DORADO

MIDAS MANUAL

24 June 1983

by

Edward R. Fiala

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

Filed on: [Indigo]<DoradoDocs>DoradoMidas.press
Sources on: [Indigo]<DoradoSource>DoradoMidasManual.dm

This manual describes a largely machine-independent loader/debugger for microprocessors originally developed for the Maxc2 computer and since used for the Dorado, Dolphin, and M68 microprocessors. This manual is specialized for the Dorado version of Midas.

TABLE OF CONTENTS

LIST OF TABLES

l. Introduction

Midas is a loader/debugger that runs on an Alto and controls its target machine remotely. It can load/dump microprograms assembled by Micro, examine and modify storage, and test hardware in an assortment of ways. Versions exist for Maxc2, Dorado, Dolphin, and M68 microprocessors.

Midas is coded about 95% in Bcpl and 5% in assembly language. The Maxc2 version was implemented by E. R. Fiala and H. E. Sturgis. The Dorado, Dolphin, and M68 versions consist of machine-independent modules implemented by E. Fiala (Overlay and LoadRAM 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; and D. Charnley, C. Thacker, B. Rosen, C. Hankins, and E. Fiala for Dolphin.

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

2 Storage Requirements

Midas requires about 500 Alto disk pages, using the following files:

Midas.Run	~350 pages	
Midas.Syms	~40 pages	
Midas.Errors	~8 pages	Error message strings for Midas swat calls
Midas.Programs	~2 pages	(Discussed below)
Midas.UserPrograms	~2 pages	(Discussed below)
*.Midas	~2 pages each	Command files for "RunProg" and "RdCmds" actions
*.mb		Assorted micro-binary files loaded by command files
Midas.RunProg	~31 pages	Built by Midas/I
Midas.Dtach	~31 pages	Built by Midas/I
Midas.FixUps	~2 pages	Created by Midas/I (used when loading .MB files)
Midas.Compare	~2 pages	Created by Midas/I, written when "Cmpr" action fails

Dorado Midas can be obtained by loading [Indigo]<Dorado>DoradoMidasRun.Dm and retrieving <Dorado>Midas.Programs with Ftp. You must do Midas/I to initialize Midas on your disk after retrieving these. Subsequently, new versions of Midas can be retrieved by executing the D1NewMidas.cm command file from the Alto Executive. Midas runs only under Alto OS versions 17 or later.

To setup an Alto disk for use in Dorado microcode development or hardware debugging, you can install the Alto OS on a blank disk using the long installation dialog and erase the disk. When this finishes, fetch [Indigo]<Dorado>DoradoUnbugDisk.cm and execute this command file from the Alto Executive; it will retrieve Midas and a number of other files that are needed when using an Alto to control Dorado.

3. Starting and Exiting from Midas

Midas may be started from the Alto Executive in the following ways:

midas	start Midas.
midas/i	initialize and fire up Midas. This is about 8 seconds slower than a normal start. /i must be used after any Midas files move or change because Midas keeps the FP's (File pointers) in its saved state. Changing any files named in Midas.Programs or Midas.UserPrograms, changing the Alto OS, and changing files named Midas.* all require /i.);
midas debug	starts Midas and immediately reads commands from the "Debug.Midas" command file

Dorado Midas may attach to any of up to 20 machines accessible through its Diablo Printer interface. If only one machine is accessible, it immediately connects to it; if more than one machine is there, it first puts up a menu of accessible serial numbers, and then connects to the one selected by the user. After connection, subsequent actions affect only the connected machine.

Although Midas itself can deal with up to 20 machines, the hardware is presently limited to less. It was observed that ringing on the communication cable introduced extra clocks when too many machines were connected, so Midas couldn't communicate reliably with target machines. The PARC technicians presently observe a limit of about 6 machines, which seems to work. No hardware fix is likely to be made for this problem.

Midas will seize the hardware only if the connected machine is halted; if running, Midas waits for the machine to halt or for you to execute "Abort" or "Dtach" actions. "Dtach", equivalent to exiting to the Executive and restarting Midas, also appears in the main command menu and in the submenu put up by "Go", so you can start a microprogram running with "Go" on one Dorado, then "Dtach", connect Midas to another machine, and do something else (However, if you Dtach and later reattach to a machine, you will have lost the display configuration and address symbols, which might be inconvenient).

Note that there are two different arrangements for the initial Midas display. For both arrangements the left-hand display column shows the principal Dorado registers, and the middle column shows several other items. When you initially attach to a machine, the right-hand column will show voltage, temperature, and current readings collected by the baseboard microcomputer; after a "RunProg" action, the right-hand column will show items used by Midas hardware testing actions.

To exit from Midas type SHIFT-SWAT (i.e., simultaneously depress the left-hand shift key and the right-most, lowest unmarked key); this will close any open output files prior to exit and disconnect the Alto from the Dorado it was controlling. Note that on "Dtach" or exit, if the Dorado was running, it will not be disturbed, but if halted, Midas first restores the hardware state as though it were about to continue, so it will be possible later to reattach and continue a program that was stopped at a breakpoint.

4. Midas Display and the Mouse

The Midas display is arranged as follows:

Blank area at the top (unused);
20 lines x 3 columns of name-value menus;
Blank line;
Program and elapsed time line;
Blank line;
Two command comment lines;
Blank line;
Three lines of command menu;
Blank line;
Input text line;
Blank area a the bottom (unused).

The program line will show the Midas release date or the name of the last program loaded. The right-most part of this line will show elapsed time during long-running actions such as "Go" or "Test"; it shows the execution time of Midas initialization, the last command file, or the last action at other times.

Midas uses the two comment lines to report results of actions that it executes.

When you move the mouse over a name-value menu or the command menu, the selected item (if any) inverts black and white. Mouse actions execute when you RELEASE all mouse buttons, so you can move the mouse with buttons depressed without causing damage. If the mouse has moved off of the menu that was selected when the first button went down, nothing will happen when the buttons are released.

Some menus have additional actions "underneath" the ones normally displayed which will appear when you depress appropriate button combinations, as discused below. In other words, when you DEPRESS buttons, the menu may change; when you RELEASE ALL BUTTONS the selected action will get executed. On Dorado Midas, only name-value menus have actions underneath the ones normally displayed.

Since you can neither depress a button combination simultaneously nor release the buttons simultaneously, Midas accumulates the *union* of all buttons that go down. This button-union governs the "underneath" menu displayed, if any, and is the argument passed to the action procedure when all buttons are finally released.

5. Name-Value Menus

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. If a number (default radix 8) is examined, the memory name is defaulted to "VM," so examining "1234" will cause "VM 1234" to be displayed.

Name-value areas are of *different sizes*. Smaller menus on the left are already filled in when you fire-up Midas; others are empty. Any item can be put in any menu, but larger menus on the right are better for items with long names or values. If an item overflows its menu, the right-most characters of its name get truncated, then the left-most characters of its 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 (top)* mouse button. Memory addresses in your microprogram may optionally be followed by a displacement "+n" or "-n"; " n" is the same as "+n". Midas will obtain the value of the item from the hardware and display it.

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

The address and data items in a name-value menu are affected by the *radix* and *display mode* for the item, initially defaulted from a table indexed by the register or memory number. The address offset and value radices are always identical--Midas does not allow these to be independently specified. On Dorado, octal radix is the default for everything except the microcomputer memories ($ABSOLUTE and $ABS), where hexadecimal is used. The user may change the radix with the actions discussed below.

The display mode for a value may be either *numeric*, *search*, or *symbolic*.

> Numeric mode shows the value as a sequence of numbers (in the chosen radix) separated by blanks; this is the default for almost all items.
>
> Search mode shows the value as an address symbol plus offset; this is illegal except for registers or memories that normally contain pointers into some other memory (e.g., on Dorado, search mode for TPC, TLINK, etc. shows the nearest IM address symbol less-than-or-equal to the value plus an offset; for MEMBASE, BR address symbols are shown; for TIOA, DEVICE address symbols are shown.). Search mode is not the default for any memory or register because it is slightly slower than numeric mode due to symbol table access and because more screen area is required to accommodate long address symbols; however, you may find search mode convenient for some of the items mentioned above.
>
> Symbolic mode results in a special procedure being called to print the value for the item. Symbolic mode is illegal except for the MSTAT memory and the UPTIME and TGLITCH registers on Dorado; for these it is the default; special procedures do not exist for any other items.

When Midas thinks that the value in a register may have changed, it reads its value from the hardware and updates the display; the times when Midas does this are discussed later. Names are sometimes preceded by *, indicating that the value has changed, or by ~, indicating that Midas was unable to read the value for some reason (e.g., the machine was running). For an item marked with ~, the old value, which might be wrong, is displayed.

Once some register or memory address has been put into a name-value menu, various mouse button combinations over the name or value may be used to modify the way it is displayed, sequence through words in a memory, pretty-print the value on the comment lines, or show address equivalences. These are summarized in the table below:

Buttons	*Name-field*	*Value-field*
Left	Examine	Change value
Middle	Alternate printout	Pretty-print value on comment lines
Right	A+1, A-1 menu	Append value to input line
Left+Middle	Radix menu	Radix menu
Middle+Right	Fill column menu	Display mode menu

When a button combination selects an alternate menu, the alternate menu will replace the standard menu while the mouse buttons are depressed; if you release the buttons over an alternate menu item, it will be executed; if you are outside the menu when the buttons are released, the standard menu will be restored and nothing will happen.

The "A+1", "A-1" menu appears for memory addresses, but not for registers; these increment or decrement the memory address in the menu, displaying the successor or predecessor. The "FillC" menu allows you to examine successors (A+1, A+2, etc.) in the menus below the selected one; the whole column is filled with successors, if the input text line is blank; otherwise, the input text line is evaluated to a number N, and N lines are filled in with successors. The last address examined is left on the input text line, so you can iterate the examine and fill column actions to achieve scrolling.

Releasing the *left button* over a value stores the value of the input text (or 0 if no text typed) in the selected register. For memories and registers whose values are displayed as several fields, the input text must also be divided into fields; omitted fields are zeroed. Each field may consist of numbers or memory addresses separated by +/-; expressions are evaluated using the radix for the item.

> Note: On Dolphin and Dorado, IM memory words show an absolute address with each value; it is impossible to modify this address from Midas--the correspondence between virtual and absolute addresses can only be established by loading a microprogram. Several other items also have read-only fields that cannot be written, as discussed later.

Provision is made for *special input evaluation* based upon the register or memory; whenever the input text cannot be evaluated as a sequence of fields, the special input procedure (if any) is called. At the present time, special input procedures are implemented for registers and memories that contain microinstructions (MIR, IM, IMX, IMBD, and LDR on Dorado) and for 16-bit registers. These are discussed later.

Releasing the *middle button* over a value pretty-prints 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. Registers and memories that contain microinstructions may also be printed symbolically. Other pretty-print information is detailed 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 or to display a memory address that is pointed to by the value in some other register.

6. Command Menu

The command menu holds a list of *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 "Ld". During execution, some actions show alternate menus.

For almost all actions in the command menu, mouse buttons are equivalent. On Dorado, the "Go" and "SS" actions are an exception; executing one of these with the right button is interpreted as "proceed," left button as "new go." The "DMux" action is also an exception.

Many common actions may alternatively be initated by *keyboard command characters*, as given in the action table below.

Table 1: Command Menu Actions

Input	Char	Menu Item	Comments
Actions (potentially) available on all implementations of Midas			
[File]		RunProg	Reset symbol table and display, then do RdCmds.
[File]		RdCmds	Executes command file (def. ext. ".Midas") on input text line or from submenu.
		ShowCmds	Shows command file text for selected menu items.
File		WrtCmds	Write subsequent commands on File.
Files	;L	Ld	Loads .MB files (names separated by ",").
Files		LdSyms	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	Cmpr	# Compares hardware data to that in .MB file.
Addr	=		Prints value of an address (illegal in com-file)
IMaddr	;B	Brk	Inserts break point.
[IMaddr]	;K	UnBrk	Removes breakpoint (default address = last break).
[IMaddr]	;G	Go	* Start at address (continue if nothing typed).
[IMaddr]	;P	Proceed	* Start at address without IOReset or control section reset (continue from break if nothing typed).
[IMaddr]	:	SS	Single-step at address (continue-step if nothing typed).
IMaddr	)	--	Call subroutine with args (e.g., "FOO(A1,A2)").
		Reset	Reset the machine; assorted options from a submenu.
		Test	Test register, memory, or other test with data pattern and item selected from submenus.
		TestAll	Test everything.
IMaddr	;R	RepGo	* Go at address, repeatedly restart after halts.
IMaddr	;S	RepSS	* Repeatedly SS at address.
		PEscan	Scan local memories for parity errors (IMX, IFUM, RM, STK, CACHEA, and CACHED on Dorado).
		Fields	For scoping.
LDRaddr		LDRtest	* Manually-constructed test sequences.
		Virt	Changes IM address interpretation to be virtual.
		Abs	Changes IM address interpretation to be absolute.
Actions peculiar to Dorado Midas			
		SetClk	Set the clock speed 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).
		RepT2	* Repeatedly does t2 (for scope loops).
		Dtach	Disconnects Midas and repeats the connection procedure.
		Config	Modify board configuration, cache, map, storage, and IM parameters from submenus and adapt Midas to these.
		DMux	Modify display of DMux items as discussed later.
[IMaddr]		OS	* "Opcode step" = SS program until a halt condition occurs or an IFUJump has been executed.
[IMaddr]		SimGo	* Like "Go" invoking the DMux checker after each step.
		SimTest	* Random instruction test using MIR and the DMux checker.
		HWChk	Displays submenu of tests and scope loops for hardware checkout.
		Active	Active, PrePassive, and Passive modes discussed later.
		Update	Read registers and display new values (used while passive)

* = requires preceding "TimeOut" action in command 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 ShowCmds by StopShow, WrtCmds by StopWrt, Virt by Abs. The Active, PrePassive, and Passive actions are in a "ring"; the current hardware mode is shown in the menu; bugging it will change to the next mode. The DMux action will be displayed as "DMux", "OldDMux", "DWrong", or "DChk" according to which DMux table is currently displayed. All of these actions are discussed later.

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 type an address; then you could bug "Go" in the command menu, but probably "address;G" is more convenient 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.

7. Keyboard

"=", "+", "-", "#", and "!" are legal symbol constituents in microprograms but will cause trouble for Midas if they appear in address symbols. "=" is an action character that will prettyprint the memory name and offset and the nearest address symbol less-than-or-equal to the value of the string on the input text line. "+" and "-" have their usual sum and difference meanings in evaluating input expressions. "#" (octal), "!" (decimal), and "%" (hexadecimal) may be inserted anywhere in a number to overrule the default radix; e.g., "#123" or "123#" will force the evaluation of the number "123" to be in octal. The default input/output radix for almost everything on Dorado Midas is 8 (octal).

> Exceptions are UPTIME and TGLITCH, which show hr:min:sec in decimal and the $ABS and $ABSOLUTE memories, which use hexadecimal for both the address and value.

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

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 (only defined for actions that display "Abort")
control-Z	abort all command files in progress
escape	repeat previous action (special for "Test" and "TestAll")
return	special following "Test" or "TestAll"
control-D	turns on the display (used during command files)
control-O	turns off the display (used during command files)
shift-swat	exit cleanly from Midas

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.

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 writing a register or in some other way or have interrupted an instruction from which continuation is impossible.

Although command menu items "SS", "Go", "Brk", "UnBrk", "RepSS", and "RepGo" are provided, the keyboard characters equivalent to these are usually more convenient.

8. 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 "RunProg" or "RdCmds". Alternatively, you may type a file name first, then bug one of these actions (If you type a file name after the subsidiary menu is put up and then bug "Abort", the command file will also be executed; it is not clear whether this is a bug or a feature.).

> File names in these sub-menus are contained in the files Midas.Programs and Midas.UserPrograms, each of which has a list of file names separated by blanks, commas, or carriage-returns. Midas.Programs is part of the standard Midas release; Midas.UserPrograms is an optional file that a user of Midas can prepare with his own stuff. The names must be UPPER-CASE. These lists serve two purposes: building file FP's to speed OpenFile, and preparing the menu items for "RunProg" and "RdCmds".
>
> > 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 "RunProg" menu. (However, the hint FP's are not built unless the file exists, and the file name will not be put in the "RunProg" menu unless name.MIDAS exists).

If the name ends in "*", a quick OpenFile table entry is made for name.midas and the name will appear in the "RdCmds" menu.

If the file name contains an extension, then it will be put in the quick OpenFile table, but won't appear in the "RunProg" or "RdCmds" menus.

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

"RdCmds" executes the actions stored in the command file; it is frequently used to modify the display in various ways by executing command files that show collections of items that are of interest.

"RunProg" first clears the symbol table and restores the display to its initial arrangement; then it executes the actions in the selected command file; "RunProg" is used to completely change contexts--to run a new microprogram, for example.

Generally, there is one "RunProg" 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.

The command-file facility is actually a (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 with "RunProg" and "RdCmds" subject to 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) Nesting is limited to 8 levels (a parameter that could be increased if more levels are needed).

(3) Command file names appearing in the "RdCmds" or "RunProg" menus must not duplicate any other action names used by Midas. If this happens inadvertently, command file interpretation will be substituted for the intended action whenever that action is executed from a command file. Fortunately, interactive execution of the duplicated action is unaffected. Midas does not detect this.

(4) Midas is tight on storage when running the simulator (SimGo and SimTest actions), and available space might become exhausted. Three separate storage resources are husbanded by Midas: stack, sysZone, and other. Some sysZone is consumed by each open file, and the simulator puts one overlay there. If command, output, and WrtCmds files are simultaneously open when SimTest or SimGo is executed from a command file, I think sysZone will overflow,

but there should be enough sysZone storage for two open files plus the overlay.

Note that even when command files are nested to several levels, Midas only keeps open the current command file, so nesting command files does not affect the storage requirement.

The simulator also steals space from the stack for one overlay, and it is conceivable that running SimGo from a command file and then aborting from the keyboard will exhaust the stack. However, if Midas is properly checked out, then this should not happen.

Finally, unused storage and the name-value menu bit buffers are used for the rest of the simulator. Since this storage is consumed by file handles and name strings which you add to Midas.Programs and Midas.UserPrograms, this storage might become too small for the simulator. If this happens, you will fall into Swat with an "Out of storage" message when you try to run the simulator.

(5) There are other size limits you must observe. First, the number of RdCmds and RunProg actions you may add to the ones already in Midas.Programs is limited to about 25 (on 24 June 1983). This limit is imposed by the table storage reserved for these actions when Midas is built. If you exceed this limit, the submenu for the "Test" action cannot be shown, and Midas will fall into Swat during Midas/i initialization with a message indicating the number of excess actions. The limit applies not to the number of names in Midas.Programs and Midas.UserPrograms, but rather to the number of these names which exist on your disk. So to recover from this overflow, you can either prune the number of names in Midas.Programs or Midas.UserPrograms, or you can delete some of the *.Midas files from your disk.

Secondly, the list of names in the RdCmds or in the RunProg menu must not overflow command menu space. If one of these lists overflows, Midas will fall into Swat during its Midas/i initialization with a message telling you by how many characters you have exceeded the limit. Here again, you recover from the overflow by either pruning names or deleting *.Midas files. Or you can continue from the Swat call; if you continue, Midas will simply truncate the names which won’t fit when it displays the menu.

(6) Some actions occur in command overlays where a sequence of menu items must be bugged before the action is specified. For example, on "Test", you must specify the data pattern (e.g., "Random"), then a "TimeOut", and finally the register or memory to be tested (e.g., "RM"). It is illegal to start a nested command file before the subsidiary actions are specified; if you try to do this, Midas will give an error message.

