	Written when "Compare" fails

Altogether this is about 400 disk pages.

D0 SIMULATOR MANUAL

14 December 1977

by

Will Crowther
Robert Garner

Xerox Business Systems
Systems Development Department
3408 Hillview Road
Palo Alto, California 94304



1. Introduction

This manual is based on a 28 October 1977 memo from Will Crowther to the D0 Simulator users. It is unchanged from that memo except for corrections and updates. All questions and problems concerning the Simulator should be addressed to Bob Garner at SDD in Palo Alto.

2. Documentation

The user of the simulator must be familiar with three other systems which are documented elsewhere:

1. The D0 Assembler System, documented in this manual;
2. The mesa system, documented on [maxc]<mesa-doc>;
3. The D0 Functional Specification;

This manual will assume that the reader knows how a D0 works, presumably from reading the documentation on the processor. If the reader is unfamiliar with mesa, he is advised to get help from an expert in preparing his starting disk; thereafter mesa can be ignored.

3. Getting Started

To run the simulator you need a disk with the following modules on it:

Mesa.run (renamed runmesa.run if using Johnsson's exec)
 Mesa.Image
 wmanager.bcd
 s.bcd

s.bcd is on [Iris]<D0>s.bcd. Johnsson's exec is on [Iris]<johnsson>exec.run. All the other files can be found on the mesa directory on your local file server. It is also helpful but not necessary to have an installed mesa debugger on the disk. With a debugger, if the system crashes the user gets some clue about what happened.

To run the simulator you must start the file s. If using the regular exec type the following (user types the bold characters, system the normal characters):

```
mesa
new filename s
start filename ESC
```

If using Johnsson's exec (which I recommend) type:

```
mesa s
```

After a delay for loading, the Alto screen comes alive and is waiting for your instructions. In order to understand what to do now, you must realize that there are really three quite separate programs running in the machine at this instant.

1) There is the standard *mesa window package*, which is documented with the mesa system: it has complete control of the mouse, and will let you move the window(s) on the screen or create and destroy new windows. You can scroll any window in the normal mesa way.

2) There is a *D0 simulator*, which has a complete simulated state for a D0 (without any I/O). The D0 design is documented elsewhere. This simulator has a few features which the real D0 lacks: in particular, there is a control register which will start the machine when something is written into it. Depending on whether a 1, 2, or 3 is written the simulated D0 will run for one cycle, one instruction, or forever (until a break is encountered). The D0 simulator communicates with the rest of the system through a pair of routines which read and write simulated D0 memories. This is the only path into the simulator.

3) There is a *ddt*, which is a complex teletype-oriented user interface. The *ddt* accepts one character user commands, optionally preceded by a single parameter, converts them into commands to the simulator, and displays some sort of result at their completion. Since the *ddt* is trying to present a nice interface, it knows something about the format of D0 memory and D0 instructions, and can print the latter in a fairly reasonable way. It is often unnecessary to make a distinction between the *ddt* and the simulator, and I will occasionally confuse the two in the following. But sometimes the distinction is crucial for an understanding of the whole package.

4. Using DDT

You are now in the *ddt* and able to type *ddt* commands. The format of almost all commands is a single optional parameter followed by a single command character. The parameter is either an octal number or an alphanumeric string, while the command is either a punctuation character or a control character (written in this memo as ↑X. Note that "↑" written by itself signifies the up arrow command character). The characters "+", "-", "*", and Space are really command characters, but their only effect is to help build up a complex parameter from a simple one. Using these commands one could type *myStart+5* and use it as a parameter. In the use of strings the distinction between uppercase and lowercase is ignored, so that string, String, and STRING are all the same symbol.

Note that the parameter is specified in *octal*. Except for two minor exceptions, the whole of the *ddt* operates in octal mode only, both on input and on output, and there are never any decimal numbers involved.

5. Load and Dump

The simulator is not of much use without a microcode program to simulate. The simulator will load the output of the micro/microd microcode assembly system (a ".mb" file). In addition the simulator load command expects there to be a source file (a ".mc" file), which it places into a second window on the alto display. The relevant *ddt* command is "name↑L", which loads files name.mb and name.mc. The assembler and its input language are described elsewhere. Usually ↑L is the very first command given to the *ddt*.

6. Examine and Change

After the load the user may wish to examine or change some of the memory locations in the simulated machine. He may examine a location by typing its address as the parameter and "/" as the command. The address may be specified either as an absolute octal number or as a symbol (which presumably came from the .mb file out of the assembler, but see below for a way to define symbols in the *ddt*). Since there are several memories, the user is expected to precede the octal number with a single letter to indicate the desired memory. For example, "i23/" would inspect register 23 of the instruction store. The defined memories are i (instruction), r (register), m (main), c (control=d0 hardware registers), and z (map). If the single letter is omitted it defaults to whatever the previous memory was.