(7) Some actions require a preceding "TimeOut" action. The command file must have two actions after the "TimeOut" before terminating.

A number of actions, some of which cannot be given interactively, are useful in command files. These, not given in the table earlier, are shown below. The first table is for actions that operate on name-value menus (A0 ... C19); the second table for command menu (X) actions.

Table 2: Command File Name-Value Actions

Text Arg	Action	Comments
Address	Addr	Button actions as discussed earlier.
Value	Val	Button actions as discussed earlier.
	A+1	Increment memory address, as discussed earlier.
	A-1	Decrement memory address, as discussed earlier.
NCols	FillC	Fill name-value menus beneath the one selected with consecutive addresses starting at the address contained in the selected menu.
	Oct	Display address offset and value in octal.
	Dec	Display address offset and value in decimal.
	Hex	Display address offset and value in hexadecimal.
	Num	Display value numerically.
	Sym	Display value symbolically.
	Search	Display value as an address symbol plus offset in the appropriate memory.
Value	SkipE	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. The command file must have at least one more action after the SkipE.
Value	SkipG	Skip if input text greater than the contents of the item in the name-value menu (unsigned compare).
Value	SkipL	Skip if input text less than name-value item.
Value	SkipNE	Skip if input text unequal to name-value item.
Value	SkipLE	Skip if input text less than or equal name-value item.
Value	SkipGE	Skip if input text greater than or equal to name-value item.

Table 3: Command File Command Actions

Text Arg	Action	Comments
Octal no.	Skip	Skip N following actions, where N is the value of the input text. An error occurs if the command file does not specify at least N more actions after the Skip.
.Tag	Skip	Skip following actions until one is encountered with the label ".Tag". An error occurs if ".Tag" is not found.
Octal no.	BackSkip	
.Tag	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 illegal for Return.).
	DisplayOn	Turn on the display, so that effects of subsequent commands can be observed. The display is initially off for a command file. This can also be done with control-D.
	DisplayOff	Turn off the display. This can also be done with control-O.
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. Note that TimeOut is *required* before the actions *'ed in the table on page 4 and is *illegal* before other commands; Midas will complain if you do not use TimeOut appropriately. The command file should include at least two more actions after the TimeOut. Command files written by WrtCmds will include necessary TimeOuts with a default of 30000 (~12 seconds); you will generally have to replace this default value with something more appropriate.
	Confirm	Supplies confirmation for the command which follows (which should be one of the commands requiring confirmation). If you omit a confirm in a command file, the user is queried for confirmation.
File name	OpenOutput	Opens an output file (default extension ".Report") on which text can be written.
File name	AppendOutput	Append to an output file (default extension ".Report")
	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> and "\" into a blank.
	WriteDT	Appends the current date and time to the output file.
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. Aborting will terminate all nested command files back to the top level.
--	DumpDisplay	Writes the current display image on the output file.
text	PrettyPrint	Evaluates text to a memory address, register name, or memory name; writes this name on the output file; then pretty-prints the value on the output file exactly as it would be pretty-printed on the comment lines if the item were displayed in one of the name-value menus and middle-buttoned.
File name	WriteState	Used by Midas initialization to create the Midas.Dtach and Midas.RunProg files--users shouldn't use this action.

9. Syntax of Command-file Actions

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

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

where the "[]" denote that the ".tag", input text, and ";comment" are optional. $" " denotes a sequence of one or more blanks or tabs (any characters with Ascii codes less than 40_8 except carriage return are equivalent to blanks). The sequence of characters excluding the comment must not exceed 99 characters.

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).

<action> is the text name for one of the actions (upper/lower case must match the definition).

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

> Note that if a *single blank* terminates <action> 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 reigster). However, one or more extra blanks will reset the input text, so the action is executed with null input text.

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

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

I think Midas will report all command file syntax errors intelligibly. Error handling works approximately as follows: Whenever Midas is watching the keyboard (which only happens during long-running actions such as "Go" or "Test"), control-Z will abort the current and all nested command files; the ShowError action in a command file also aborts all command files. If Midas detects an error while executing an action and queries, then "Continue" will abort the action in progress and continue with the next command file action. "Abort" will terminate the current command file, but not higher command files. However, a syntax error in the command file itself always aborts the current command file.

To find out what text should be put in command files, you can bug "ShowCmds" 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 "ShowCmds", bug "StopShow" (which appears only when "ShowCmds" is in progress.).

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

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" before actions which require confirmation and modify the "TimeOut" stuff which Midas uses to surround actions which might hang indefinitely (See the table given earlier for the actions that require this.).

Here is a sample command file:

```
L X Ld dg1;                          Equivalent to typing "dg1" and bugging "Ld" 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 SkipE 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
```

10. Registers and Memories Known to Midas

Table 4: Memories

Memory	*Width (octal)*	*Length (octal)*	*Notes*	*Comments*
IMBD	44	10000	4,5,6	Control store (via mufflers, manifold ops.--for testing only)
IMX	44	10000	6,7	Control store (absolute).
IM	100	10000		Control store (virtual).
TPC	20	20	1,2,6,7	Shows CIA for current task.
TLINK	20	20	1,2,6,7	Shows Link for selected task.
OLINK	20	20	1,2,3,9	Shows address of last call (if any).
ALUFM	10	20	6,7	0 and 16 smashed and restored by Midas.
RM	20	400	6,7	
STK	20	400	6,7	
STKX	20	77		= STK[STKP-address]
T	20	20	1,6,7	Waystation for A or Mar registers.
RBASE	4	20	1,6,7	Used in read-write of RM.
TIOA	10	20	1,6,7	
MEMBASE	5	20	1,6,7	
MD	20	20	1,9	
PIPE	40	20	8,9	Shows Pipe0 to Pipe5 (all signals high true)
BR	30	40	6,7,11	
BRX	30	4	11	Shows 4 BR words pointed at by MemBX
ROW	--	100	11	Shows 4 cols and Victim/NextV of a cache row
CACHEA	23	400	6,7,11	Length is 2000 with 16k cache
CACHED	20	10000	6,7,11	Length is 40000 with 16k cache
MAP	20	2^{16}	6,11	Length is 2^{16} or 2^{18} with larger map ic's
VM	20	2^{28}	6,11	
IFUM	40	2000	6,7,10	
DMUX	20	200	4,9	
DHIST	54	40	3,4	Discussed in the "DMux" section.
VH	40	40	3,4,9	Discussed in the "DMux" section.
$ABSOLUTE	10	2^{16}		Includes all state of microcomputer.
$ABS	20	2^{15}		$ABSOLUTE shown in 20 bit units
MSTAT	40	24	9	Low words of $ABSOLUTE shown symbolically
LDR	44	200	3	Holds microinstructions used by Midas.
MDATA	und.	10	3	BITS-CHECKED etc. for testing.
MADDR	40	14	3	LOOP-COUNT etc. for testing.
TASKN	0	20	3	Symbolic task definitions
DEVICE	0	400	3	Symbolic device address definitions

1. Task-specific
2. Virtual/absolute stuff applies
3. Fake memory--artifact of stuff in Midas
4. Readout via DMux, so value shown is correct in passive mode.
5. Resets the control section, so "Continue" from b.p. illegal.
6. Appears in Test menu.
7. Appears in TestAll menu.
8. SRN addressed.
9. Read-only to Midas.
10. Resets the IFU, so "Continue" from b.p. illegal.
11. Smashes the fault task pipe entry to access the item, so "Continue" from task 17 b.p. illegal.

Table 5: Registers

Register	*Width (octal)*	*Notes*	*Comments*
CPREG	20	2,3	Alto-baseboard interface register, freely smashed by Midas except in passive mode.
MIR	44	2,3,6	Microinstruction register, used ubiquitously by Midas.
IMOUT	44	1,6	Direct IM outputs
TASK	4	5	Discussed in the "Task-Specific Register" section.
Q	20	2,3	Waystation for write of registers on external BMux.
CNT	20	2,3	
SHC	20	2,3	Special tests for RF_, WF_, and FF-controlled shifts.
MEMBX	2	2,3	
STKP	10	2,3	
PROCSRN	4	2,3,6	*Must be 0 on a "Go" to operate memory system normally.*
MCR	20	2,3,6	Several bits are not testable; smashed and restored for memory stuff.
CONFIG	20	1	
TESTSYN	10	7	*Must be 200 (error correction on) or 0 (error correction off) to operate storage normally*
PCX	20	1,2,3	
INSSET	5	2,3	Shows the _Id count and instruction set (only the instruction set is writeable)
UPTIME	60	1	Time since boot-button pushed from microcomputer
TGLITCH	60	1	Time of worst power glitch seen by microcomputer
STROBE	20	5,7	Discussed in the "Passive Mode" section.
D1OUT	20	5,7	Discussed in the "Passive Mode" section.
EVCNTA	20	1	EventCntA register
EVCNTB	20	2,3	EventCntB register
ESTAT	20	2,6	Read-write error halt enables, read error conditions
AATOVA	20	5	Translate absolute address to virtual

1. Read-only to Midas.
2. Appears in Test menu.
3. Appears in TestAll menu.
4. Virtual/absolute stuff applies
5. Fake register--artifact of stuff in Midas
6. Readout via DMux, so value shown is correct in passive mode.
7. Write-only

Most registers and memories listed above correspond to ones discussed in the "Dorado Hardware Manual". Others are discussed in the sections which follow.

MDATA and MADDR memories contain words used to report or control the activity of the "Test" and "TestAll" actions discussed later. MADDR also contains DWATCH (used to control the DMux address for scoping), MIR-PES (error-reporting), and COM-ERRS (error-reporting), which will be discussed later.

TASKN and DEVICE are fake memories used to pass symbolic information from the assembler to Midas in the .mb file, as discussed in the "Dorado Microassembler" document. Their only purpose is to provide symbolic equivalents to task and device numbers for ease of debugging.

For approximately all registers and memories that contain 16-bit quantities, Midas will evaluate input of the form "m,,n", storing the value of "m" into bits 0:7 of the word and the value of "n" into bits 8:15.

> On Dorado, the items that accept "m,,n" are RM, Q, CNT, SHC, EVCNTB, T, STK, STKX, CACHED, VM, DMUX, and $ABS.

11. The IM Memory and Virtual Addresses

Because the placement transformations performed by MicroD make it difficult to correlate microstore locations with positions in microprogram source files, the Dorado and Dolphin Midas implementations use a map to transform virtual addresses produced by Micro into absolute microstore locations produced by MicroD.

Two memories, IMX and IM, each show the microstore. IMX is absolutely addressed; IM, virtually addressed. When you fire up Midas, IM is "empty"; when you load a microprogram, IM is filled with consecutive instructions from your source file, irrespective of where MicroD decides to place these; the value displayed for an IM address includes both the absolute address assigned to it, the microinstruction, and some other information discussed in the next section.

> In other words, if your microprogram is 10 words long, the meaningful part of IM is only 10 words long. In this case, if you examine IM addresses greater than 7, the printout will show an absolute address of 7777 and zeroes for the rest of the value.
>
> Midas will not allow you to modify the mapping between virtual and absolute addresses interactively--you can only do this by loading a microprogram.

To facilitate dealing with virtual/absolute correspondences, Midas has a mode switch that controls the way in which registers and memories that normally contain microstore addresses are handled. When you fire up Midas, the display is in absolute mode and the "Abs" action appears in the command menu; when you load a microprogram, the display switches to virtual mode and the "Virt" action appears in the command menu. Test actions will switch to absolute mode. The *current* mode always appears in the command menu.

In virtual mode, the display shows the virtual equivalent for the value in any register that normally contains a microstore address. When the value is outside the virtual memory, it prints as 7777. To find the absolute value in this case, you have to switch to absolute mode.

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

A fake register called AATOVA converts absolute addresses to virtual. For example, copying the value in some RM word into AATOVA will show the virtual equivalent; this is useful when return links are saved in RM words.

The general idea is that, if you suspect a hardware problem in the control section, you might work in absolute mode, but in all other situations when a program is loaded you will work in virtual mode, and the complications created by scrambled instruction placement will be concealed.

12. Registers and Memories that Contain Microinstructions

The MIR and IMOUT registers and the IMBD, IMX, IM, and LDR memories all contain microinstructions. A middle-button action over the value will print these symbolically on the comment lines.

The value for an IM address is shown as five fields on the display:

> two PE bits (PE020 and PE2141);
>
> Undef and Emu bits;

14_8-bit absolute address;
bits 0-21_8 of microinstruction (RSTK, ALUF, BSEL, LC, ASEL);
bits 22_8-41_8 of microinstruction (BLOCK, FF, JCN).

A "1" in PE020 indicates a parity error in bits 0_8-20_8 of the value; a "1" in the second bit means PE in 21_8-41_8. Both bits "1" normally indicates a breakpoint. Midas will store the data with bad parity, if you request it. Note that these are parity-bad bits; on a write, Midas will compute correct parity for each half of the microinstruction and xor that with the parity-bad bit; on a read, Midas will determine whether or not the location has correct parity and report accordingly.

The "Undef" bit is set when no absolute address is assigned to this virtual address--in this case the absolute address should print as 7777. The "Emu" bit tells the pretty-print routine to show the instruction as though it were being executed by the emulator (task 0).

IMX, IMBD, MIR, IMOUT, and LDR have a three-field printout in which the two PE bits are left-most followed by the left and right halves of the microinstruction.

IMX and IMBD each address the microstore absolutely and differ only in the way data is read and written. IMX is read and written by executing multi-cycle microinstructions that write the microstore from the BMux and read the data into Link. This requires that both ContA and ContB boards be present (plus ProcH and ProcL to compute parity). IMBD uses manifold operations to address and directly write the microstore and uses the muffler system to read out the microstore; this requires only ContB; however, the addressing method for IMBD makes continuation from a break impossible, so users should normally display IMX in preference to IMBD.

The IMOUT register contains the 44_8 DMux signals which are the direct outputs of the microstore, as addressed by the complicated stuff in the control section. At a breakpoint (t_0) IMOUT shows the bits that will be loaded into MIR at t_2, provided that the state of the branch condition does not change at t_1.

The LDR memory is an array in Alto core that contains microinstructions used by Midas when operating the hardware; it should ordinarily be of no interest to users, although the "LDRtest" action allows use of instructions stored in this memory for low-level hardware debugging.

Note that a bit pattern in LDR identical to one in IMX, IM, or IMOUT in general is *not the same instruction* because the ALUFM memory may contain different contents when the LDR instruction is executed. The pretty-print procedures account for this difference and show different stuff for these two cases. However, if you copy an LDR instruction into IM or IMX, watch out! In debugging regular microcode (i.e., any microcode that doesn’t test ALUFM itself), this incompatibility is usually avoided because ALUFM 0 and 16 are assembled with the "B" and "NOT A" alu operations, which are identical to the operations used by Midas.

Also note that the microinstruction pretty-print procedure does not have available all of the information that the microassembler had when you assembled your program, so the printout is not always beautiful. The following are deficiencies you should be aware of:

From the hardware manual, you will remember that the interpretation of the BLOCK bit depends upon whether or not the task executing the instruction is the emulator, and memory references are interpreted differently for the fault and emulator tasks than for io

tasks, so Midas will disassemble this stuff correctly only when it is able to deduce the task that executes the microinstruction. Midas does have available the Emu bit for instructions in IM, and if you pretty-print an IM address or an IMX or IMBD address that also appears in IM, Midas will be able to distinguish between emulator and non-emulator instructions; however, Midas cannot distinguish fault task microinstructions from other non-emulator instructions, so fault-task memory references will be pretty-printed erroneously. However, Midas very cleverly deduces the task for microinstructions in MIR and IMOUT in most cases, so the pretty-print will usually be correct for these.

Midas is not clever enough to figure out what will be in RBASE when an instruction is executed, so RM addresses from your program are not normally pretty-printed; Midas instead uses the generated names R0 to R17 for RM references.

There are many possible assembler macros that you might use to generate constants to control the shifter; for an instruction that does this, Midas will pick one of the forms, probably not the one you used in the source file.

Midas sometimes pretty-prints control clauses differently from the assembler. IFUJump's and IM/TPC read-write clauses are the same; the decision to print Return or CoReturn, LocBr or LocCall, LongBr or LongCall, GBr or GCall is dependent upon Midas deducing the virtual location for the instruction being printed and finding .+1 in the virtual space at .+1 in the absolute space, so this might be wrong sometimes. Conditional branches are always printed like "LocBr[addr1,addr2,BC]".

Modifying IM words in octal is inconvenient, so you will normally want to use the symbolic method below for patching IM.

Writeable registers and memories that contain microinstructions (MIR, IM, IMX, IMBD, and LDR) 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 RSTK, ALUF, BSEL, LC, ASEL, BLOCK, FF, JCN, PE020, PE2141, and EMUL. EMUL, the emulator mode bit affecting pretty-printing of the microinstruction is only defined for IM.

In addition to "field_value" clauses, Midas interprets the standalone clause RETURN, and several other items with "[]" enclosing a following argument. GO[va] (local branch), LONGGO[va] (long branch), and GCALL[va] (global branch) evaluate the argument enclosed in brackets and treat this as a virtual address in virtual mode or an absolute address in absolute mode; then they store a branch of the selected type in the JCN field of the microinstruction; IFUJUMP[n] evaluates n which should result a number in the range 0 to 3, and stores an IFUJump instruction in JCN. When you modify a microstore word (IM, IMX, or IMBD memories), Midas will error-check that the target for GO is, in fact, on the same page; Midas will always error-check that the argument of a GCALL is at a global address. Arguments to GO, LONGGO, and GCALL will usually be simple integers in absolute mode but may be expressions such as FOO+3, where FOO is an IM address, in virtual mode.

13. Task-Specific Registers

Midas treats all task-specific registers (T, RBASE, TLINK, OLINK, TPC, TIOA, MEMBASE, and MD) 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", "RBASE 20", etc.) and have that be interpreted as the register for the *currently selected task*. The currently selected task is the value in TASK; the TASK register is an artifact of Midas that is initialized to CTASK (i.e., to the "current task") at breakpoints.

In other words, when a microprogram halts at a breakpoint or because of a mouse-abort, CTASK is read from the DMux--suppose that it contains 6. This value is copied into TASK. If "T 20", "TLINK 20", etc. appear on the display, these will show values for task 6. The idea is that you can change the display for all eight task-specific registers by storing a new value into TASK. The task selected by TASK is also the one started by "Go", "SS", etc. as discussed later.

The hardware's LINK register, suppressed by Midas, is shown as the current task's TLINK word. The OLINK memory shows the absolute value in TLINK less 1. When microstore addresses are displayed in absolute mode, this is useless. However, in virtual mode OLINK will usually show the location that last did a CALL. This is useful in diagnostics which do BRANCH[ERROR], where ERROR is at a global call location. After one of these branches, OLINK shows the location that made the error branch, while TLINK shows an unrelated location.

14. BR Addressing Kludges

BR 40 is another addressing kludge used to represent the "currently selected" base register, or BR MEMBASE[TASK] (i.e., the BR location pointed at by MEMBASE for the currently selected task).

The BRX memory is another addressing kludge that allows the 4 BR words pointed at by MemBX to be displayed.

15. STKX Kludge

In debugging emulators, it is frequently desirable to view the STK entries relative to STKP rather than relative to STK 0 (i.e., relative to the top-of-stack rather than the bottom-of-stack). To aid in this, Midas defines STKX as an alternate memory for STK. STKX[n] shows STK[STKP-n], where valid values for n are 0 to STKP-1; hence, the top stack entries are STKX 0, STKX 1, etc.

STKX does not allow you to view entries on the wrong side of the stack pointer, and the display will preface those names with "~", indicating unreadable, if they appear on the display.

16. Memory System Registers and Memories

The cache, map, and storage arrangement may vary from one Dorado to another but Midas can deduce the configuration by reading the mufflers and looking at the CONFIG register; Midas does this automatically when you attach to a new machine or when you execute the "Config" action. Midas adjusts to the configuration by varying the lengths of its ROW, CACHEA, CACHED, and MAP memories and adapting its algorithms for reading and writing these.

Midas always uses task 17 (the fault task) and srn 1 (the fault task srn) to access BR, ROW, CACHEA, CACHED, MAP, and VM. Consequently, pipe entry 1 is smashed and (for CACHED and VM) MD is smashed, which may prevent continuing from a breakpoint, as discussed later.

ROW shows the cache flags and address bits in each of the four columns of a cache row and the victim and next-victim for the row on five consecutive lines of a display column. The length of the ROW memory is adjusted to the number of rows in the cache. Displaying an address in ROW is normally the most convenient way to view the cache; you can prettyprint the cache flags and address bits for each column independently, and this also shows the 16 data words in the associated munch (if any).

CACHEA is a memory of length equal to 4 times the number of cache rows; it shows the cache flags and address bits for a single entry in the cache. In a 100-row cache, the entries for the four columns in row *i* are CACHEA *i*, CACHEA *i*+100, CACHEA *i*+200, and CACHEA *i*+300. CACHEA is intended primarily for the "Test" and "TestAll" actions; on the display, it will usually be more convenient to look at ROW.

CACHED is a memory containing all the data words in the cache; word *m* in the munch for row *r* and column *c* is at CACHED 20_8*nrows**c* + 20_8**r* + *m*. CACHED is intended primarily for "Test" and "TestAll".

Addresses in the MAP memory are displayed with the *MapPE* and *PgFault* bits in a 2-bit field followed by *wp*, *dirty*, and *ref* bits in a 3-bit field followed by the 16-bit *ra* field on the display. When a MAP address is written, *ref* is zeroed and map parity is always written correctly; *dirtyb* (the copy of *dirty*) and *MapParity* are not readable (they appear in CONFIG in other situations).

VM accesses the virtual memory using Fetch_ and Store_ with the current contents of the map and cache; map and data error faults are not detected or indicated in any way, and the "RunRefresh" and "EnRefreshPeriod" clock enables must be true for storage to work properly. Midas sets the length of VM to the largest limit imposed by the map and cache geometries. Although VM appears in the "Test" menu, the user must setup the cache and map reasonably and select a suitably small sub-range of addresses in LOW-ADDR, HIGH-ADDR, and ADDR-INC before attempting to test VM. Midas uses a user-settable base register called VMBASE, initially 0, to offset the address specified by the user. VMBASE is an address in the fake MADDR memory.

In looking at VM, it is sometimes desirable to determine the MAP and ROW entries through which a VM word is accessed; if you middle-button any VM address, these will be displayed on the comment lines.

Midas does not provide any direct method of accessing storage; the user has to setup CACHEA and MAP with appropriate values and then use VM to do this.

Note: The code for accessing CACHEA and CACHED is complicated and unlikely to work unless the memory system is functional; these can be tested with "Test" and "TestAll" but the more basic "ProcVA" test, which exercises *VA* paths in the memory system, may be more helpful in isolating problems.

17. Memories and Registers Associated With the DMux

At those times discussed later, the 4000_8 DMux signals (or mufflers) are read from the hardware and stored in the first 200_8 words of a table. These are arranged so that hardware DMux address 0 corresponds to bit 0 of word 0 in Midas' DMUX memory, hardware address 17_8 to bit 17_8 of word 0, .., up to hardware address 3777_8 in bit 17_8 of word 177_8. Then the value on the BMux and the error status, which can also be read passively, are appended to the table. Finally, table data is rearranged, so that the DMUX memory looks as shown in the tables later.

Inside Midas associated with the DMUX memory are four separate tables, named as follows:

DMuxTab	current DMux readout
OldDMuxTab	previous DMux readout when running the simulator
DCheck	signals checked by the simulator
DWrong	errors detected during simulation

The last three tables are only significant when the DMux simulator is used by the "SimGo" and "SimTest" actions, as discussed later. In other words, when one of these actions halts, OldDMuxTab holds the t_0 DMux readout, DMuxTab the t_2 readout, and DWrong the errors that were detected in DMuxTab. DCheck is initialized by Midas to values that are reasonable for the boards that are plugged in, and the "Config" action also initializes DCheck to reasonable values; the user may manually modify DCheck, as discussed below, in order to disable checking of signals that are incorrectly simulated (This won't be particularly useful after the simulator is thoroughly debugged).

Normally, DMux addresses and registers derived directly from DMux readout (i.e., MIR, IMOUT, MCR, IMBD, DHIST, VH) show values taken from DMuxTab. However, the user may execute the "DMux" action with various button combinations to view the other three tables; the name printed for this action in the command menu will be "DMux", "DWrong", "DChk", or "OldDTab" according to which table is currently viewed. When the action is executed with the *right* (bottom) mouse button, OldDMuxTab values are viewed; both *left* and *right* buttons shows DCheck; *middle* button shows DWrong. The symbolic names of the first 11 errors in DWrong will also be printed on the comment lines when the middle button is released.

DMUX prettyprinting (middle button over value) of regular (DMuxTab or OldDMuxTab) values works differently from DWrong and DCheck pretty-printing. Regular printout of single-bit items shows symbolic names of "true" signals; "false" signals are not printed. In other words, low-true signals are printed when 0, high-true when 1. Multi-bit items (e.g., foo.0, foo.1, foo.2) are always printed (e.g., foo=3).

You should note that modified printout of DMUX also affects registers whose values are obtained by reading the DMux; this includes MIR, MCR, and IMOUT (but not IMBD). The DMUX memory itself and IMOUT are read-only except when DCheck is being shown. MIR and MCR are writeable when DMuxTab is viewed but read-only when OldDMuxTab or DWrong is viewed; writing modifies DCheck when DCheck is viewed.

The DHIST memory contains a DMUX bit address in bits 40_8 to 53_8 (displayed left-most by Midas) and a history of the last 40_8 values read from the DMux in bits 0 to 37_8 (displayed as the two right-hand fields by Midas). This memory may be useful in checkout of multi-state stuff in the memory and IFU sections of the machine when the DMux simulator is unable to detect

problems. Each time the DMux is read the 40_8-bit data field of each word in DHIST is left-shifted 1 and the new value brought into the low bit.

The VH memory provides another view of DHIST. Word 0 in VHIST shows the 40_8 DHIST signals at t_0, word 1 at t-1, word 2 at t_{-2}, etc.

When it is done reading the mufflers or done with a manifold operation, Midas loads the DMux address register with the value contained in DWATCH, an address in the MADDR memory. This means that during a "Go" or when Midas is not reading the mufflers, a scope probe attached to the DMux data line on the backpanel will show the DMux signal selected by the low-order 11 bits of DWATCH. However, if DWATCH contains 0, Midas will be turning control of the muffler/manifold system over to the baseboard at regular intervals, and the microcomputer will smash the DMux address.

18. Interface Registers

CPREG is one of the central interface registers used by the Alto in loading information into Dorado. It can be tested, but should not otherwise be of interest except in passive mode. Midas freely smashes the value in this register.

MIR is also special. It is loaded directly from the Alto and read via the DMux; Midas faithfully restores MIR after executing instructions.

19. Config

Midas automatically determines the hardware configuration when it connects to a particular dorado by means of DMux signals that it can read from each board. The configuration consists of the following parameters:

which boards are plugged in--debugging is frequently carried out with some boards disconnected;
Map ic size;
storage ic size;
cache size (4K words or 16K words);
whether the 16th bit in a cache entry is used as a parity bit or an address bit;
number of storage modules.

Midas automatically adjusts its length parameters for VM, CACHEA, CACHED, ROW, MAP, etc., enables and disables various tests in the Test and TestAll actions, and modifies the behavior of SimTest and SimGo according to which boards are plugged in.

The automatic determination of the hardware configuration should not fail, but if it does, the Config action can be executed to manually set the configuration by means of actions in a subsidiary menu. Manually controlling the configuration may also be useful when testing with SimTest or SimGo.

20. SetClk

The baseboard microcomputer presently initializes the clock to a 30 nsec period (= 60 nsec instruction cycle) when the boot button is pushed. The current clock period can be determined by pretty-printing the value of the CLKRUN DMux word which normally appears on the Midas

display.

The "SetClk" action allows the clock period of the mainframe to be specified from a subsidiary menu. You will probably be able to continue from a break after changing the clock speed, but Midas warns you that continuation is impossible.

21. Reset

The "Reset" action shows an elaborate subsidiary menu with many options. The options are: run enables for different stuff; parity-error enables for the different data paths that are parity-checked; and initialization of memories.

The general ideas that determined exactly how "Reset" is implemented are as follows: First, memories and registers should be reset only if they have to be for some reason. For example, memories that are parity-checked, such as T, RM, and STK, have to be reset to prevent parity errors when you start running a program; TIOA has to be reset in case some io device has variant behavior when TIOA contains its device number (building an io device that did this would be a poor idea); it is desirable to reset IMX and IFUM before loading a program, so that run-away branches and out-of-control programs will be trapped. However, other memories and registers such as RBASE, MEMBASE, Q, CNT, etc. need not be reset--your microprogram should contain code to initialize these, so Midas doesn’t have to.

Next, memories that require a long time to initialize, such as MAP (9 seconds now, 35 or 140 seconds with larger ic’s in the Map), should be *optionally reset* so that you won’t have to wait for their initialization unnecessarily.

Also, memories loaded by a microprogram (IM, IFUM, RM, ALUFM, and STK) should be *optionally reset*, if at all; if they are optional, you will be able to reset other parts of the machine without smashing your program. However, there does not seem to be any advantage in initializing ALUFM, so this memory is never initialized.

Each option is of an on-off form. The current state of the option is shown on the comment lines, while the other state appears in the command menu. The options as originally chosen are reasonable for a total reset, such as you would carry out at the onset of a "RunProg" command file; you may also want to turn on MAP initialization.

To carry out a reset, you bug the sequence of options you want, then bug the "Do-It" menu item.

When you bug "Do-It", initialization is carried out as follows (not exactly in this sequence since some initialization is done twice):

Run enables (RunRefresh and EnRefreshPeriod) are set as chosen;

Parity-error halt enables and MIRDebug are set as chosen; Midas remembers the halt enable settings so that they can be simulated for "SimGO" (discussed later) and remembers the setting of MIRDebug, so that it can warn against continue after breaks with MIRDebug true;

Manifold stuff used for testing IMBD is cleared;

Midas error counters MIR-PES and COMM-ERRS are cleared;

Hold and task simulators are cleared;

ALUFM 0 and ALUFM 16 are loaded with the "B" and "NOT A" alu controls needed by Midas;

The IFU is reset;

TestSyndrome is loaded for normal error-correction;

Several IOFetch_'es are done in task 2 to make sure that Asrn is .ge. 2 after power up;

Tasking is turned on;

Junk io, the fault task, and io devices are reset;

If MAP initialization is selected, each MAP address is loaded with Dirty and a pointer to the corresponding absolute page;

If MD initialization is selected, then CACHEA is loaded to map the first 4k (or 16k) of virtual memory, BR and CACHED are zeroed, and, for each task, the MD tag is reset, T and TIOA are zeroed, TLINK and TPC are loaded with 7777;

RM and STK are optionally zeroed;

If IM initialization is selected, then absolute mode is selected, IM is made empty, and every IMX address is loaded with "Branch[.], Breakpoint" except that 7776 is loaded with "Return, FreezeBC, Breakpoint" for the "Call" action;

If IFUM initialization is selected, then the Reschedule condition is turned off, and each IFUM address is loaded with the descriptor for a two-byte regular opcode with no operand, using MemBase 0 and RBase 1, starting at IMX 0.

MCR is loaded with NoRef and ProcSRN with 0;

The test control stuff BITS-CHECKED, LOW-ADDR, HIGH-ADDR, ADDR-INC are reinitialized;

After the reset is complete, Midas reads the DMux and checks the run-enable initialization, most of the control section initialization, and halt-enable initialization; if any failures are found, the errors are reported on the comment lines.

The "Go" action performs a subset of "Reset" prior to starting at a new address, as discussed later; parity-error halt enables can be modified without resetting anything else by writing an octal number into the ESTAT register.

22. Loading Programs

The "Ld", "LdSyms", and "LdData" actions are used to load micro-binary files into the machine. These actions are executed by first typing a list of file names (default extension ".mb") separated by commas, then bugging "Ld" or "LdSyms" (typing ";L" is equivalent to bugging "Ld"). These actions require confirmation by <cr>, "Y", or "." iff a previously-loaded program is being overwritten; in a command file where it is not known whether or not another program is being overwritten, a "Confirm" action should precede the load action, as discussed earlier.

"Ld" loads the entire .mb file--symbols into the Midas symbol table and data into the hardware.

"LdSyms" loads only the address symbols and IM mapping table from the .mb file. This may be useful when reattaching Midas to a machine that is already running a microprogram.

"LdData", (in command files but not available interactively), loads only the data blocks from the .mb file. "LdData" is provided so that a microprogram can be loaded without cluttering the symbol table--this is primarily for Midas initialization and should not be of frequent use to users.

> On Dorado, the DMUX, MADDR, MDATA, $ABSOLUTE, $ABS, and MSTAT memories are treated as exceptions by "LdData"--symbols for these are loaded anyway.

Midas uses several 1024-word core buffers (about 8 on Dorado Midas) and the Swatee file to manage its symbol table and virtual memory mapping information; the largest existing programs use 10 buffers for VM information and about 25 more (out of 64 available on Swatee) for symbols. For nearly all symbol and VM accesses, Midas will reference only one or two symbol blocks, so there should be no appreciable slow down when handling large programs.

> The symbol table management algorithm used by Midas is an extremely fast merge that works well when the symbol table is nearly empty at the onset of a load but suffers somewhat from block fragmentation when the initial symbol table has many items.
>
> To avoid fragmentation, don't load one microprogram on top of another--use "RunProg" to reset the symbol table, then do the "Ld". 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.
>
> Midas uses the symbol table in two ways: looking up the value of a symbol, requiring at most one disk access; and searching for the symbol in a particular memory which best matches a value, requiring at most one access for RM, BR, DEVICE, and TASK address symbols, or at most two accesses for IM address symbols; the best matching value for addresses in all other memories is determined by scanning every block. Searching every block requires about (.22 seconds * no. symbol blocks) - (.15 seconds * no. blocks in core) or about 4.7 seconds for the largest program thus far. However, since best matches for the five most important memories are obtained quickly, it will rarely be necessary to wait for a search.

In most situations where a "Ld" is going to be done, many other actions will also be carried out to setup the display appropriately for the program and to initialize the hardware by doing "Reset" or whatever. For this reason, you will ordinarily want to define a command file that does all these other actions as well as the "Ld" and you will ordinarily do "RunProg" on this command file; direct use of "Ld" in the command menu will be rare.

23. Dump and Cmpr

Both "Dump" and "Cmpr" require confimation 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 Dolphin 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 "Cmpr" (no problem on Dorado or Dolphin).

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.)

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

> In microprograms, avoid loading initial values into memory words modified during execution. The usefulness of "Cmpr" is enhanced when programs are clean, because no fictitious errors will be reported.
>
> For diagnostics, "Cmpr" can report what has been smashed when something goes off the deep end--this has frequently been helpful.
>
> Following system microcode crashes, "Cmpr" may provide the only clue about the nature of an intermittent storage failure.

24. Brk and UnBrk

On Dorado breakpoints are created by deliberately storing bad parity in both halves of a microinstruction. Since double parity failures are highly unlikely, there is usually no ambiguity between deliberately set breakpoints and hardware failures.

Since Dorado does not halt until t_2 of the instruction containing a parity failure, the break will occur *after* the instruction containing it has been executed.

Since the two parity-bad bits are part of the value displayed for an instruction, it would be possible to insert or remove a breakpoint by examining an instruction and storing 3 or 0 into the parity-bad field; however breakpoints are inserted and removed often enough to warrant an easier method for doing this. The "Brk" and "UnBrk" actions are provided for this purpose.

"Brk" inserts a breakpoint in the IM or IMX address typed on the input text line. 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.

"UnBrk" removes a breakpoint. If no text is typed, the address defaults to the breakpoint that caused the last program halt or to the address of the last breakpoint inserted. You will normally find it faster to type "address;K" or ";K" to remove a breakpoint.

25. Go, SS, Proceed, OS, and Call

These are actions that result in the microprocessor executing instructions from the control store starting at the selected address; "SimGo", which will be discussed later, also does this. Each of these accepts an input argument (optional except on "Call") that must evaluate to an IM or IMX address; a simple number is defaulted to an IMX address in absolute mode or an IM address in virtual mode. If the optional argument is omitted, Midas will continue from the last break.

When you start at a new address, the value in TASK (lower left-hand corner of the normal display) is the task activated. TASK is initialized to the value in CTASK (i.e., to the task for which an instruction was about to be executed) when Dorado halts or when you abort. You must change TASK on the display to initiate execution for a different task.

The distinctions among these actions are as follows:

"Go" and "Proceed" will start the machine running and wait either for it to halt or for you execute the "Abort" or "Dtach" actions which are displayed during the "Go". When going or proceeding at a new address (as opposed to continuing from the last break), "Go" will reset io devices and the control section, while "Proceed" does not do this; in other respects these actions are identical.

"SS" (single-step) executes one microinstruction.

> Although "Go" and "SS" (single-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:", an alternative to "SS"; "Proceed" is only executable by typing "address;P." Similarly, ":" is equivalent to a continue-"SS" and ";G" or ";P" (proceed) to a continue-"Go".

"OS" (opcode-step) keeps single-stepping the machine until either you execute the "Abort" action, a halt condition occurs, or an IFUJump has been executed. In other words, it simulates a "Go" with repeated single-steps, but stops after the next IFUJump. This is intended to facilitate debugging emulators that use the IFU.

> There are some hardware restrictions on single-stepping discussed in the next section. The most serious of these is that it is illegal to single-step across an instruction that does Fetch_ and _Md. Since this is expected to be common in emulators, there will be many times when OS doesn't work.

"Call" allows a microprogrammed subroutine to be called with an optional argument passed in T. By convention both the microassembler and the "Reset" action plant a "FreezeBC, Breakpoint, Return" microinstruction at IMX 7776. A call is initiated by typing "SUBR(ARG)" or "SUBR()". This causes ARG (if any) to be evaluated and stored in T; LINK is loaded with 7776; then "SUBR;G" is done. If the subroutine returns (to 7776) Midas prints an appropriate message.

> Note that subroutines called this way need not start at "call" locations in the microstore because Link is loaded prior to jumping to the starting address.

"SimGo" (simulated-go) is a variation of single-step that keeps single-stepping the machine until either a halt condition occurs or the DMux consistency checker finds an error, as discussed later.

Before stepping or going at a new address (as opposed to continuing or proceeding), Midas carries

out an extensive reset sequence, as follows:

IO devices and fault task are reset.

Ready flipflops, CTASK, CTD, etc. are cleared by executing "TaskingOn", "No-op", and then "Goto[7777], Block" for each task. Your microprogram should probably load IMX 7777 with some instruction to handle bogus task wakeups.

TPC is set to 7777 for every task except the one being started.

Memory "tag" mechanism is NOT reset.

The IFU and Reschedule condition are NOT reset.

When the microprocessor halts after a breakpoint, due to an error, or because you aborted, Midas prints the location of and reason for the halt and saves the information that it needs to continue. The form of the printout is "task:address". Subsequently, if you attempt to continue, Midas restores the hardware as nearly as possible to its state at the break before continuing.