Normally the contents will print out in octal, but for the i memory that is not much use, so the i printout attempts to interpret the instruction symbolically. For the most part this is possible, but

sometimes the meaning of an instruction depends on the context in which it occurs. One will frequently see a goto 45 interpreted as a []+T, goto 45. Here the ddt is not smart enough to realize that the assembler specifies a T source for the alu and no store back when there is nothing else to do. (Of course the very next instruction might test the alu, so one cannot know for sure that this is a null operation). Typical instruction printouts are:

```

i300/  bfbx,R64db←R64db SALUF T  Dispatch BB
i301/  T←ldf[pos=4,size=6] of CSData  goto l11
i111/  Pstorel[R20]←R4  goto 130
i130/  freezeResult, T←R4←R4-T  Return
i131/  t←pcf AND ~T  Call Loop

```

There are other ways to inspect memory. One can type linefeed(LF) to inspect the next location, and "↑" to inspect the previous. One can type TAB to inspect the location specified as the destination in the previous instruction printout (but watch out for Call, Return, and Dispatch, which may not do what you expect - the ddt is only looking at memory, not executing it). TAB is the most useful way of examining instruction memory.

One can change memory. This is particularly useful for setting up test cases during a debugging session. The method is to examine the desired memory location by any of the methods described above, and then to type a new contents followed by a carriage return (CR). For example, "r15/123 456 CR" will change r memory location 15 from 123 to 456. LF, ↑, and TAB will work just as CR, and in addition will go on to inspect a new location.

To change the instruction memory, first display its contents as you normally would (such as with "/") then type "~" and the instruction will be typed out by labeled fields, with each field given as an octal number. Then, to change any field, type "field+valueCR", where "field" is a field name and "value" is the new octal value for the field. The new i memory value must be opened by "/" again before "~" will type the new field value. One can also change the i memory by typing in a 16 digit octal number (yes, the input is triple precision), but in practice that is too painful to attempt.

7. Simulator Execution

Eventually one tires of looking at the program and decides to run the simulator on it. The easiest way to do this is to type the start address followed by ↑G. If you are just learning the simulator I do not recommend this way, but if you use it, the simulator will execute instructions as the program directs until one of three things happens:

1) a breakpoint is encountered. You may set breakpoints in the file loaded from the assembler, or you may set them by typing address Ctrl B. The simulator will stop with the breakpoint instruction about to be executed.

2) one of many illegal instructions is encountered.

3) you type a backspace (BS).

Another way to run the simulator is to type "address↑S". This will prime the simulator to start at the specified address, but will actually execute nothing. Another ↑S without a parameter will step the simulator forward one instruction. In this way one can step the program forward one instruction at a time. I recommend this mode when first learning the simulator.

When the simulator stops execution and returns to the ddt, all of the active registers of the simulated D0 are accessible, as well as all of the memories described before. One can type t/ or apc/ and see what is currently in these registers. One can even change these registers in mid stride. At each return to ddt, two especially useful registers are automatically printed. These are the register holding the address of the next instruction to be executed, and the MIR which holds the instruction itself.

I want to tell you at what part of its cycle the simulated D0 stops, for that is vitally important for understanding what the various registers mean. In order to do that I must explain a little bit about how the simulator treats time. The answer will sort of turn out to be that the D0 has stopped just after the start of cycle zero of the machine, so that all of the registers which are loaded at time zero have actually been loaded, but none of the gates which hang off of those registers have yet started to change.

At the beginning of every cycle the simulator starts with a record which contains the complete state of the D0. This record contains things like h1 and cia and apc. It first executes a set of procedures whose job is to compute various gating functions from that record. For example, the actual r address specified and the output of the cycle/masker. It next executes another set of procedures whose job is to compute a new record which will be the state of the machine at the start of the next cycle. This new record is kept completely separate from the old one until the very end of the simulated cycle. Finally, the new record is copied into the old one (with due care for the abort case), and the cycle repeats. When the machine stops, the copy over has *not* happened, but the ddt is looking at the *new* registers. Normally the simulator stops at the end of cycle 3, but because the ddt is looking at the new register it seems that it is the beginning of cycle 0. Actually, since an abort can prevent the normal loading of some of the registers, one must take care when interpreting the ddt output. The ddt is willing to display not only the contents of the new state record, but also any of the gating functions which seem to be of interest. There are approximately 60 values which can be examined in this way.