The primary error indicators for a break are in ESTAT; Midas analyzes these and other DMux signals such as "Task2Back", "Task3Back", "_MDSaved", etc. and pretty-prints a message about the reason for halting and the task that executed the instruction that caused the halt.

There are many complications surrounding Midas' ability to restore the state of the program, after doing other things, so that continuation is possible. These are discussed in the next section. When these complications are insurmountable, "passive mode" may be used as discussed later.

26. When Registers are Read/Written--Restrictions on Continuing

When a microprogram halts at a breakpoint or due to a mouse-halt, Midas has two objectives: to read the contents of registers and memory addresses so that they may be shown to the user, and to be able to continue from the interrupt or breakpoint. The methods for reading machine state are detailed in the "Dorado Debugging Interface" document and outlined here.

Midas first reads the DMux (which includes MIR, MCR, and some other items), BMUX, and ESTAT (error status); these are read first to capture their values before they change. Since all of these items are readable without issuing any clocks to the Dorado microprocessor, Midas can still continue execution of the microprogram in ordinary situations. In passive mode (discussed later), these are the only items which Midas reads from the hardware.

In active (i.e., normal) mode, Midas next executes a no-op, clears the hold and task simulator, does 30 no-op's, and then saves values of (current task) registers as follows: LINK, T, Q, TIOA, STKP, ALUFM 0, ALUFM 16, RBASE, MEMBASE, PROCSRN, and RM 0; these might get smashed while reading registers that the user has put or will put on the display.

Finally, Midas reads all registers displayed going top-to-bottom through the name-value menu lines and left-to-right through the columns within each line. In passive mode, only those items whose values were obtained passively will be updated; others will be marked with a "~" indicating that Midas couldn't obtain the current value. In active mode, many microinstructions will be executed to correctly address each item, route its value onto BMux, where Midas can read it, and then restore registers smashed while doing this.

When Dorado is not running, Midas loads ALUFM 0 and 16 with the "B" and "NOT A" alu operations, and TPC (i.e., CIA) is always in a smashed state. If one of these three items is displayed, the value in the Alto static is read; if written, the static is written. The value in the static is not written into the hardware until either a "Go", "SS", "OS", etc. action occurs or the "Dtach" or "RunProg" actions are executed. ALUFM 0 and 16 are effectively untestable from Midas (sorry). TPC will get read for the new task and restored for the old task whenever Midas has to do a SelectTask, as discussed in the "Dorado Debugging Interface" document. Midas has no trouble testing TPC, but if you examine a particular TPC register several times on the display, there is no guarantee that the values displayed will be ones independently read from the hardware.

With the exception of these three items and the DMux, Midas always reads values from the hardware--other saved values are only used for restoration purposes. In other words, if "SHC" is displayed 10 times, it will be read 10 times from the hardware.

MIR, MCR, Q, T, RBASE, MEMBASE, TLINK, STKP, RM 0, and PROCSRN are smashed and restored while reading other stuff; these are read from the hardware independently each time they appear on the display, but Midas might rewrite these registers from the saved values, so if one of these isn't working correctly, the exact nature of the failure may be obscured.

Several memories and registers are "always updated" when they appear on the display, which means that they will be reread at frequent intervals by the Midas main loop, and if the value has changed the display will be updated. The UPTIME and TGLITCH registers and the MSTAT memory, which show items continuously recomputed by the baseboard microcomputer, are treated this way; and COMM-ERRS and MIR-PES (in the fake MADDR memory), which report errors detected by the Midas hardware interface, are always updated.

Values in other registers and memories are only reexamined when you do some "dirty" action. When you write a value into some register on the display, for example, Midas tries to restore any other registers and memories that were clobbered as a side effect; then it rereads the DMux and all registers on the display.

There are a number of situations that may prevent continuation from a breakpoint or interrupt; Midas warns you about some of these when you try to continue but does not warn you about others. Some of the ones that Midas does not warn you about are as follows:

The machine stopped at t_2 of an instruction that both started a new fetch and either read Md onto A or B or used Md in a shift-and-mask operation; the value of Md for the new fetch will be erroneously used in completing the Md read.

The break occurred at t_2 of an instruction doing a dispatch.

The break occurred immediately after an IM or TPC read instruction--the value read will be garbage if you continue;

You were using the hold simulator--Midas resets the hold simulator at breakpoints;

Your microprogram was using the muffler/manifold system--Midas smashes the DMux address and resets some of the manifold stuff at breakpoints;

Input/output tasks were not serviced properly due to the delay at the breakpoint, so these are not continued correctly;

Your microprogram is relying upon the exact timing of the memory system to write the cache flags for a reference--the moment will have passed when continuation occurs (There are probably other situations when the memory system is operated in unusual ways that will prevent continuation.).

Some situations that Midas does warn you about are as follows:

> You have displayed some address in BR, BRX, ROW, CACHEA, CACHED, MAP, or VM; Midas will use task 17 and pipe entry 1 to access these, and if the break occurred in task 17, Midas will warn you that continuation is impossible because Pipe entry 1 and (for CACHED and VM) task 17 MD are smashed.
>
> Some address in IFUM is displayed; Midas has to reset the IFU to read IFUM and will warn you that continuation from a breakpoint is impossible.
>
> Some address in IMBD is displayed; Midas has to reset some of the control section to access IMBD.
>
> You broke on or single-stepped across an instruction that did both a Fetch_ and either a T_Md or RM/Stk_Md; if you continue, data from the new fetch rather than data from the preceding fetch will be used to complete the T_Md or RM/Stk_Md operation.
>
> A breakpoint on or single-step through an instruction that does NewPC_ is illegal (??).

27. Hardware Failure Reporting

Midas checks for several kinds of hardware errors and reports them in MIR-PES and COMM-ERRS, which are addresses in the MADDR memory; these are shown in the upper right-hand name-value menus by the normal Midas display. MIR-PES is shown on the display as two 16-bit fields; the first field counts parity errors detected in MIR[0:20] and the second, parity errors in MIR[21:41]. MIR-PE’s is zeroed when you start Midas, "Dtach", or "Reset", or when you start a "Ld". Whenever Midas loads a microinstruction into MIR, it checks for good parity in MIR before executing it and counts MIR-PES if the parity is no good; however, even if the parity is bad, Midas goes ahead and executes the microinstruction. If any MIR parity errors occur during a load, the message "**MIR-PE’s occurred**" is printed on the comment lines after the load; however, except for that message, Midas does not print any special messages after these errors--the user will have to notice when MIR-PES changes at other times.

COMM-ERRS is also shown as two 16-bit fields. The first field counts glitches in the "Stopped" line, which Midas samples repeatedly during "Go" (The serial 1 Dorado seemed to report "Stopped" when the microprocessor did not have any reason for stopping, so some glitch detection software was added to Midas to detect this situation.); the second field counts microcomputer timeouts. Midas initializes these error counters to 0 after initially connecting to a Dorado, during "Reset", and during "Dtach". Midas allows about 2 msec for the baseboard microcomputer to service interrupt requests; if this timeout is exceeded, the right-hand field of COMM-ERRS is counted.

Midas also shows a number of hardware conditions collected by the baseboard microcomputer; these include power supply information summarized in PROBLEMS, OUTOFSPEC, BADSUPPLYSPEC, and TGLITCH as discussed later.

28. Hardware Checkout Facilities

Midas checkout facilities fall into the following categories:

Observation	Observe registers and signals invisible to the microprogrammer (DMux stuff, print routines, passive mode).
Poking	Trying out elementary actions to observe what happens (T1, T2, T3, Poke stuff).
Testing	Exercise various hardware sections, verifying that they work correctly or reporting the nature of failures (Test, TestAll, SimGo, SimTest, LDRtest).
Scope loops	Repeatedly do something to observe failures with the scope (RepGo, RepSS, RepT2, Fields, HWChk, test actions).
Diagnosis	Relate failures to particular hardware components (SimGo, SimTest).

The LDRtest action must be preceded by the "Debug" command file (in the "RunProg" submenu), which loads the LDR memory addresses needed for LDRtest. The "Debug" command file should not be needed in any other cases.

With diagnostic microprograms, you can use the PROC, CONTROL, MMC, MMD, MMX, IFUD, DSKETH, and DSP command files (in the submenu put up by RdCmds) to display DMux addresses for various hardware sections.

29. Parity-Error Scanning

The "PEscan" action scans memories and reports parity errors. It presents a submenu consisting of "Scan-and-report" and "Scan-for-totals" actions followed by the names of the memories that can be scanned for errors. The user interacts with the submenu, selecting and deselecting memories to be scanned; then he bugs either "Scan-and-report" or "Scan-for-totals".

> On Dorado, the memories that can be parity scanned are IMX, RM, STK, IFUM, CACHEA, CACHED, and MAP.

"Scan-and-report" will sequence through all the words in the selected memories, reporting on the comment lines the first 20 addresses that have parity errors and the total number of parity errors for each memory. "Scan-for-totals" reports only the parity error count for each memory.

In general, very long memories such as main storage are not included in the "PEscan" submenu because Midas cannot scan them fast enough to report results in a reasonable time.

30. Testing Directly From Midas

"Test" and "TestAll" allow the target machine to be tested directly from Midas. Although diagnostic firmware can test faster and more thoroughly than is practical from Midas, Midas direct testing permits the hardware to be checked out well enough to get basic diagnostics loaded and started. On Maxc1, which had no direct testing in Midas, many hardware failures of the "nothing works" variety were harder to fix than on Maxc2 and Dorado, where Midas test software is

available. However, on Dolphin and M68 implementations of Midas, the test features in Midas are of doubtful usefulness because the hardware is accessed through communication with a small "kernel" microprogram that only works when most of the hardware is functional.

Data patterns for test actions are determined from the first subsidiary menu, as follows:

Table 6: Test Data Pattern Actions

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

The CYC0, CYC1, and SEQUENTIAL patterns vary according to the size and arrangement of the data vector for the item being tested. CYC0, for example, starts off with leading 1’s and a 0 in the right-most bit of the data vector. The 0 is shifted left (bringing in 1’s to its right) each iteration; when the 0 is shifted out of the left-most bit in the data vector, the vector is reinitialized to leading 1’s and a 0 in the right-most bit. The CYC1 pattern is like CYC0 with 1’s and 0’s interchanged. The SEQUENTIAL pattern is initialized to 0 and is incremented by 1 in the right-most bit of the data vector each iteration.

This treatment of CYC0, CYC1, and SEQUENTIAL patterns is conceptually correct for items that are described inside Midas by dense, left-justified data vectors whose bits are displayed left-to-right on the screen. Most, but not all, items are handled this way.

> On Dorado, the exceptions are as follows: IMX, IMBD, and MIR have the parity-bad bits displayed left-most but stored internally right-most in the data vectors. The parity bits do not participate in determining the data pattern for CYC0, CYC1, and SEQUENTIAL patterns; i.e., the two parity-bad bits will always be tested with 1’s (i.e., bad parity) for CYC0 or always with 0’s (i.e., good parity) for CYC1 and SEQUENTIAL patterns.
>
> ALUFM, CACHEA, MAP, BR, and MCR have holes between bit 0 and the right-most bit of the data vector. The CYC0, CYC1, and SEQUENTIAL patterns for these are generated as though these holes didn’t exist. I.e., ALUFM has an 8-bit data vector in which bits 1:2 are unused; CACHEA has flags in bits 0:3 and VA[4:n] in subsequent bits, but the leading bits of VA are not actually stored in the address section for most cache configurations, so the unstored bits are a hole; MAP has RP in bits 16:31 of the data vector and various flags in bits 12:15, so bits 0:11 are a hole; BR uses 4:31 of the data vector, so bits 0:3 are a hole; and MCR uses 0:15 with several unused bits in its interior.

Testing is controlled/described by 12 addresses on the display as follows:

Table 7: Test Items in the Name-Value Display

SHOULD-BE	On a failure, the correct data; after control-C or Abort, the next pattern.
DATA-WAS	On a failure, what the data was; after control-C or Abort, the data read last time.
BITS-CHECKED	Mask of bits checked (see below).
BITS-PICKED	Union of bits that should have been 0 but were erroneously 1 during testing. This accumulates failure information when you continue a Test using <escape> or <cr>.
BITS-DROPPED	Union of bits that should have been 1 but were erroneously 0.
LOOP-COUNT	32-bit iteration count at which failure occurred or after which the test was aborted.
NFAILURES	32-bit count of test failures.
Memory tests only	
LOW-ADDR	
HIGH-ADDR	
CURRENT-ADDR	
ADDR-INC	32-bit addresses: 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. CURRENT-ADDR contains the last address tested.
ADDR-INTERS	Intersection of address bits where failures were detected.
ADDR-UNION	Union of address bits where failures were detected.

SHOULD-BE, DATA-WAS, BITS-CHECKED, BITS-PICKED, and BITS-DROPPED are addresses in the MDATA memory; LOOP-COUNT, NFAILURES, LOW-ADDR, etc. are addresses in the MADDR memory. These two memories (which are tables in Alto storage) exist on all versions of Midas that implement the test actions.

The handling of the MDATA memory is complicated by the fact that items in this memory have to be shown in the same format as the memory or register being tested. This is accomplished as follows: When the selected test item is different from the last, the width and print-format of MDATA are set to be identical to the new item; in this case BITS-CHECKED is initialized to test all bits in the new item. Then when the test is aborted or halts due to a failure, the display of BITS-CHECKED, etc. is identical to that of the item tested. The user may then modify BITS-CHECKED and continue, restart, or free-run the test, as discussed below; in this case the item tested is identical to the last item tested, so BITS-CHECKED is not reset.

The handling of MADDR is also tricky. ADDR-INC is allowed to be any value except 0; if it is 0, Midas will reset it to 1 before testing. When HIGH-ADDR is initially greater than the largest legal address in the memory, it is reset to memlength-1 prior to testing. Then if LOW-ADDR is greater than HIGH-ADDR, it is reset to 0 before testing. When the selected memory differs from the last item tested, and when the length of the memory is less-than-or-equal to 10000_8 words long, Midas will reset LOW-ADDR to 0 and HIGH-ADDR to memlength-1 prior to testing. This is done because a common operational error is failure to reset the address range when switching from one memory test to another. However, Midas does not reset the address range for very long memories because they are normally tested with small address ranges that cannot be predicted in advance--full-length testing of long memories from the Alto is so slow as to be impractical.

"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:

Table 8: Other Test Actions

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_
ProcVA	Tests BR+Mar via DummyRef_

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

<cr> will continue a register or memory test that has halted but will free-run the test rather than halting on the next failure. While free-running, LOOP-COUNT and NFAILURES are reported continuously on the display, and BITS-DROPPED, BITS-PICKED, ADDR-INTERS, and ADDR-UNION accumulate failure information. When you stop the test by bugging "Abort" or typing control-C, the accumulated failure information is displayed in these registers.

"TestAll" 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 "TestAll" whenever the hardware is in a suspicious state.

31. LDRtest

On Dorado and Maxc2, the "LDRtest" 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. "LDRtest" allows these to be executed in non-standard sequences to beat on particular hardware problems.

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

When two, three, etc., up to five LDR addresses are typed, a test loop occurs whereby CPREG is loaded with the next data pattern, the first instruction is executed with UseCPReg 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.

32. Scope Loop Actions: Fields, RepGo, RepSS, RepT2

The "Fields" action 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 sub-menu is replaced by various values. The second subsidiary menu allows the value in the selected field to be incremented, decremented, and shifted.

"RepGo" 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.

"RepSS" repeatedly single-steps the microprocessor at the address typed on the command line.

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

On Dorado, "RepT2" endlessly executes the instruction in MIR and reloads that value into MIR. Unlike "RepSS", "RepT2" doesn't issue extraneous clocks while looping, so it is ordinarily more convenient for scoping.

33. HWChk

The "HWChk" action puts up a submenu that contains several test and scope loop actions. Once started, one of these actions runs until you abort it; the iteration count will be in LOOP-COUNT when the test is aborted. The HWChk submenu currently contains the following actions:

"Read-DMux-Signal" requires a non-zero value in DWATCH; a scope loop is generated in which the DMux address selected by DWATCH is strobed out to the hardware and then the value read. A count of the number of times the value is 0 and the number of times it is 1 are showed on the comment lines. Microcomputer DMux reading is disabled during this action.

"Read-All-DMux" repeats the following sequence indefinitely: (1) Execute an almost-random microinstruction as in SimTest; (2) Read all 4000_8 DMux signals three times, accumulating in DWrong the union of signals which had inconsistent readout. A count of the number of inconsistent signals is displayed. Microcomputer DMux reading is disabled during this action. Also, signals which legitimately may change value (primarily Ethernet and Disk signals) are not reported as inconsistencies. This action can be used to test the reliability of the DMux and strobe data paths, which are known to become unreliable for sufficiently long Midas cables. It can also be used as a scope loop for observing DMux data paths. If this action reports no inconsistencies, then one can be fairly confident that the simulator will report Dorado hardware failures rather than failures in the Midas communication data paths.

"Connect-Disconnect" first evaluates the input text line, which must contain a valid Dorado serial number (0 to 377_8). A scope loop is generated in which Midas alternately connects to the selected serial number and to that serial number xor 377_8 (= disconnects). A count of successful and unsuccessful connects is displayed on the comment lines.

"Alto/MC-control" generates a scope loop in which the Alto and the microcomputer alternately are given control of the muffler/manifold system.

34. DMux Consistency Checker

The DMux consistency checker, or simulator, used with the "SimGo" and "SimTest" actions examines all of the DMux signals (or mufflers), checking for inconsistencies. The simulation verifies consistency of signals from the previous readout (call this "t_0") to the current readout (call this "t_2").

In all cases, only *passively-accessible* DMux signals and BMUX and ESTAT are involved in the simulation--registers that can be read only by issuing clocks to the hardware are not checked.

The simulation subroutine behaves differently based upon the time at which the DMux was read (t even or t odd) and upon whether or not the DMux readout at t_{n-2} is available. Currently, the simulator is only called by "SimGo" and "SimTest", and for these the simulation subroutine is always called with the t_0 and t_2 DMux tables.

The operation of the simulator is reported in and controlled by five tables, each containing one bit for every DMux signal:

OldDMuxTab	DMux readout for t_0.
DMuxTab	DMux readout for t_2.
DCheck	mask of bits whose simulated values are to be checked for errors.
DWrong	mask of signals whose DMuxTab values do not agree with the simulation and whose mask in DCheck is 1.

Setup for the simulator begins during initialization, when the "Config" action is executed. At that time, DCheck is initialized to reasonable values for the hardware configuration. In other words, if a particular section of the machine is not in the chasis, then the signals in that section cannot be checked, so they are zeroed in DCheck. In addition, other signals whose simulated values depend upon signals from the missing section cannot be checked. "Config" sets up DCheck so that only signals which can be checked will be examined for error. Finally, "Config" also zeroes the DCheck bits for signals known to be simulated incorrectly.

A simulation step consists of the following parts:

1) Copy DMuxTab into OldDMuxTab.

2) Single-step the Dorado; then read the DMux into DMuxTab.

3) Copy DMuxTab into DWrong.

4) Execute the simulation program which will predict many signals as functions of values in OldDMuxTab and DMuxTab. The predicted values overwrite values in DWrong. Unsimulated signals are not modified in DWrong.

5) DWrong _ (DWrong xor DMuxTab) & DCheck.

6) Stop and report failures if any bits in DWrong .ne. 0.