One of the entries in the state record is a 2 bit counter (called *Cycle*) which cycles through the four stages of the instruction being executed. Cycle is used to set one of four corresponding booleans called time0, time1, time2, and time3. The booleans in turn are used to decide whether a particular part of the simulator logic should execute. Thus after four passes through the main loop of the simulator one instruction will be completely executed. To mimic the D0 overlap, the simulator sets another of the time booleans on each pass. This will force the execution of logic corresponding to the appropriate cycle of the overlapped instruction, and because of the nature of the D0 design the two cycles will not conflict. But the simulator will work equally well if the overlap is not called for, which means that it is easy to run the D0 simulator in a non-overlapped mode. The value of such a mode is in the debugging of microcode; it is much easier to understand what is going on if you have all of the variables relevant to the current instruction at hand, instead of seeing half of them as they have been stepped on by the next instruction. There is a control register (c1 = "overlap") which can be set to zero for overlapped mode and one for nonoverlapped. The default is *nonoverlapped*.

I recommended the single step mode for the initial experience with the simulator because I found the various registers did not always have the values I expected, even when the simulator was working correctly. With the single step mode one is at least confident where the program has stopped and by what path it got there. One final caution: the main memory is of course asynchronous, and the result does not always show up until several instructions have been executed. If you stop just after computing your final answer, you may never get to see it! Also, the memory does not slow down during nonoverlap mode, so memory operations will happen sooner than they would during overlap mode (with respect to the rest of the program).

By empirically timing some programs, it seems that the simulator runs approximately 25,000 times slower than a real 70 nanosecond D0. In other words, one second of D0 time is equivalent to approximately 7 hours of simulator time. In general the simulator will run twice as slow if in nonoverlap mode.

8. Command Strings

The user has the ability to enter a string (called a *command string*) for the ddt to remember. He can later specify that the ddt execute the whole string as though it had been typed from the keyboard. A typical string might single step the simulator and print out several registers for the user to examine. The ddt has storage for four such strings, labeled 0,1,2,and 3. The syntax for command string entry is "label↑Zcommand string↑Z", where "label" is the string label, and "commandstring" is a list of ddt commands with arguments written as they would normally be entered into ddt. The syntax for executing the command string is "label ESC". An ESC with no label repeats the last command string. A ↑Z with no label implies string 0.

It is also possible to read into ddt a string which resides in a file. Type "name↑F" and the commands in file "name" will be executed by ddt. They can also be loaded into a command string by typing "name↑F" immediately after the first ↑Z used to set up the string (i.e. "label↑Z name↑F↑Z").

9. DDT Commands

The rest of this manual lists and describes each of the ddt commands, including all those mentioned above plus a few other less used ones. Following the ddt commands is a list of the simulator memories, with special emphasis on the 64 simulator control registers.

editing:

DEL, BS ,↑A	abort the current command
CR	with no parameter moves the carrot

inspect and change(change only if explicit parameter):

memloc/	display contents of location "loc" of memory "mem"
value CR	change contents to "value" (i.e., m20/123 456CR)
value >	same as CR (for wasting less display space)
valueLF	change and inspect next location
value↑	change and inspect previous location
valueTAB	change and inspect jump address of displayed inst
field←valueCR	change field of last displayed thing

building parameters:

+	plus (i.e., m20+30/)
-	minus
*	times
Space	plus

load/dump:

name↑L	Load
name↑D	Dump [hasn't worked since 36 bit D0 change]

D0 control:

address↑G	run D0 (Go)
address↑S	Step D0
address↑B	set Breakpoint
address↑C	Clear breakpoint
num↑W	set task number (W stands for Wakeup task)
↑O	put the simulator in Overlap mode (like the D0)
↑N	put the simulator in Nonoverlap mode
BS	halt a running simulator

ddt control:

↑Q	exit ddt (Quit) (Shift swat is faster)
?	type list of ddt commands
=	type the last thing in octal
~	type the last displayed thing in instruction format
label↑Z	enter a command string terminated by another ↑Z
label ESC	play the command string through the ddt
label↑T	type out command string "label"
name↑F	play the command string from file "file"
↑R	type out all the R memory symbols with their values
name: valueCR	define symbol "name" to have value "value"

10. The Simulator Memories

i memory: 4K 48-bit words

- 1) only 36 bits are used.
- 2) the parity bit holds breakpoint information. IF YOU USE THE PARITY BIT FOR DATA, BEWARE - CLEAR ALL BREAKS WILL CLEAR IT.
- 3) the simulator keeps i memory on the disk, and caches two 256 word pages in core.

m memory: 2K 16-bit words

r memory: 256 16-bit words

z memory: 8 16-bit words

nominally 16K of 13 bit words pointing from virtual to real addresses. Actually 8 words pointing from real to virtual addresses. Searching the 8 words slows the simulator down a little, but not as much as keeping the z memory on disk.

t memory: 16 16-bit words

only [ctask] can be read and written by the ddt.

c memory: 64 48-bit words

- 1) most of these addresses have only 16 bits of memory behind them, but a couple have more.
- 2) most of these addresses are implemented by a table of pointers to various structures in the simulator data region. In particular, there are a lot of pointers into the output version of the state vector, and a lot into the computed gating functions.