"SimTest" is executed with IOReset, RunRefresh, and EnRefreshPeriod false. It loads MIR with a randomly chosen microinstruction (except that some illegal microinstructions are weeded out--presently, the Output_, UseDMD, MidasStrobe_, and IFUTest_ functions are illegal; also, the Block bit in the next microinstruction is chosen to equal whatever was coming from IMX just

before t_2 of the last microinstruction to avoid screwing up the control section); then it reads the DMux and steps the microinstruction through t_2. This is repeated, and after each repetition the previous and current DMux readout are checked for consistency.

"SimGo" is similar, but a microprogram stored in IM is executed one step at-a-time rather than random microinstructions; there are no illegal microinstructions for SimGo. When a diagnostic or other microprogram is known to fail, it can be run full speed up to a breakpoint a little before the sequence that fails; then the program can be continued with "SimGo" which might pinpoint the hardware failure. However, since RunRefresh and EnRefreshPeriod are false during "SimGo", any microprogram that uses Storage or the Map might not run correctly. "SimGo" continues until either a simulation error is detected or ESTAT contains a halt condition; the halt conditions for "SimGo" are identical to those for "Go" (The halt conditions can be modified by the user with the "Reset" action.).

For the most part, mufflered signals in the different hardware sections relate to control paths rather than to data paths, so the consistency checker will be less effective in finding failures in data paths. However, Midas register and memory tests and diagnostic firmware can usually pinpoint data failures, so this limitation is not too serious.

The ContA/B, ProcH/L, MemC/D/X, and IFU sections are presently simulated.

How to Interpret Simulator Failures

When the simulator detects one or more failures, it reports a message like "2 DMux errors". You can find out which signals are believed inconsistent by executing the "DMux" action in the command menu with the middle button. When the middle button is released, the names of the first 11 signals that were incorrect are printed on the comment lines; each name is followed by a suffix such as "/A" indicating the section in which the error was detected; possible suffices are:

/B Baseboard	No signals are currently simulated.
/A ContA	
/B ContB	
/L ProcL	
/H ProcH	
/I IFU	
/C MemC	
/D MemD	
/X MemX	
/K Disk controller	No signals are currently simulated.
/E Ethernet controller	No signals are currently simulated.
/V Display controller	No signals are currently simulated.

The next step is to display the DMux words associated with one of the hardware sections that failed; this is done by executing the "RdCmds" action and selecting the command file that displays that section (PROC, CONTROL, MMC, MMD, MMX, IFUD, DSKETH, or DSP).

Then find the source for a signal that failed in the hardware drawings; you will probably be able to deduce its dependency upon other DMux signals and can then determine where the failure occurred. You can view the signals relevant to the simulation by viewing the OldDMuxTab or DMuxTab signals on the display, as was discussed in the "Memories and Registers Associated

With the DMux" section.

Normally, DMux addresses and registers derived directly from DMux readout (i.e., MIR, IMOUT, MCR, IMBD, DHIST, VH) show values taken from DMuxTab. However, the user may execute the "DMux" action with various button combinations to view the other three tables; the name printed for this action in the command menu will then show the selected memory. The button combinations for this are as follows:

DWrong	middle button
DCheck	left and right buttons
OldDMux	right button
DMux	left button

Only when DCheck is displayed is it legal to write words in the DMux memory; the other forms are read-only. DCheck can be modified to remove signals from the checking process (or to add them back).

35. Poking: T1, T2, and T3

The "T1", "T2", and "T3" actions allow the instruction currently in MIR to be executed exactly as though it were spliced into the execution flow of the program. The DMux is read after t_1, t_2, or t_3 of the instruction, then, for "t1" and "t3", the machine is clocked once more (to t_2 or t_4). MIR is restored after execution.

36. Passive Mode

Passive mode suppresses automatic readout of registers that require clocks to be issued by Midas. This allows scope observation without interference from automatic parts of Midas.

Midas implements three "states" called active, prepassive, and passive. The command menu always prints the current state; bugging "active" will change the state to "prepassive"; bugging "prepassive" will change to "passive"; and bugging "passive" will change to "active"--in other words, these three states are sequenced through in a "ring."

In active mode, Midas will jam instructions into MIR and execute them to obtain the contents of various Dorado register or memory words or to restore registers incidentally smashed while doing something else; as discussed earlier, there are some situations when continuation is impossible after doing this, and some hardware problems are difficult to observe when Midas is interfering to this extent.

PrePassive mode is identical to active mode, but if you start the machine with "Go," "SS,", "SimGo", "SimTest", or whatever, then Midas will automatically flip into passive mode the next time the machine halts.

When you enter passive mode from the keyboard action, the state of the hardware is restored as though it were about to continue from a step or breakpoint and TASK is restored to its value at the last step or breakpoint. After this, no further clocks are given to the hardware except those explicitly initiated by the user.

After becoming passive, Midas doesn't update registers on the display unless their values can be read without issuing clocks. Since only DMux locations (includes MIR, IMOUT, IMBD, MCR, TESTSYN, PROCSRN, TASK) can be read without clocks, only their values change while passive.

Further, if you display a new non-passive register on the display, its value will not be read from the hardware and garbage will be displayed as the value.

Items on the display for which the displayed value is doubtful will be flagged with a "~" as discussed earlier.

Similarly, only registers whose values can be modified without issuing clocks may be written while passive--these are MIR, CPREG, STROBE, and D1OUT (plus the artificial registers and memories). Midas rejects attempts to modify other registers on the display. Of the writable registers, only MIR, CPREG, and IMBD can be read, and only MIR can be read passively. Consequently, if you write into CPREG, STROBE, or D1OUT by clicking the mouse over its value, it will be written but the display will show the contents of a static, not something read from the hardware--since other parts of Midas don't update the statics, the value displayed only means something immediately after the write.

The command menu is drastically altered while passive; only actions which can be executed while passive are shown.

"Update" reads the machine state actively and then becomes passive again.

While passive, "SS" and "Go" at new addresses work as usual, so extra clocks are issued to do these. However, "SS" and "Go" to continue a program do not issue any extraneous clocks--all of the setup to continue took place at the time passive mode was entered; or after a step or breakpoint, no clocks are issued to readout the machine state, so it is possible to continue simply by modifying Stop, SetRun, and SetSS.

To do the most primitive kind of debugging while passive, it is expected that users will work as follows: First, the POKE command file will be executed to become prepassive and display STROBE, D1OUT, and CPREG, not ordinarily on the display. The user will then either do a Go or SS, becoming passive at the break, or will bug prepassive to become passive immediately. Next, MIR and CPREG will be written by modifying the displayed value. Then the Clock and Control registers and Strobe can be manipulated by storing values into STROBE and D1OUT.

STROBE is displayed as two fields and D1OUT as three fields; when storing into these, you must partition the input into fields as well. For STROBE the two fields are the address field (3 bits) and data field (9 bits). Storing into STROBE will give a three-step strobing sequence using the value of address and data you have selected. For D1OUT the three fields are the Strobe bit, address, and data. (*Note*: The DMux will be read after writing MIR, but it is not read after writing CPREG, STROBE, or D1OUT, used for lowest level debugging of the Midas communication interface.).

37. MIRdebug Feature

During ordinary operation, an IMX parity error or breakpoint halts Dorado after t_2 of the instruction affected by the parity error. Since MIR is loaded at t_2, the MIR value with bad parity has been overwritten when the machine stops, so if the path between the microstore and MIR is experiencing intermittent failures, it will be difficult to diagnose what has gone wrong.

To aid checkout in this case, the control section has a debugging aid called MIRdebug, which will disable the clock to MIR at t_2 of an instruction with bad parity. When this aid is enabled, MIR will still contain the bad data after the error-halt. This feature can be invoked by enabling "MIRdebug" in the sub-menu put up by the "Reset" action. If a parity error halt occurs while MIRdebug is enabled, then Midas will print the value read from IMX[CIA] so that you can compare this with the value in MIR on the display to find out which bits are not propagating from IMX into MIR.

The liability of this debugging aid is that you will not be able to continue from a breakpoint or IMX parity error halt, so you should not enable MIRdebug unless you are searching for this type of hardware failure.

38. Failure Diagnosis

Some actions to analyze test failures and report the hardware components involved have been considered, and are likely to be implemented for IMBD, IMX, IFUM, RM, and STK.

Storage, Map, and cache failure analysis programs are essential, but should be provided outside Midas.

39. Baseboard Microcomputer Stuff

The Alto can communicate directly with the Baseboard section of any Dorado connected to it through its Diablo Printer interface as detailed in the "Dorado Debugging Interface" document. It can:

(a) Select any one of the connected Dorados;
(b) Control the muffler/manifold system or give control to the baseboard microcomputer
(c) Interrupt the baseboard microcomputer;
(d) Pass information to the microcomputer through CPREG; and
(e) Read 8 bits of information from the microcomputer through the DoradoIn mechanism.

Midas does (a) and (b) during initialization and during the "Dtach" action, as discussed in the "Starting Midas" section; Midas uses (c), (d), and (e) together with a large set of communication conventions to exchange information with a program running on the baseboard microcomputer.

$ABSOLUTE is the fundamental microcomputer memory, 8 bits wide. It contains all information other than mufflers which Midas can access on the baseboard. This memory is divided into a RAM (addresses 0 to 777_8 or 0 to $1FF_{16}$) and a ROM (addresses 100000_8 to 177777_8 or 8000_{16} to $FFFF_{16}$). The amount of ROM is adjustable; current Dorados have storage only for addresses 140000_8 to 177777_8. The microcomputer stores its internal registers and other information of

interest in the lowest approximately 200_8 bytes of $ABSOLUTE.

The $ABS memory is identical to $ABSOLUTE except that it shows the information 16 bits wide rather than 8 bits wide. The MSTAT memory and the UPTIME and TGLITCH registers present special information from $ABSOLUTE in human-readable form. The initial Midas display shows this information. The information in the main display is easily interpretable once you get used to it, or you can pretty-print the values in expanded form.

UPTIME is a six-byte counter that counts time in 102.4 msec ticks, starting at 0 after a boot. TGLITCH holds the value that was in UPTIME at the end of the last power transient in which some voltage or current was outside its specified range. Midas prints these items like "1 day 2:23:32", i.e., in standard day hours:minutes:seconds form, when they appear on the display.

MSTAT contains the current, maximum, minimum, and first values for each of the four power supply voltages and currents and for temperatures on each of the 12 boards in the main frame. The "first" items are recorded at completion of the power-up sequence; the maximum (minimum) items are initialized to 0 (infinity) and then increased (decreased) when the current values exceed (are less than) the previous maximum (minimum); the current values are updated repetitively by the microcomputer main program. Each word in MSTAT contains four one-byte items: Voltage and Current items have one byte for each of the four power supplies, and the printout is in volts or amperes; temperature items are shown in degrees centigrade, and there is one of these for each of the 12 boards in the main frame, arranged four-per-word in MSTAT.

Midas repetitively updates displayed values for UPTIME, TGLITCH, MSTAT, and the PROBLEMS, OUTOFSPEC, and BADSUPPLYSPEC addresses in $ABSOLUTE that appear on the display.

The microcomputer can update power supply information and temperatures for itself and for ContB irrespective of whether or not it controls the muffler/manifold system, but other board temperatures can only be determined when the microcomputer controls the muffler/manifold system. Board temperatures can only be read when the -5 volt power supply is up.

When Dorado is running (i.e., SetRun is true), Dorado controls the muffler/manifold system and neither Midas nor the microcomputer can access it; when the boot button is pushed or when Midas detaches from a particular Dorado, the microcomputer controls the muffler/manifold system, so its main program can read temperatures unless that Dorado is running. Finally, when Midas is attached to a machine, it controls the muffler/manifold system but releases control to the baseboard at regular intervals *unless DWATCH is non-zero*; when DWATCH (an address in the fake MADDR memory) is non-zero, Midas will retain control of the muffler/manifold system and arrange to select the muffler signal whose number is in DWATCH whenever possible.

The main breaker switch on the Dorado environmental carrier (near the floor) will turn on the 5 volt supply and one fan. The baseboard microcomputer automatically boots itself from ROM whenever this main breaker is turned on, and then follows (approximately) the sequence discussed below to bootstrap the rest of Dorado into operation:

turn on disk logic power and wait 20 seconds;
turn on disk spindle motor and wait 20 seconds;
turn on fans and +12, -5, and -2 volt supplies and wait 20 seconds;
initialize machine status information (discussed below);

load and execute Dorado boot microcode, which loads and starts the system microcode.

During this sequence and afterwards, the microcomputer reports what is happening on its status light, which will repeat a sequence of blinks followed by a pause during any problem condition. The light sequences are interpreted as follows (the light blink information is also in PROBLEMS on the Midas display):

1 blink	boot in progress	Wait for disk, power supplies, stable clock, etc. This is normal during a power-on or 3-push boot sequence, as discussed below.
2 blinks	boot failed	Tried to boot Dorado microcode but didn't get the appropriate handshake.
3 blinks	transient power problem	Power supply voltages went bad, now good again (details in BADSUPPLYSPEC on display; TGLITCH shows the time when this transient ended; MAXVOLTS or MINVOLTS reveals the magnitude of the transient). Presently, only voltage variations cause this condition, but eventually amperage variations may also cause it (MAXAMPS and MINAMPS on the display).
4 blinks	power problem	Voltages are now out-of-spec (details in OUTOFSPEC and VOLTS on the display); eventually amperages may also cause this condition (AMPS on the display).
5 blinks	powered down	Get this after powering down with a seven button-push sequence (see below).
6 blinks	over temperature	Powered down because the temperature on some board went over 60^{o} C (MAXTEMP, MAXTEMP+1, and MAXTEMP+2 show details).
7 blinks	can't get CP control	Can't get muffler/manifold control because Midas is hogging it.
solid green	AOK	Bootstrap sequence is believed to have completed successfully, there have been no occurrences of the error conditions indicated by 3 through 6 blinks., and the baseboard microcomputer gets regular CP control.
light off	microcomputer down	power is off, the microcomputer crashed, or the light burned out (unlikely because LED's are long-lasting)

As mentioned above, the main breaker switch on the Dorado turns on only the +5 volt supply and one fan. The basic "on" state minimizes power consumption, and enables the baseboard microcomputer to turn on other power supplies and fans.

When the user depresses the boot button on the back of the keyboard for at least 0.2 seconds and not more than 2.5 seconds, the microcomputer records an event called a "button push"; depressing for less than 0.2 seconds or longer than 2.5 seconds is ignored; depressing for longer than 2.5 seconds will nullify the entire boot sequence. The microcomputer will count button pushes until 1.5 seconds has elapsed with the button up; then it will carry out an action as follows:

1 push--ignored; standard emulators also monitor the raw boot button and may take some action. Currently, they go through a software bootstrap sequence under the assumption that currently loaded microcode is correct and running normally.

2 pushes--stops and resets the microprocessor and starts it in task 0 with tasking turned

off at location 1067_8 (which is "InitMap" for the Alto emulator). This is intended to be a forcible restart of currently-loaded microcode.

3 pushes--load IM from the Dorado boot loader and start it running as for 2 pushes. This initiates a complete microcode bootstrap sequence, similar to the automatic power-on boot. It assumes nothing about the current state of the machine.

4, 5, or 6 pushes--same as 3 pushes.

7 pushes--power down; all of the supplies and fans except the 5 volt supply and 1 fan are powered down in a safe sequence, taking about 30 seconds. The 5 volt supply can then be shut down from the main breaker switch; *avoid turning off the main breaker switch until the microcomputer has completed shutting down the disk and other supplies because you will invoke the power failure safety circuits in the disk drives.*

8 or more pushes--ignored; the user does this when he makes a mistake and wants to start over.

Under normal conditions, in response to the boot button being pushed or the main breaker being turned on, the microcomputer will show 1 blink for about 60 seconds and then show solid green; if Midas attaches to the machine, the status light will usually show solid green, but will show 7 blinks (Midas hogging CP bus) during long-running Midas actions.

Note: if the Dorado was powered down at the onset of a button push sequence, any number of pushes from 1 to 3 will do a total (3 push) boot.

Note: it is unsafe to turn on disk power when the -5 volt, -2 volt and +12 volt supplies are on because, the resulting power surge will blow breakers in the building wall circuits. For this reason, be sure to power down the Dorado logic supplies (7 push sequence discussed above) before turning on the disks; then go through the complete power up sequence with a normal boot sequence (1 to 3 pushes).

Note: Since the microcomputer uses the +5 volt supply itself, it will crash if that supply fails and might subsequently auto-boot itself if the +5 volt supply starts working again. An over-temperature shutdown never turns off the +5 volt supply.

Note: If the Dorado is alreay powered-on, the full bootstrap sequence can also be initiated by pressing the reset button on the front panel of the Dorado chassis (inside the cabinet, if the Dorado is cabinet-mounted). However, if the Dorado is in the shut down state, pushing the reset button has no effect.

For full user-level details on booting, consult "Dorado Booting" by Ed Taft ([Indigo]<DoradoDocs>DoradoBooting.press) and "Dorado Booting--Implementation" by Ed Taft ([Indigo]<DoradoDocs>DoradoBootingImple.press).

40. Command Files Used With "RdCmds"

At the time this was written, the following command files were in use:

Table 9: Command Files

poke	show CPREG, STROBE, and D1OUT in the right column and become passive for manual hardware poking.
normal	restore "normal" Midas display with the baseboard voltages, temperature, and currents in the right display column.
tests	restore "normal" Midas display with the hardware testing items in the right display column.
svcrash	write the Midas display followed by a pretty-print of all DMux registers on the file Crash.Report.
proc	show ProcH/L DMux signals in middle column.
control	show ContA and ContB DMux signals in middle column.
mmc	show MemC DMux and other signals in middle column.
mmd	show MemD DMux signals in middle column.
mmx	show MemX DMux signals in middle column.
ifud	show IFU DMux signals in middle column.
dsketh	show disk and ethernet controller DMux signals in middle column.
dsp	show display controller DMux signals in middle column.
tpc	show 20_8 TPC registers in middle column.
tlink	show 20_8 TLINK registers in middle column.
alufm	show 20_8 ALUFM locations in the middle column.
t	show 20_8 T registers in middle column.
rbase	show 20_8 RBASE registers in middle column.
membase	show 20_8 MEMBASE registers in middle column.
tioa	show 20_8 TIOA registers in middle column.
md	show 20_8 MD registers in middle column.
brlo	show BR 0 to BR 17 in middle column.
brhi	show BR 20 to BR 37 in middle column.
hist	show first 20_8 DMux histories in right column.
vh	show first 20_8 DMux vertical histories in middle column.

41. DMux Signal Assignments

Table 10A: Control Section DMux Signals

*Original addresses 0-77 and 260-377 are from ContA, 100-257 from ContB. Midas rearranges many signals for convenient viewing. The second column shows the way Midas displays them.

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
0	Stop	CJNK0	0	0	Stop	Always
1	preStartCyclea			1	preStartCyclea	Always
2	dStartCycle			2	dStartCycle	Always
3	Phase0			3	Phase0	Always
4	Phase4			4	Phase4	Always
5	RWTPCorRWIM			5	RWTPCorRWIM	Always
6	BigBDispatch			6	BigBDispatch	Tn ge 2 & switch’
7	Dispatch			7	Dispatch	Tn ge 2 & switch’
10	WIM’			10	WIM’	Always
11	RIM’			11	RIM’	Always
12	WTPC’			12	WTPC’	Always
13	RTPC’			13	RTPC’	Always
14	FF=Notify’			14:17	0	
15	FF=MulStep					
16	FF=BDispatch					
17	FF=BigBDispatch					
20:37	CIAInc[0:15]	CIAINC	1	20:37	CIAInc[0:15]	Tn ge 1
40:57	CIA[0:15]	CIA	2	40:57	CIA[0:15]	Tn ge 2
60	* CABlock	BNT	3	60:73	0	
61:70	* bFF[0:7]			74:77	Bnt[0:3]	Tn ge 2
71:74	* JCN[0:3]					
75:77	* bJCN[4:6]					
100:117	* MIR[1:16]	PENC	4	100:113	0	
				114:117	bPEnc[0:3]	Always
120:121	--	TNIA	5	120:121	--	
122:137	TNIA[2:15]			122:137	TNIA[2:15]	Unless return or IFUJump
140:141	--	BNPC	6	140:141	--	
142:157	BNPC[2:15]			142:157	BNPC[2:15]	Never
160	CBTempSense	CTASK	7	160:173	0	
161	bSWd’			174:177	CTASK[0:3]	Tn ge 2
162	* IMLH					
163	* bRSTK.0					
164	* bdRSTK.0					
165	* bdIMLH					
166	* bdIMRH					
167	* bdJCN.7					
170:173	CTASK[0:3]					
174:177	CTD[0:3]					

* Midas extracts the 44 MIR and 44 bdIM signals and arranges these as registers (MIR and IMOUT). This information resides in DMuxTab in the peculiar MIR-loading format discussed in the "Dorado Debugging Interface" document, but is viewed by users in the standard IM format.

Table 10B: Control Section DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
200:217	* bdxx for IM[1:16]	NEXT	10	200:203	0	
				214:217	Next[0:3]	Always
220:237	* bdxx for IM[17:32]	CTD	11	220:233	0	
				234:237	CTD[0:3]	Tn ge 1
240:243	CS[0:3]'BDb	RA	12	240:243	CS[0:3]'BDb	Always
244:245	RAQuad[0:1]i			244:245	RAQuad[0:1]i	Never
246:257	RA[1:10]			246:257	RA[1:10]	Always
260	Call	TOPE	13	260	0	
261:277	ToPE[1:15]			261:277	ToPE[1:15]	Always
300	* bJCN.7	CJNK1	14	300	Call	Usually
301	* IMRH			301	bSWd'	Always
302	GND			302	GND	Always
303	LocalBr'a			303	LocalBr'a	Always
304	IFUNext'a			304	IFUNext'a	Always
305	LongJump'a			305	LongJump'a	Always
306	Return'a			306	Return'a	Always
307	CondBr'a			307	CondBr'a	Always
310	bFFok'c			310	bFFok'c	Always
311	FA=0'			311	FA=0'	Always
312	FA=1'			312	FA=1'	Always
313	bDoCBr			313	bDoCBr	Never
314	FF=UseDMD			314	Link_BMuxa	Tn ge 1
315	FF=TOffIsOK			315	B_Link'	Tn ge 1
316	RIMorRTPCdly			316	RIMorRTPCdly	Tn ge 2
317	MulStep			317	MulStep	Tn ge 2 & no switch
320	FF=TaskingOn	FFEQ	15	320	FF=TaskingOn	Always
321	FF=TaskingOff			321	FF=TaskingOff	Always
322	FF=MidasOn			322	FF=MidasOn	Always
323	Link_BMuxa			323	0	
324	FF=WriteLink			324	FF=WriteLink	Always
325	FF=Link_CPReg			325	FF=Link_CPReg	Always
326	FF=ReadLink			326	FF=ReadLink	Always
327	B_Link'			327:331	0	
330:333	Bnt[0:3]			332	FF=UseDMD	Always
334:337	bPEnc[0:3]			333	FF=TOffIsOk	Always
				334	FF=Notify'	Always
				335	FF=MulStep	Always
				336	FF=BDispatch	Always
				337	FF=BigBDispatch	Always

* Midas extracts the 44 MIR and 44 bdIM signals and arranges these as registers (MIR and IMOUT). This information resides in DMuxTab in the peculiar MIR-loading format discussed in the "Dorado Debugging Interface" document, but is viewed by users in the standard IM format.

Table 10C: Control Section DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
340:343	pNext[0:3]	CJNK3	16	340:343	0	
344	Next=0			344	Next=0	Always
345	CTask=0			345	CTask=0	Always
346	PEncGtTrueNext’			346	PEncGtTrueNext’	Always
347	PEncLtTrueNext’			347	PEncLtTrueNext’	Always
350	StopTasks			350	StopTasks	Tn ge 2
351	PEnc=CT’			351	PEnc=CT’	Always
352	TPCBypass			352	TPCBypass	Tn ge 2
353	PreEmpting’			353	PreEmpting’	Always
354	bHoldA			354	bHoldA	Always
355	RepeatCurz			355	RepeatCurz	Always
356	bSwitch’a			356	bSwitch’a	Tn ge 2
357	bSwitchUp’			357	bSwitchUp’	Tn ge 2
360	--	READY	17	360	--	
361:377	Ready[1:15]			361:377	Ready[1:15]	
		MIR	166:171	--	MIR[0:35] in MIR format	
		IMOUT	172:175	--	bdIM[0:35] in MIR format	

Table 11: BaseBoard DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
2200:2207	ClkRate	CLKRUN	110	2200:2207	ClkRate	Never
2210	ECLup			2210	ECLup	Never
2211	EnRefreshPeriod’			2211	EnRefreshPeriod’	Never
2212	IOReset’			2212	IOReset’	Never
2213	RunRefresh			2213	RunRefresh	Never
2214	MASync			2214	MASync	Never
2215	TBaseTempSense			2215	0	
2216:2217	--			2216:2217	--	

Table 12A: Processor Section DMux Signals

*Processor DMux addresses (400 to 777) are arranged so that the first 10_8 in each group of 20_8 are from ProcH, the last 10_8 from ProcL. Signals are frequently duplicated (one from each board). Midas does not rearrange any signals from the processor section.

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
ALUB	20	400:417	alub	if driven from BMux or from constant
ALUA	21	420:437	alua	Tn ge 1 driven from small constant & not shift
ABCON	22	440	MarMuxAEn’	Tn ge 1
		441	AmuxEn’	Tn ge 1
		442:443	Amux0 to 1	Tn ge 1
		444	IOBout	Tn ge 1
		445	BmuxEn’	Tn ge 1
		446:447	Bmux0 to 1	Tn ge 1
		450:457	=440:447	Tn ge 1
PERR	23	460	EMU’	Always
		461	CkMdParity’	Tn ge 2
		462:463	--	
		464	IOPerr	Never
		465	MdPerr	Never
		466	RmPerr	Never
		467	TmPerr	Never
		470	StkSela	Always
		471	StkSelSaved	Never
		472	IOBoutSaved	Tn ge 2
		473	_MDSaved	Never
		474:477	=464:467	Never
SHMV	24	500:517	shmv	Pmux odd or shift’
MAR	25	520:537	MAR.0’ to MAR.15’	Tn ge 1 driven from processor, no shift, Tn eq 2 bits 8:15 when driven by IFU, Tn eq 2 when not driven
--	26	540:557	--	
PRFA	27	560	Last=Curr’	Tn ge 2
		561	Curr=Next’	Always
		562	Shift’	Always
		563	IOBin’	Tn ge 1
		564:566	FA=0’a to FA=2’a	Always
		567	FA=3’	Always
		570:577	=560:567	As above

Table 12B: Processor Section DMux Signals

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
SCCON	30	600	--	
		601	RepeatCurrC	Always
		602	Holda	Always
		603	LdTaskSim’	Always
		604	FFshift’	Always
		605	ShcWriteEn’	Tn ge 1
		606	LoadCnt’	Always
		607	PropCnt’	if DecCnt is false
		610	--	
		611:612	= 601:602	Always
		613	LdHoldSim’	Tn ge 1
		614:616	=604:606	As above
		617	DecCnt’	Always
QPDCON	31	620	QshiftL’	Tn ge 1
		621	QshiftR’	Tn ge 1
		622	RmaskEn’	Tn ge 1
		623	LmaskEn’	Tn ge 1
		624	ShiftBitsEn’	Tn ge 1
		625:627	Pmux0 to 2	Tn ge 1
		630:633	=620:623	Tn ge 1
		634	ALUFWriteEn’	Always
		635:637	= 625:627	Tn ge 1
ALUCON	32	640	Pdata.00	Tn ge 1 if source is ALU barring shifter
		641	Pdata.04	Tn ge 1 if source is ALU
		642	TIOAWriteEn’	Tn ge 1
		643	TIOABypass	Always
		644	MBWriteEn’	Tn ge 2
		645	MBBypass	Always
		646:647	MBMux0 to 1	Tn ge 1
		650	aluCin	Never
		651	Pdata.08	Tn ge 1 if source is ALU
		652	Pdata.12	Tn ge 1 if source is ALU
		653:656	aluF0 to 3	Never
		657	aluM	Never
NEXTCL	33	660:663	LastNext.0’ to .3’	Tn ne 1
		664:667	CurrLast.0’ to .3’	Tn ne 1
		670:677	=660:667	Tn ne 1
RADDR	34	700:703	Task2Back.0’ to 3’	Tn ge 2
		704:707	Task3Back.0’ to 3’	Tn ge 2
		710:713	RbWadr.0’ to 3’	Sometimes Tn ge 2
		714:717	RbWadr.4 to 7	Tn ge 2

Table 12C: Processor Section DMux Signals

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
STKRB	35	720	BCWriteEn’	Tn ge 2
		721	Cnt=Zero’	Never**
		722	Ioatta	Never**
		723	ResEqZero’	Never**
		724	ResLtZero’	Never**
		725	ALUCarry	Never**
		726	Overflow’	Never**
		727	RmLtZero’	Never**
		730	RBaseBypass’	Tn ge 1
		731	SelRBaseWadr’	Always
		732	RBaseWriteEn’	Tn ge 1
		733	BumpRBase	Always
		734	BumpRSTK	Always
		735	StkPMux1	Always
		736	StkPWriteEn’	Always
		737	RmOdd’	Never**
RTSB	36	740	ReSchedWrEn’	Always
		741	NextMacro	Always
		742	RbWriteEn’	Tn ge 2
		743	RbSelMd	Tn ge 2
		744	RbBypassDly	Never**
		745	TbWriteEn’	Tn ge 2
		746	TbSelMd	Tn ge 2
		747	TbBypass	Tn ne 1
		750	StkPSaveEn’	Tn ge 1
		751	StkError	Never**
		752:757	= 742:747	As above
PJUNK	37	760	FFok’a	Always
		761	--	
		762	NextData’	Always
		763	B_Ext	Always
		764	FF.0mem	Always
		765	FF.1mem	Always
		766	RisIFdata	Always
		767	TisIFdata	Always
		770	FFok’b	Always
		771	_MD	Always
		772	_MDI	Always
		773	B_Ext	Always
		774:775	SbTskDly.0’ to 1’	Never
		776	RisIFdata	Always
		777	TisIFdata	Always

Table 13A: MemC DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
1000	ProcVA.04	PVAH	40	1000:1003	0	
1001	true			1004:1017	ProcVA.04 to 15	Tn ge 2 & no clk
1002	WantCHdly’					
1003:1017	ProcVA.07 to 19					
1020	MemB.0	PVAL	41	1020:1037	ProcVA.16 to 31	Tn ge 2 & no clk
1021:1022	ProcVA.05 to 06					
1023	MemB.1					
1024:1037	ProcVA.20 to 31					
1040:1047	Aad.0a to 7a	MAPAD	42	1040:1046	0	
1050:1057	MapAd.1 to 8			1047:1057	MapAd.0 to 8	Never**
1060	dVA_Vic					
1061	ForceDirtyMiss					
1062	UseMcrV					
1063	DisBR					
1064	DisCflags					
1065	DisHold					
1066	NoRef	HIT	43	1060:1066	0	
1067	MiscPCHP’			1067	MiscPCHP’	Tn ge 2
1070:1071	ColVic.0 to 1			1070:1071	ColVic.0 to 1	Sometimes
1072	HitColVA.par			1072	HitColVA.par	0 on miss
1073	HitColDirty			1073	HitColDirty	0 on miss
1074	Hita			1074	Hita	0 on ForceMiss if Tn ge 2
1075:1077	MemB.2 to 4			1075:1077	0	
1100:1101	Victim.0’ to 1’	HOLD	44	1100:1101	0	
1102:1103	NextV.0’ to 1’			1102	true	
1104	MiscHold’			1103	WantCHdly’	Tn ge 1
1105	MDhold’			1104	MiscHold’	Tn ge 2
1106	RefHold’			1105	MDhold’	Tn ge 2
1107	BLretry			1106	RefHold’	Tn ge 2 & not ForceMiss
1110	AwasFree’			1107	BLretry	If forced to 0
1111	Dbusy			1110	AwasFree’	Tn ge 2
1112	DbufBusy			1111	Dbusy	Tn ge 2
1113	AtookST			1112	DbufBusy	Tn ge 2
1114	SomeExtHold’			1113	AtookST	Tn ge 2
1115	Afree’			1114	SomeExtHold’	On StkError % CHoldReq
1116	StartMap’			1115	Afree’	if EcHasA
1117	AwantsMapFS’			1116	StartMap’	Always
				1117	AwantsMapFS’	Always

Table 13B: MemC DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
1120	Store_InA	PAIR	45	1120	Store_InA	Tn ge 2 % EcHasA
1121	IoStoreInA			1121	IoStoreInA	Tn ge 2 % EcHasA
1122	Map_InPair’			1122	Map_InPair’	Tn ge 2
1123	FlushInA			1123	FlushInA	Tn ge 2 % EcHasA
1124	PrefetchInA			1124	PrefetchInA	Tn ge 2 % EcHasA
1125	IfuRefInA			1125	IfuRefInA	Tn eq 2 % EcHasA
1126	IoRefInA’			1126	IoRefInA’	Tn ge 2 % EcHasA
1127	CacheRefInA			1127	CacheRefInA	Tn ge 2 % EcHasA
1130	MapAd.0			1130	0	
1131	PrivRefInPair			1131	PrivRefInPair	Tn ge 2
1132	VicInPair’			1132	VicInPair’	Tn ge 2 sometimes
1133	FSinPair’			1133	FSinPair’	Tn ge 2 sometimes
1134	bEcHasA			1134	bEcHasA	Tn ge 2
1135	KillIfuRef			1135	KillIfuRef	Always
1136	_PrVArow			1136	_PrVArow	Tn eq 2
1137	PairFull’			1137	PairFull’	Tn ge 2
1140:1143	PipeAd.0 to 3	PIPEAD	46	1140:1143	PipeAd.0 to 3	Tn eq 2
1144:1145	CacheConfig[0:1]			1144:1145	CacheConfig[0:1]	Never
1146:1147	PageConfig[0:1]			1146:1147	PageConfig[0:1]	Never
1150:1157	--			1150:1157	--	
		MCR	57	3760	dVA_Vic	Never
				3761	ForceDirtyMiss	Never
				3762	UseMcrV	Never
				3763:3764	Victim[0:1]	Never
				3765:3766	NextV[0:1]	Never
				3767	DisBR	Never
				3770	DisCflags	Never
				3771	DisHold	Never
				3772	NoRef	Never
				3773:3774	0	
				3775	WakeOnCL	Never
				3776	ReportSE’	Never
				3777	NoWakeups	Never
		AAD	161	3660:3663	0	
				3664:3673	Aad.0a to 7a	Never
				3674:3677	0	
		MEMB	162	3700:3712	0	
				3713:3717	MemB.0 to 5	Never

Table 14A: MemD DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
1200	SinD.00	MEMD0	50	1200	SinD.00	Never
1201	CD.00			1201	CD.00	Never
1202	D0in.00			1202	D0in.00	
1203	D1in.00			1203	D1in.00	
1204	EcSout.00’			1204	EcSout.00’	
1205	EcInD.0			1205:1206	0	
1206	Dbuf_’					
1207	--			1207	--	
1210	D.00			1210	D.00	
1211	dMD.00			1211	dMD.00	
1212	D1BCE’c			1212	Fout.00	
1213	WriteD1’d			1213:1217	0	
1214	DontWriteMDM					
1215:1217	Dad1.10b to 12b					
1220	D0BCE’c					
1221:1222	Dad.00f to 01f	DAD	51	1220:1221	Dad.00f to 01f	
1223:1231	Dad.02’c to 08’c			1222:1230	Dad.02’c to 08’c	
1232	Dad.09’			1231	Dad.09’	
1233:1235	Dad0.10c to 12c			1232:1234	Dad0.10c to 12c	
1236	D0ACE’c			1235:1237	Dad1.10b to 12b	
1237	WriteD0’e					
1240	F_D	FD	52	1240	F_D	Tn ge 2
1241	D_Dbuf			1241	D_Dbuf	Tn ge 1
1242	Sout_D			1242	Sout_D	Tn ge 1
1243	Fout_D			1243	Fout_D	Tn ge 1
1244	D_CD			1244	D_CD	Tn ge 1
1245	Md_D			1245	Md_D	Tn ge 1
1246	MakeMDM_D’			1246	MakeMDM_D’	Always
1247	bFastD_Dbuf			1247	bFastD_Dbuf	Always
1250	Fout.00			1250	Dbuf_’	Tn ge 1
1251	DadH_’			1251	DadH_’	Tn ge 2
1252	DontLoad1			1252	DontLoad1	Always
1253	GenPh1			1253	GenPh1	Tn ge 2 & EnEcGen
1254:1257	--			1254	DontWriteMDM	Tn eq 2
				1255:1257	--	

Table 14B: MemD DMux Signals

DMux Address (Octal)	Signal Name	Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
1260:1263	MDMad.0’ to 3’	EC	53	1260:1263	0	
1264	StartEcChk’			1264	StartEcChk’	Always
1265	StartEcGen’			1265	StartEcGen’	Always
1266	D1ACE’c			1266	0	
1267	--					
1270	EcInD.1			1267:1270	EcInD.0 to 1	Never
1271	WordInError’			1271	WordInError’	When DisableEc true
1272	DisableEc’			1272	DisableEc’	Never**
1273	ChkPh1			1273	ChkPh1	Tn ge 2 & preEcEn
1274	ChkPh4’			1274	ChkPH4’	
1275	ChkLastPh6’			1275	ChkLastPh6’	
1276	DoubleError’			1276	DoubleError’	
1277	ChkErrEn’			1277	ChkErrEn’	
1300:1306	tSyn0 to 6	TSYN	54	1300:1306	tSyn0 to 6	
1307	tSyn7x			1307	tSyn7x	
1310:1317	--			1310:1317	--	
		MDMAD	55	3540:3553	0	
				3554:3557	MDMad.0’ to 3’	Tn ge 2
		DADE	56	3560	D0ACE’c	Two chip enables always
				3561	D0BCE’c	predicted false, other two
				3562	D1ACE’c	if (T1Transport & (Tn ge 2))
				3563	D1BCE’c	
				3564	WriteD0’e	Tn ge 2
				3565	WriteD1’d	Tn ge 1
				3566:3577	0	

Table 15A: MemX DMux Signals

* = moved elsewhere (ProcSrn[0:3] and 3 bits for Mcr_ are moved)