10.1. The C Memory in Detail

The following list gives the c address, followed by the ddt symbol for that address, followed by a brief description of the register. There is no longer any method behind the ordering of these registers.

Note: If you use the following c memory names as symbols in your microassembly source (such as

by the SET,MP,SP,MC,RM, or RV macros) then the c memory value defined below will be overwritten by your source code values.

c00 none: write 1-3 issues command to the simulator:
 1 = > run (↑G)
 2 = > step (↑S)
 3 = > single cycle

c01 overlap: zero=overlap mode(↑O), 1=non overlap mode (↑N)

c02 pc: D0 Register (i mem Program Counter)

c03 break: set break (↑B)

c04 clearBreak: clear break (↑C)

c05 clear: write 1-3 issues command to the simulator:
 1 = > clear i,m,r,z,t, and tpc memories
 2 = > clear all breakpoints
 3 = > clear output state record (D0's Registers)

c06 stkp: D0 Register (STack Pointer)

c07 pcf: D0 Register (mesa Program Counter Fetching)

c10 cycleCtl: D0 Register (CYCLE Control = dbx2..5,,mwx)

c11 sstkp: D0 Register (Saved STACK pointer)

c12 sb: D0 Register (Source Bit)

c13 t: D0 Register (Task temporary[ctask])

c14 hl: D0 Register (cycler/masker input)

c15 h2: D0 Register (ALUb input)

c16 stack: RMemory[stkp]

c17 alua: asynchronous A input of ALU

c20 alu: asynchronous ouput of ALU

c21 mpanel: D0 Register (Maintance PANEL - decimal output)

c22 mir: D0 Register (Micro Instruction Register)

c23 rselGates: asynchronous R Address computed from mir

c24 jumpGates: asynchronous jump Address computed from mir

c25 cycle: the value of Cycle for the cycle last executed

c26 apc: D0 Register (Alternate Program Counter)

c27 flags: a set of bits indicating control conditions:
 1 = > abort
 40 = > steal
 100 = > r write back
 200 = > t write back
 400 = > time3
 1000 = > dispatch
 2000 = > freezeResult

c30 ctask: D0 Register (Current TASK)

c31 mw: asynchronous function of sb,db, and mnbr

c32 db: D0 Register (Destination Bit)

c33 pcx: D0 Register (mesa Program Counter eXecuted)

c34 rs232: D0 Register

c35 printer: D0 Register

c36: unused

c37 saluf: D0 Register (Special ALU Function)

c40 mnbr: D0 Register (Minus Number of Bits Remaining)

c41 page: D0 Register

c42 cia,next: D0 Register (Current Instruction Address)

c43 tpc: D0 Register (Task Program Counter[ctask])

c44 conds: a set of booleans related to skip conditions:
 1 = > attention
 2 = > r neg
 4 = > r odd

10= >a carry
 20= >a neg
 40= >a zero
 100= >overflow
 c45 apctask: D0 Register
 c46: mc2 real memory page
 c47: mc2 virtual memory quad word rounded address
 c50: mc2 type
 c51: mc2 r address
 c52 mc2going: mc2 has been going for this many cycles
 c53: mcl real memory page
 c54: mcl virtual memory quad word rounded address
 c55: mcl type
 c56: mcl r address
 c57 mclgoing: mcl has been going for this many cycles
 c60: unused
 c61 clock: number of cycles executed (decimal output)
 c62: unused
 c63 csData: D0 Register (Control Store DATA)
 c64 csin: D0 Register (Control Store INput)
 c65 csinExtend: D0 Register (Control Store INput EXTENDED)
 c66 sbx: D0 Register (Source Bit eXecuting)
 c67 dbx: D0 Register (Destination Bit eXecuting)
 c70 mwx: D0 Register (Minimum Width eXecuting)
 c71 nextm7: address of instruction executed 7 insts ago
 c72 nextm6: address of instruction executed 6 insts ago
 c73 nextm5: address of instruction executed 5 insts ago
 c74 nextm4: address of instruction executed 4 insts ago
 c75 nextm3: address of instruction executed 3 insts ago
 c76 nextm2: address of instruction executed 2 insts ago
 c77 nextm1: address of instruction executed last