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
MAPBUF	60	1400:1417	Mapbuf[0:15]	Tn ge 2
P34INEC	61	1420:1421	Mapbuf.16 to 17	Tn ge 2
		1422	ProcTagInA	Tn ge 2
		1423	PrivRefInPair	Always
		1424:1427	Pipe34Ad.0 to 3	Tn ge 2
		1430	WPinEc1	Tn ge 2
		1431	MapTroubleInEc1	Tn ge 2
		1432	TagInEc2	Never
		1433	CacheRefInEc2	Never
		1434	Store_InEc2’	Never
		1435	IFURefInEc2	Never
		1436	MapPEInEc2	Never
		1437	MapTroubleInEc2	Tn ge 2
MCDTSK	62	1440:1443	MDMtagAd.0 to 3	Always
		1444:1447	CurTask.0 to 3	Always
		1450	ProcTag	Always
		1451	MDMtag’	If CacheRefInPair & (Atask eq CurTask)
		1452	At=Curt’	Always
		1453	Dt=Curt’	Always
		1454:1457	Dtask[0:3]	Never
STA	63	1460	VictimInST	Tn ge 2
		1461	STIdle’	Always
		1462	StartST	Always
		1463	STWait-Mem’	Tn ge 2
		1464:1467	STState[0:3]	Tn ge 2
		1470	STfree’	Tn ge 2
		1471	VictimInA	Always
		1472	MapRfshDly	Tn ge 1
		1473	RefUsesDInEc1	Tn ge 2 & StartEc1
		1474	AWordRefToD	Always
		1475	MapWantsPipe	Tn ge 2
		1476	MapFree	Tn ge 2
		1477	UseAsrn	Tn ge 2
APESRN	64	1500:1503	Asrn.0 to 3	Tn ge 2
		1504:1507	ProcSrn.0 to 3	Tn ge 2
		1510	MapIs16K	Never
		1511	MapIs64K	Never
		1512	MapIs256K	Never
		1513	RfshAd.0	Never
		1514:1517	Ec2Srn[0:3]	Tn ge 2

Table 15B: MemX DMux Signals

* = moved elsewhere (3 bits for Mcr_ are moved)

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
STOUT	65	1520	LoadEn’	Never
		1521	EcLoadEn’	Never
		1522	ShiftEn’	Tn ge 2
		1523	EnEcGen’	Tn ge 2
		1524	MapWait-ST’	Tn ge 2
		1525	STPerrNow’	Never
		1526	EnableAllMods	Never
		1527	StartEc1	Never
		1530	PairFull	Always
		1531	Transporta	Always
		1532	EcFault’	Never
		1533	MemError’	Never
		1534	--	
		1535	ChipsAre256/16K	Never
		1536	ChipsAre64K	Never
		1537	VicSTPerr-.	Never
TAGAT	66	1540	MemColSela	Never
		1541	EcHasA	Tn ge 2
		1542	Ptag	Never
		1543	MapWait-Ec2	Tn ge 2
		1544	Dtag’	Tn ge 2
		1545	sHold	Always
		1546	MapWait-MemState’	Always
		1547	MapRfsh	Always
		1550	AcanHaveD	Tn ge 2
		1551	CacheRefInPair’	Tn ge 2
		1552	EcWordRefToD	Always
		1553	ChkLastPh6	Tn ge 2
		1554:1557	Atask.0 to .3	Tn ge 2
MEMST	67	1560	MapWait-MemD	Never
		1561	MapWait-MemIO	Always
		1562	MemIdle’	Always
		1563	MemFree	Tn ge 2
		1564:1567	MemState.0 to 3	Tn ge 2
		1570	FinNext	Tn ge 2
		1571	MemRfsh	Tn ge 2
		1572	StopFinTaskLoad	Tn ge 2
		1573	DdataGood’	Tn ge 2
		1574	MakeSout_D	Tn ge 2
		1575:1577	MakeTransport[0:2]	Never**
--	70	1600:1607	--	

Table 15C: MemX DMux Signals

* = moved elsewhere (ProcSrn[0:3] and 3 bits for Mcr_ are moved)

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
FLTMEM	71	* 1620	WakeOnCL	
		* 1621	ReportSE’	
		* 1622	NoWakeups	
		1623	ProcSrn_’	Tn ge 1
		1624	Faults	Never
		1625	LoadFltSrn	If independent of FaultSrn eq 0
		1626	ReportFault	Always
		1627	MapPEInMem	Never
		1630	MapTroubleInMem	Never
		1631	RfshInMem	Tn ge 2
		1632	WriteInMem’	Tn ge 2
		1633	MemWP	Never
		1634	IOFetchInMem’	Tn ge 2
		1635	RefUsesD10InMem’	Tn ge 2
		1636	RefUsesDInMem	Tn ge 2
		1637	DirtyIOFetchInMem	Tn ge 2
RFSSRN	72	1640	STPerr	Never
		1641	MapPerr	Never
		1642	HitPerr	Never
		1643	WantRfsh	Tn ge 2
		1644	NeedRfsh	Always
		1645	StartMema	Tn ge 2
		1646	StkWake	Never
		1647	_FaultInfoDly’	Never
		1650:1653	MapSrn.0 to 3	Tn ge 2
		654:1657	MemSrn.0 to 3	Tn ge 2
EC1MAKE	73	1660	StartEc2’	Tn ge 2
		1661	Ec1Free’	Tn ge 2
		1662	Ec1Idle	Always
		1663:1664	Ec1Func.0 to 1	Tn ge 2
		1665:1667	Ec1State.0 to 2	Tn ge 2
		1670	EcWantsAa	Tn ge 2
		1671	FoutNext	Tn ge 2 usually
		1672	MakeFout_D	Tn ge 2 usually
		1673	MakeD_CD	Tn ge 2 usually
		1674	MakeD_Dbuf	Always
		1675	MakeF_D	Always
		1676	MakeMD_D	Always
		1677	MakeMDM_D’	Tn ge 2
MAPCTRL	74	1700:1701	MapbufHi.0 to 1	Never
		1702	MapRAS’	Tn ge 2 when forced high
		1703	MapCAS’	Tn ge 2 when forced high
		1704	MapWE’	Tn ge 2 % StartMap
		1705	RefWE’	Tn ge 2
		1706	DirtyWE’	Tn ge 2 % StartMap
		1707	0	
		1710	MapWait	Always
		1711	WantMapWait’	Always
		1712	ValidMapFltInEc2’	Tn ge 2
		1713:1714	MapFnc.0’ to 1’	Tn ge 2
		1715:1717	MapState.0 to 2	Tn ge 2

Table 15D: MemX DMux Signals

* = moved elsewhere (ProcSrn[0:3] and 3 bits for Mcr_ are moved)

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
PEEC	75	1720:1723	PEsrn.0 to 3	Always
		1724:1727	Ec1Srn.0 to 3	Tn ge 2
		1730	CacheLoad’	Always
		1731	Ec2Free	Tn ge 2
		1732	Ec2Idle	Always
		1733:1734	Ec2Func.0 to 1	Tn ge 2
		1735:1737	Ec2State.0 to 2	Tn ge 2
INMAP	76	1740	RefUsesDInMap’	DirtyIOFetchInMap % ((Tn ge 2) & StartMap)
		1741	RefUsesD10InMap’	Tn ge 2
		1742	DirtyIOFetchInMap’	Never
		1743	WriteInMap’	Tn ge 2
		1744	IOFetchInMap’	Tn ge 2
		1745	_MapInMap	Never
		1746	Store_InMap’	Tn ge 2
		1747	EcWantsPipe4’	Tn ge 2
		1750:1757	--	

Table 16: Disk Controller DMux Signals

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
KSTATE	100	2000	0	
		2001	IndexTW	
		2002	SectorTW	
		2003	SeekTagTW	
		2004	RdFifoTW	
		2005	WrFifoTW	
		2006	ReadData	
		2007	WriteData	
		2010	EnableRun	
		2011	DebugMode	
		2012	RdOnlyBlock’	
		2013	WriteBlock’	
		2014	CheckBlock’	
		2015	Active	
		2016:2017	Select[0:1]	
KSTAT	101	2020	SeekInc	
		2021	HeadOvfl	
		2022	DevCheck	
		2023	NotSelected	
		2024	NotOnLine	
		2025	NotReady	
		2026	SectorOvfl	
		2027	FifoUnderflow	
		2030	FifoOverflow	
		2031	ReadDataErr	
		2032	ReadOnly	
		2033	CylOffset	
		2034	IOBParityErr	
		2035	FifoParityErr	
		2036	WriteError	
		2037	ReadError	
KRAM	102	2040:2043	RamAddr[0:3]	
		2044:2057	Ram[4:15]	
KTAG	103	2060	DriveTag	
		2061	CylinderTag	
		2062	HeadTag	
		2063	ControlTag	
		2064	Tag.000	
		2065	Tag.00	
		2066:2077	Tag[0:9]	
KFIFO	104	2100	ShiftIn	
		2101	ShiftOut	
		2102	ComputeECC	
		2103	NextBlock	
		2104	LoadTag	
		2105	CntDone’	
		2106	OutRegFull	
		2107	InRegFull	
		2110:1113	FifoWaddr[0:3]	
		2114:2117	FifoRaddr[0:3]	

Table 17: Ethernet Controller DMux Signals

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
ERX0	105	2120	PDNew	
		2121	PDOld	
		2122:2125	PDCnt[0:3]	
		2126	PDCntCtrl	
		2127	ReportCollisions	
		2130	RxWakeupsOn	
		2131	EthData.18	
		2132	RxCRCError	
		2133	--	
		2134	RxDataLate	
		2135	RxBusRegFull	
		2136	RxFifoFull	
		2137	RxFifoEmpty	
ETX	106	2140:2142	TxState[0:2]	
		2143	TxEOP	
		2144	TxBusRegFull’	
		2145	TxGone	
		2146	TxSREmpty’	
		2147	TxCntDwn’	
		2150	TxCRCEnbl	
		2151	TxGo	
		2152	TxData	
		2153:2154	TxSRCtrl[0:1]	
		2155	PEOutput	
		2156	TxFifoFull	
		2157	TxFifoEmpty	
ERX1	107	2160:2162	RxState[0:2]	
		2163	RxCollision	
		2164	PDCarrier	
		2165:2166	PDEvent[0:1]	
		2167	RxSRFull’	
		2170	RxEOP	
		2171	RxSync’	
		2172	RxIncTrans	
		2173	RxCRCReset	
		2174	RxCRCClk	
		2175	RxData	
		2176:2177	RxSRCtrl[0:1]	

Table 18A: IFU DMux Signals

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
MEMRQ	120	2400:2407	PcF[8:15]	Tn eq 2 & Testing’
		2410	NewF	Tn eq 2 & Testing’
		2411	KillResponse	Tn ge 1 & Testing’
		2412	Pause	Tn ge 2 & Testing’ unless IFUM write
		2413	RefOutstanding	Never
		2414	IncPcF	Never
		2415	IncPcFG’	Always
		2416	WantIfuRef’	Always
		2417	ThreeOutOfFive	Always
LOADS	121	2420	ValidRam	Always
		2421	J_OddF	Always
		2422	RealPcFG.15	Tn eq 2 & Testing’ unless IFUM write
		2423	FDv	Tn eq 2 & Testing’ unless IFUM write
		2424	GDv	Tn eq 2 & Testing’ unless IFUM write
		2425	HDv	Tn eq 2 & Testing’ unless IFUM write
		2426	JDv	Tn eq 2 & Testing’ unless IFUM write
		2427	MDv’	Tn eq 2 & Testing’ unless IFUM write
		2430	EnableFG’	Always
		2431	XLd	Always
		2432	AlphaXLd	Always
		2433	BrkLd	Tn ne 1
		2434	MLd	Always
		2435	InstrAddrLd	Tn ge 2
		2436	JLda	Always
		2437	GLd’	Always
HJ	122	2440:2447	H[0:7]	Never
		2450:2457	J[0:7]b	Tn ge 2 & Testing’ unless IFUM write when no clock or on J_H the 1’s are checked
MX	123	2460	TwoAlphaX	Tn ge 2 & Testing’
		2461	JFault	Tn ge 2 & Testing’ unless IFUM write
		2462	HFault’	Tn eq 2 & Testing’
		2463	NM=17	Tn ge 2 & Testing’
		2464	TwoAlphaM	Tn ge 2 & Testing’
		2465	TypeJumpM’	Tn ge 2 & Testing’
		2466:2467	LengthM[0:1]	Tn ge 2 & Testing’
		2470:2471	DSel[0:1]	Tn ge 2 & Testing’ unless XShift with DSel eq 0
		2472:2473	LengthX[0:1]	Tn ge 2 & Testing’
		2474:2477	NX[0:3]	Tn ge 2 & Testing’ when unclocked or NM eq 17

Table 18B: IFU DMux Signals

Midas Word Name	Midas Word Number	DMux Address	Signal Name	Simulation Condition
JMPEXC	124	2500	Exception	Always
		2501	SayNotReady	Always
		2502	WantResched	Tn ge 2 & Testing'
		2503	SawRamParityErr	Only during Reset
		2504	SawFGParityErr	Only during Reset or when testing
		2505	ReschedPending	Tn ge 2 & Testing'
		2506	KReady	Always
		2507	--	
		2510	ZapFGH	Always
		2511	ZapJ	Always
		2512	NewJ	Tn ge 2 & Testing' unless IFUM write
		2513	DoJump	Tn ge 2 & Testing' unless IFUM write
		2514	TurnOffAlu	Tn even
		2515	NewGo	Always
		2516	BMuxEnable	Tn even
		2517	FGFault	Never
PCJ	125	2520:2527	PcJ[8:15]	Tn eq 2 & Testing' unless IFUM write
		2530	MLdDly'	Tn ge 2 & Testing'
		2531	BetaInM	Tn ge 2 & Testing'
		2532	FGErrDly	Tn ge 2
		2533	RamErrDly	Tn ge 1
		2534:2535	InstrSet	Never**
		2536	OneByteJumpInJ	Tn ge 2 & Testing'
		2537	OneByteJumpInJd	Tn ge 2 & Testing'
FFK	126	2540	Test_	Tn ge 1
		2541	GenOut_'	Tn ge 1
		2542	NewPC_	Tn ge 2 & Testing'
		2543	IfuReset	Tn ge 2
		2544	BrkIns_	Tn ge 2 & Testing'
		2545	Testing	Tn ge 2
		2546	SignX'	Never
		2547	BrkPending	Tn eq 2 & Testing'
		2550:2552	--	
		2553	TypeJumpK'	Never
		2554	TypePauseK'	Never
		2555:2556	LengthK[0:1]	Never
		2557	SignK	Never

Table 19: Display Controller DMux Signals

Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
APTRS	140	3000	ACurrentWCBFlag	Never
		3001:3007	AReaderPtr.1 to 7	Never
		3010	ANextWCBFlag	Never
		3011:3017	AWriterPtr.1 to 7	Never
BPTRS	141	3020	BCurrentWCBFlag	Never
		3021:3027	BReaderPtr.1 to 7	Never
		3030	BNextWCBFlag	Never
		3031:3037	BWriterPtr.1 to 7	Never
ITEMS	142	3040:3047	AItem.0 to 7	Never
		3050:3057	BItem.0 to 7	Never
SPSIZE	143	3060:3063	AServicePtr.1 to 4	Never
		3064:3067	BServicePtr.1 to 4	Never
		3070	AFifoFull	Never
		3071	BFifoFull	Never
		3072	ASize8	Never
		3073	ASize8-4	Never
		3074	ASize8-4-2	Never
		3075	BSize8	Never
		3076	BSize8-4	Never
		3077	BSize8-4-2	Never
RESON	144	3100	AOn	Never
		3102	BOn	Never
		3103:3104	ARes.0 to 1	Never
		3105:3106	BRes.0 to 1	Never
		3107	OISRcvdData	Never
		3110:3117	--	

Table 20: Other DMux Stuff

* BMUX and ESTAT signals are obtained from the four-bit slice readout. The temperature sensing signals are moved from the position in which the hardware reads them out.

Midas Word Name	Midas Word Number	Midas DMux Address	Signal Name	Simulation Condition
TEMP	160	3500	CBTemp	Never
		3501	BaseTemp	Never
		3502	ProcHTemp	Never
		3503	ProcLTemp	Never
		3504	IFUTemp	Never
		3505	DskEthTemp	Never
		3506:3517	--	
BMUX	163	--	BMux[0:17]	if driven from ALUB
ESTAT	164	4020	PEIMrh	
		4021	PEIMlh	
		4022	MdPE	
		4023	RAMPEen	
		4024	IOBPE	
		4025	RAMPE	
		4026	MemPE	
		4027	MemPEen	
		4030	CIMPErh	
		4031	CIMPElh	
		4032	Stopped	
		4033	MdPEen	
		4034	IMrhPEen	
		4035	IMlhPEen	
		4036	IOBPEen	
		4037	MIRDebugen	

42. Hardware Read/Write Methods

This section discusses the methods Midas uses to read and write each register and memory, so that failing data paths can be identified when Midas reports problems via "Test" or "TestAll". These sequences are included to help maintainers determine what registers or data paths might be malfunctioning when something is found to be non-working.

To understand how the sequences given below communicate information between Midas and the Dorado, you have to understand the lowest-level communication protocols which are discussed in "Dorado Debugging Interface" ([Indigo]<DoradoDocs>DoradoDebugging.press). These primitives are outlined here:

DoradoOut	Storing into DoradoOut sends 13d bits of control information over the printer interface to the (connected) Dorado. This information is interpreted by the receiving hardware as a 3-bit address field, 9-bit data field, and 1-bit strobe.
Strobe	A strobe operation consists of 3 DoradoOut's identical except for the strobe bit, which is first off, then on, then off again. Strobe sequences are used to send commands to the Dorado.
Load Clock	A register internal to the communication interface that can be loaded with one strobe operation.
Load Control	Another register internal to the communication interface that can be loaded with one strobe operation.
Load MIR	MIR can be loaded by four strobe operations, each loading 9 bits of the microinstruction. Midas computes and sends the parity also.
Load CPReg	Two strobe operations load the 16d-bit CPReg, the register from which Midas usually sends data to the Dorado.
Xct(mic)	A microinstruction can be executed by loading it into MIR and single-stepping the Dorado. To get data from Midas to the Dorado, Midas first loads CPReg with data, then executes a microinstruction which does "Q _ Link" or "T _ Link", for example, while the UseCPReg bit in the Clock register is true. This kind of sequence is denoted by "Q _ CPReg(data)" below, which means that the data is routed from CPReg through the multiplexor on the ContA board to the Q or T register. The fact that the B data path is used is not explicitly stated in the microinstruction, but B is the only possible data path; this implicit use of data paths in the examples below is consistent with the conventions of the microprogramming language. Also, a function called "B _ RWCPReg", solely for use by Midas and the baseboard microcomputer is used to do "Link _ B _ CPReg(data)". This function is needed when reading and writing some registers in the control section.
DoradoIn	Reading from DoradoIn obtains 5 bits of data selected according to bits 0..4 in the last DoradoOut operation; it is not necessary to use a strobe operation for the purposes of DoradoIn. The fifth bit is always the current DMux bit; the first 4 bits can be any of the 4 B nibbles, any of 4 error nibbles, either of two MAReg nibbles or the MASync bit (for communication with the baseboard microcomputer).
Read B	Reading 16 bits of B is accomplished by a sequence of four DoradoOut/DoradoIn operations to obtain the 4 nibbles of B data. When a microinstruction is executed for the purpose of extracting data on the B, it is written like "B* _ Q"; "*" denotes that Midas captures the B data.
Read DMux	Midas has special microcode to extract all 2048d DMux signals using strobe operations. This is done in about (32+39)/2 * 2048 * .00018 msec = 13 msec. Assembly code uses about 15 msec more appending B and ESTAT to the readout, rearranging certain signals, and computing histories.
SelectTask	Complicated. See "Dorado Debugging Interface".
SingleStep	Complicated. See "Dorado Debugging Interface".

Run	Complicated. See "Dorado Debugging Interface".
Stop	Complicated. See "Dorado Debugging Interface".
LoadDMD	Execute a "manifold" operation by loading the 11-bit DMux address with a control function and then executing it.
MCXct	MCXct is used to communicate with the baseboard microcomputer. Midas first loads CPReg with a command and then interrupts the microcomputer with DoStrobe(Clock+BaseBAtten); a slowed strobing sequence is used because the microcomputer requires it. Then Midas waits for an acknowledge by doing DoradoIn's until the microcomputer responds with MASync. MCXct can be used while the Dorado is running to extract voltage, current, temperature, and daytime information from the baseboard microcomputer. DoradoIn can be used to get two nibbles of information from the baseboard.

Each sequence below gives the microinstructions or other sequences executed by Midas to read and write each register and memory. These sequences do not include the shifting and masking and other transformations which occur within Midas to position data. Sequences bracketed with "[]" are used to restore registers incidentally smashed on the read or write. The restoration sequences ARE NOT executed when using "Test" or "TestAll"; they ARE executed when registers or memory words appearing in a name value menu are accessed.

D1OUT	Write only. This artificial register allows the user to execute the most primitive control function for the Dorado interactively.
STROBE	Write only by DoStrobe(D), which is equivalent to three D1OUT's with the strobe bit first off, then on, then off again. This artificial register allows the user to send strobed commands interactively.
CPREG	Read by B* _ CPReg; [Restore MIR]. Write by Midas direct handle.
MIR	Read from DMux. Write by Midas direct handle.
IMOUT	Artificial read-only register (part of DMux memory).
Q	Read by B* _ Q; [Restore MIR]. Write by Q _ CPReg(new value); Noop; [Restore MIR; read DMux].
CNT	Read by T _ Cnt; B* _ T; [T _ CPReg(SavedT); Noop; restore MIR]. Write by Q _ CPReg(new value); Cnt _ Q; [Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].
SHC	Read by T _ ShC; B* _ T; [T _ CPReg(SavedT); Noop; restore MIR;]. Write by

Q _ CPReg(new value);
ShC _ Q;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].

MEMBX

Read by
T _ Pointers;
B* _ T;
[T _ CPReg(SavedT); Noop; restore MIR].

Write by
MemBX _ <new value>S;
[Restore MIR; read DMux].

STKP

Read by
T _ TIOA&StkP;
B* _ T;
T _ Pointers;
B* _ T;
[T _ CPReg(SavedT); Noop; restore MIR]. Note that StkOvf and StkUnd are obtained from Pointers while the value in the register is obtained from TIOA&StkP.

Write by
Q _ CPReg(new value);
StkP _ Q;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux]. StkOvf and StkUnd are read-only.

TASK

Read returns value saved at breakpoint.

Write with SelectTask;
[Restore MIR; read DMux].

PROCSRN

Read by B* _ Config;
[Restore MIR].

Write by Q _ CPReg(new value); ProcSRN _ Q;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].

MCR

Read from DMux.

Write by
T _ CPReg(new value);
MCR _ T;
[T _ CPReg(SavedT); Noop; restore MIR; read DMux].
The MCR register is written only when DMuxTab is selected by the "DMux" action. Writing MCR is illegal when OldDMuxTab or DWrong is selected; and DCheck is written instead when DCheck is selected.

CONFIG

Read only by
B* _ Config;
[Restore MIR].

PCX

Read only by
B* _ PCX;
[Restore MIR].

INSSET

Read by
B* _ IFUMLH';
[Restore MIR].

Write by
Q _ CPReg(new value);
InsSetOrEvent _ Q;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].

TESTSYN

Write only by
SelectTask(16B);
T _ CPReg(constant);
MCR _ T;
Q _ CPReg(new value);
Noop;
Store _ T, DBuf _ Q;
LoadTestSyndrome;
[T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); Noop; Q _ CPReg(SavedQ);
SelectTask(SavedTask); restore MIR; read DMux].

UPTIME

Artificial read-only register (part of ABSOL memory).

TGLITCH

Artificial read-only register (part of ABSOL memory).

EVCNTA

Read only by B* _ EventCntA';
[Restore MIR].

EVCNTB

Read by B* _ EventCntB';
[Restore MIR].

Write by
Q _ CPReg(new value);
EventCntB _ Q;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].

AATOVA

Artificial register.

ESTAT

Actually read by Midas direct handle but treated as part of the DMux memory. This is a register rather than simply another word in the DMux memory so that the error enables can be written.

Write error enables with LoadDMD; write MIRDebug with four LoadDMD's.

ABSOL

Read ABSOL by two MCXct's followed by, if the Dorado isn't running, DoStrobe(Clock+UseCPReg).

Write with MCXct(constant+new value) followed by, if the Dorado isn't running, DoStrobe(Clock+UseCPReg);

TPC

Read from a static if address .eq. SavedTask; otherwise,
RdTPC _ CPReg(address);
B* _ Link;
[If address .eq. SavedTask, then done; otherwise, B _ RWCPReg(SavedLink); Noop; restore MIR].

If address eq SavedTask then write static and done;
otherwise, write by
    B _ RWCPReg(new value);
    LdTPC _ CPReg(address);
    Noop;
    [B _ RWCPReg(SavedLink); Noop; restore MIR; read DMux].

TLINK

Read by
SelectTask(address);
B* _ Link;
[SelectTask(SavedTask); Noop; restore MIR; read DMux]

Write by
SelectTask(address);
B _ RWCPReg(SavedLink);
[SelectTask(SavedTask); Noop; restore MIR; read DMux].

OLINK — Artificial memory read and written like TLINK.

IMBD — The setup code common to both the read and the write of IMBD first zeroes BNPC, which is wire-OR'ed with the control store address loaded by manifold operations. This is accomplished by loading TPC for task 17B with 0 and then notifying that task; the notify causes the priority encoder to select task 17B, so BNPC becomes equal to 0.

Read by
SelectTask(0);
RdTPC _ CPReg(17B);
B* _ Link (save old TPC(17B) for restoration later);
B _ RWCPReg(0);
LdTPC _ CPReg(17B);
Noop;
LoadMIR(Notify[17B]); single-step with Freeze off;
B _ CPReg(-1);
three LoadDMD's to setup the control store address;
LoadDMD(constant);
read DMux to get the IMOUT signals;
four LoadDMD's to clear IMBD adress again;
[B _ RWCPReg(SavedTPC17B); LdTPC _ CPReg(17B); Noop; B _ RWCPReg(SavedLink);
SelectTask(SavedTask); DoStrobe(Clock+UseCPReg+ClrReady); DoStrobe(Clock+UseCPReg);
Noop; restore MIR].

Write by
SelectTask(0);
RdTPC _ CPReg(17B);
B _ Link (UseCPReg turned off) to save TPC(17B) for later;
B _ RWCPReg(0);
LdTPC_ CPReg(17B);
Noop;
LoadMIR(Notify[17]); single-step with Freeze off;
B _ CPReg(-1);
three LoadDMD's to setup the control store address;
four LoadDMD's to load left-half of the instruction;
four LoadDMD's to load right-half of the instruction;
four LoadDMD's to clear the Control register;
three more LoadDMD's to something else;
[B _ RWCPReg(SavedTPC17B); LdTPC _ CPReg(17B); Noop; B _ RWCPReg(SavedLink);
SelectTask(SavedTask); DoStrobe(constant); DoStrobe(constant); restore MIR; read DMux;].

IM — Artificial form of IMX memory.

IMX — Read by
B _ RWCPReg(address);
B* _ ReadIM[0];
B _ RWCPReg(address);
B* _ ReadIM[1];
B _ RWCPReg(address);
B* _ ReadIM[2];
B _ RWCPReg(address);
B* _ ReadIM[3];
[B _ RWCPReg(SavedLink); Noop; restore MIR].

Write by
B _ RWCPReg(address);
IMLHR0POK _ CPReg(new value);
B _ RWCPReg(address);
IMRHBPOK _ CPReg(new value);
[B _ RWCPReg(SavedLink); Noop; Noop; restore MIR; read DMux].

ALUFM — Read from static if address is 0 or 16B; otherwise, read by
T _ ALUFM, ALUF[address];
B* _ T;

[T _ CPReg(SavedT); Noop; restore MIR].

Write static if address is 0 or 16B; otherwise, write by
Q _ CPReg(new value);
ALUFMRW _ Q, ALUF[address];
Noop;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].

T

Read by
SelectTask(address);
B* _ T;
[SelectTask(SavedTask); Noop; restore MIR].

Write by
SelectTask(address);
T _ CPReg(new value);
Noop;
[Restore MIR; read DMux].

RBASE

Read by
SelectTask(address);
T _ Pointers;
B* _ T;
[T _ CPReg(SavedT); SelectTask(SavedTask); Noop; restore MIR].

Write by
SelectTask(address);
T _ CPReg(new value);
Noop;
[SelectTask(SavedTask); restore MIR; read DMux].

TIOA

Read by
SelectTask(Address);
T _ TIOA&StkP;
B* _ T;
[T _ CPReg(SavedT); SelectTask(SavedTask); Noop; restore MIR].

Write by
SelectTask(Address);
Q _ CPReg(new value);
TIOA _ Q;
Noop;
[Q _ CPReg(SavedQ); Noop; SelectTask(SavedTask); restore MIR; read DMux].

MEMBASE

Read by
SelectTask(address);
T _ Pointers;
B* _ T;
[T _ CPReg(SavedT); SelectTask(SavedTask); Noop; restore MIR].

Write by
MemBase _ <address>S;
Noop;
[SelectTask(SavedTask); restore MIR; read DMux].

RM

Read by
RBase _ <hiaddress>S;
B _ RB, RStk[lowaddress];
[RBase _ <SavedRBase>S; Noop; restore MIR].

Write by
RBase _ <hiaddress>S;
RB _ CPReg(new value), RStk[lowaddress];
Noop;
[RBase _ <SavedRBase>S; restore MIR; read DMux].

STK and STKX

Read by
Q _ CPReg(address);
StkP _ Q;
SelectTask(0);
B _ RB, RStk[0], Blk[1];
[SelectTask(SavedTask); Q _ CPReg(SavedStkP); StkP _ Q; Q _ CPReg(SavedQ); Noop; restore MIR].

Write by
Q _ CPReg(address);
StkP _ Q;
SelectTask(0);
RB _ CPReg(new value), RStk[0], Blk[1];
Noop;
[Q _ CPReg(SavedStkP); StkP _ Q; Q _ CPReg(SavedQ); Noop; SelectTask(SavedTask); restore MIR; read DMux].

PIPE

Read only by
T _ CPReg(address);
ProcSRN _ T;
B* _ VAhi; *First screen line
B* _ VAlo;
B* _ Pipe2’; *Second screen line
B _ Map’;
B _ Errors’; *Third screen line
B _ Pipe5;
[T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedT); Noop; restore MIR].

BR

Read by
MemBase _ <address>S;
T _ CPReg(constant);
MCR _ T;
T _ CPReg(0);
DummyRef _ T;
Noop;
B* _ VAhi;
B* _ VAlo;
[MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(SavedTask); Noop; restore MIR].

Write by
MemBase _ <address>S;
T _ CPReg(constant);
MCR _ T;
T _ CPReg(highdata);
BRhi _ T;
T _ CPReg(lowdata);
BRlo _ T;
Noop;
[MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(SavedTask); restore MIR; read DMux].

For both the read and the write, the cleanup is done by a subroutine shared with other memories. "T _ CPReg(SavedSRN); ProcSRN _ T;" and the "SelectTask(SavedTask)" are extraneous to the requirements of BR.

BRX

Same as BR with "MemBaseX _ <address>S" replacing "MemBase _ <address>S".

CACHEA

Read by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;

MemBase _ 36S;
(Other stuff to save BR 36 if not saved yet since breakpoint--this stuff isn't done automatically at breakpoints because it prevents continuing.);
T _ CPReg(constant+lowaddress);
MCR _ T;
T _ CPReg(hiaddress);
Noop;
TurnOffRefresh with two LoadDMD's;
DummyRef _ T;
Noop;
B* _ Pipe5;
restore Refresh;
B* _ VAhi;
B* _ VAlo;
[T _ CPReg(DisHold+DisCF+NoWake); MCR _ T; T _ CPReg(SaveBR36!0); BRhi _ T; T _ CPReg(SaveBR36!1); Xct(BRLOFT); MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(OldTask); restore MIR; read DMux].

Write by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;
MemBase _ 36S;
(Other stuff to save BR 36 if not saved yet since breakpoint);
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hi(address - new flag value));
BRhi _ T;
T _ CPReg(low(address - new flag value));
BRlo _ T;
T _ CPReg(new flag value);
Noop;
Fetch _ T;
Noop;
Fetch _ T;
Noop;
T _ CPReg(constant+lowaddress);
MCR _ T;
T _ CPReg(new flag value);
Noop;
TurnOffRefresh;
DummyRef _ T;
CFlags _ T;
restore Refresh;
Noop;
[T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; T _ CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMemBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; (T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(SavedTask); restore MIR; read DMux].

CACHED

Read by
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hiaddress);
BRhi _ T;
T _ CPReg(lowaddress);
BRlo _ T;
T _ CPReg(0);
Noop;
Fetch _ T;
T _ Md;
B* _ T;
[T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; T _

CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(SavedTask); Noop; restore MIR].

Write by
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hiaddress);
BRhi _ T;
T _ CPReg(lowaddress);
BRlo _ T;
T _ CPReg(0);
Noop;
Q _ CPReg(new value);
Store _ T,
DBuf _ Q;
Noop;
[Q _ CPReg(SavedQ); Noop; T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; T _ CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(SavedTask); restore MIR; read DMux].

MAP

Read by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;
MemBase _ 36S;
(Other stuff to save BR 36 if not saved yet since breakpoint);
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hiaddress);
BRhi _ T;
T _ CPReg(lowaddress);
BRlo _ T;
T _ CPReg(0);
Noop;
RBase _ 0S;
RB _ CPReg(0), RStk[0];
RMap _ RB, RStk[0];
B* _ Map';
B* _ Errors';
B* _ Config';
[RB _ CPReg(SavedR0), RStk[0]; RBase _ <SavedRBase>S; T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; XctL16T(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT); SelectTask(SavedTask); Noop; restore MIR].

Write by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;
MemBase _ 36S;
(Other stuff to save BR 36 if not saved yet since breakpoint);
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hiaddress);
BRhi _ T;
T _ CPReg(lowaddress);
BRlo _ T;
T _ CPReg(0);
Noop;
Q _ CPReg(hi new value);
TIOA _ Q;
Q _ CPReg(low new value);
Map _ T, MapBuf _ Q;

```
Noop;
Q _ CPReg(SavedTIOA);
TIOA _ Q;
Q _ CPReg(SavedQ);
Noop;
[T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; T _
CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN);
ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT);
SelectTask(SavedTask); restore MIR; read DMux].
```

VM

```
Read by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;
MemBase _ 36S;
(Other stuff to save BR 36 if not saved yet since breakpoint);
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hiaddress);
BRhi _ T;
T _ CPReg(lowaddress);
BRlo _ T;
T _ CPReg(0);
Noop;
Fetch _ T;
T _ Md;
B* _ T;
[T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; T _
CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN);
ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT);
SelectTask(SavedTask); Noop; restore MIR].

Write by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;
MemBase _ 36S;
(Other stuff to save BR 36 if not saved yet since breakpoint);
T _ CPReg(constant);
MCR _ T;
T _ CPReg(hiaddress);
BRhi _ T;
T _ CPReg(lowaddress);
BRlo _ T;
T _ CPReg(0);
Noop;
Q _ CPReg(new value);
Store _ T, DBuf _ Q;
Noop;
[Q _ CPReg(SavedQ); Noop; T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0);
BRhi _ T; T _ CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _
CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT);
SelectTask(SavedTask); restore MIR; read DMux].
```

IFUM

```
Read by
IFUReset;
Q _ CPReg(F(address));
InsSetOrEvent _ Q;
Q _ CPReg(F(address));
BrkIns _ Q;
Noop;
B* _ IFUMRH';
B* _ IFUMLH';
[Q _ CPReg(SavedQ); Noop; restore MIR].
```

Write by
IFUReset;
Q _ CPReg(hiaddress);
InsSetOrEvent _ Q;
Q _ CPReg(lowaddress);
BrkIns _ Q;
Noop;
Q _ CPReg(new value);
IFUMLH _ Q;
B _ Q;
Q _ CPReg(new value);
IFUMRH _ Q;
B _ Q;
Noop;
[Q _ CPReg(SavedQ); Noop; restore MIR; read DMux].

LDR — Artificial memory.

MDATA — Artificial memory.

MADDR — Artificial memory.

DMUX — Read only by the special Alto microcode and software discussed at the beginning of this section, when "current" DMux readout is selected. Old, wrong, or checked tables may be selected for this memory by the "DMux" command action; when DChecked is selected, writes are legal.

DHIST — Artificial memory.

VH — Artificial memory.

MD — Read only by
SelectTask(address);
T _ Md;
B* _ T;
[T _ CPReg(SavedT); SelectTask(SavedTask)].

TASKN — Artificial memory which cannot be read or written (used for displaying some values symbolically).

DEVICE — Artificial memory which cannot be read or written (used for displaying some values symbolically).

MSTAT — Artificial form of ABSOL memory.

ABS — Artificial form of ABSOL memory.

ROW — Read lines 0 to 3 like CACHEA; line 4 (Victim/Next Victim) by
SelectTask(17B);
T _ CPReg(1);
ProcSRN _ T;
MemBase _ 36S;
(other stuff to save BR 36 if not saved yet since breakpoint);
T _ CPReg(constant);
MCR _ T;
T _ CPReg(F(address));
DummyRef _ T;
Noop;
B* _ Pipe5;
[T _ CPReg(constant); MCR _ T; T _ CPReg(SavedBR36!0); BRhi _ T; T _ CPReg(SavedBR36!1); BRlo _ T; MemBase _ <SavedMBase>S; T _ CPReg(SavedSRN); ProcSRN _ T; T _ CPReg(SavedMCR); MCR _ T; T _ CPReg(SavedT);

SelectTask(SavedTask); Noop; restore MIR]

Write rows 0 to 3 like CACHEA; row 4 is read-only.

```
 CONFIG          0   PROBLEMS           0   UPTIME               0 days 3:24:42
 CLKRUN       1040   OUTOFSPEC          0   TGLITCH               0 days 1:2:31
 ESTAT           0   BADSUPPLYSPEC      0   COMM-ERRS                        0
 INSSET          2                          MIR-PES                          0
 OLINK 20      344                          VOLTS      +12.07 +4.93 -1.98 -5.36
 TLINK 20      345                          AMPS            6    27    75   150
*TPC 20        346                          TEMP0         +27   +35    ??   +27
 RBASE 20       17                          TEMP0+1       +27   +33    ??    ??
 MEMBASE 20     14                          TEMP0+2       +25   +23    --    --
 T 20       177767  *RTEMP         133747   MINVOLTS   +12.07 +4.93 -1.97 -5.18
 TIOA 20         0  *LTEMP         122001   MAXVOLTS   +12.07 +4.93 -1.98 -5.36
 CNT             1                          MINAMPS         5    26    73    86
 STKP            1                          MAXAMPS        34    55   109   154
 MEMBX           3  BMUX           177777   MAXTEMP0      +27   +35    ??   +27
*Q          177766                          MAXTEMP0+1    +27   +33    ??    ??
*SHC            0                          MAXTEMP0+2    +25   +23    --    --
 PCX             0
 PROCSRN         0  PIPE 0             0   DWATCH                           0
 MCR             0                     0   IMOUT                   321747 023457
 TASK            0                     0   MIR                     124576 035777


Loaded: KERNEL

Go at 0:BEGIN, BrkP after 0:QERR+1 at 0:QERR+2


RunProg RdCmds Brk UnBrk Go SS OS Passive Ld LdSyms Cmpr Dtach Reset SetClk
Config PEscan TestAll Test SimTest SimGo T1 T2 T3 RepGo RepSS RepT2 Fields
LDRtest ShowCmds WrtCmds Virtual DMux

BEGIN;
```

Sample Midas Display