# The Dorado Kernel Diagnostics

by Gene McDaniel

December 26, 1978 1:55 PM

Kernel is an ordered set of microcoded diagnostics for the Dorado processor, ifu, and memory system. This document has two parts. The first describes the working context and assumptions of these diagnostics, and the second provides a guide to the specific tests. The guides characterize each test by its title, the function it tests and the strategy employed to test the function. There are comments on the possible implications of an error indication, and a description of register usage.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California

# The Dorado Kernel Diagnostics

Gene McDaniel

October 3, 1978 10:23 PM

## INTRODUCTION

There are several layers to the microcoded Dorado Diagnostics, each layer tests a different portion of the machine, and all the diagnostics presume the existence of an external machine that runs the Dorado version of the Midas hardware debugger. The diagnostic file named "Kernel" tests the two processor boards and the diagnostic file named "Control" tests the two control boards. The Kernel and Control programs both presume that the two control boards work reasonably well. The phrase "Kernel Diagnostics" denotes both the control and the processor diagnostics; they exercise all the functionality of the Dorado processor that is not associated with the IFU, Memory, or with IO devices.

This document describes what Kernel does and how it goes about doing it. Each series of tests is documented here. The documentation describes the name of the test, the function tested, the strategy applied to test the function, and some implications of an error failure. **This document is not a substitute for actual microcode; it is a supplement that will help the diagnostics user to understand and manipulate the diagnostics. A microcode listing must be at hand when using these diagnostics.** The rest of this introduction concerns itself with the general context and assumptions of the Kernel diagnostics. The subsequent sections contain the documentation for each functional test.

### What Kernel Does

Kernel makes two assumptions at the beginning of its execution: the control boards work most of the time and that Midas has loaded RM with predetermined values. Given this assumption known values are moved from RM to T and back. **The user is expected to single step (and check) this code the first time through.** This is the only time the user is expected to single step the diagnostics (though the user may want to singlestep a piece of code that is known to fail so that intermediate steps can be observed by using Midas). Kernel can not and does not assume that the A and B multiplexors, the Asel and Bsel logic, or the Load Control functions of the processor work. The user will single step this code from Midas in order to watch the known values move from RM into T and then back into RM. This exercise is also performed with selected B mux constants. After the user single steps the initial portion of the diagnostic, Kernel can perform the rest of its diagnostics function without operator intervention.

The kernel diagnostics usually operate with RBASE = 17B, the "defaultRegion". Initially Midas loads RBASE and the diagnostics carefully reload RBASE with that value if they change it for any reason.

### How Kernel Tests the Processor

Kernel begins by assuming that there are known values in RM that can be used to test the alu=0 fast branch function of the processor. Once it is certain that a test for =0 can be made, the known values in RM can be manipulated in predictable ways to test more functions. After a function or register has been tested, the diagnostics may presume to use that function or register in the process of testing a different part of the machine. Thus Kernel gradually increases the portion of the machine it has checked out and the portion of the machine that it uses to do the rest of the checking. The known values in RM are shown below.

| RM name | Value |
|---|---|
| R0 | 0 |

| | |
|---|---|
| R1 | 1 |
| RM1 | -1 |
| RHIGH1 | 100000 |
| R01 | 52525 (alternating 01) |
| R10 | 125252 (alternating 10) |

The diagnostics test a function by simulating it: the known RM values, selected B mux constants and the already tested functions of the processor are used to **predict** the result of some function. Then the actual function is exercised. There are two scratch registers known as RSCR and RSCR2 that hold temporary values and predicted results. Usually the xor of the predicted and actual value is placed in T. In this case where T is non-zero, there is an error and the one bits in T indicate the wrong bit values. ALL errors cause a subroutine call to the ERR code where a breakpoint has been assembled. If a breakpoint occurs at ERR, OLINK contains the address of the call to the error code. Usually there is a label close the point of the call, and that label can be used to determine which test has failed.

## Naming Conventions

Special naming conventions are followed to make the Kernel code readable, maintainable and usable. Labels carry information about what test they are attached to. Long tests are sprinkled with labels that begin with the test name end with a number or a letter to indicate where in the test the label is located. Loops are suffixed with the letter L and loop exits are suffixed with XITL In the microcode listing, upper and lower case is used to make the concatenations that construct label names more obvious. Unfortunately, Midas prints upper case labels only. Some examples follow:

| | |
|---|---|
| aluEQ0FF | test ALU=0 fast branch using Bmux constants (from FF) |
| aluEQ0RT | as above, except data from RM and T |
| NotA | test the alu function NOT A |
| QRWL | top of the read/write loop that tests Q |
| Rlsh5 | test left shift of RM by 5 bits |
| RTlcy2 | test left cycle of RM,,T by two bits |
| RFLMASK | test left mask after RF← |
| cntFcnIL | inner loop label for CNT=0&+1 test |

In addition to the register and label names there are name conventions for bit constants. For each bit in a 16 bit word there is an assembler constant for the bit and for the complement of the bit:

| Name | Value |
|---|---|
| B0 | 100000B |
| B1 | 40000B |
| B2 | 20000B |
| ... | |
| B15 | 1B |
| | |
| NB0 | 77777B |
| NB1 | 137777B |
| NB2 | 167777B |
| ... | |
| NB15 | 177776B |

There are several other important conventions associated with constants: names with the suffix "C" are FF B-mux constants, names with a hardware register name and period as a prefix refer to constants associated with that register, and names with the suffix "shift" may be used in shifter instructions to right justify the specified field.

maxCountC a bmux ff constant
mcr.noWake a bmux constant that selects the noWake bit in Mcr
pipe2.nFaultsShift a value that may be used in a shifter instruction (eg., "t ← shiftLmask, rsh t[pipe2.nFaultsShift];")

When Kernel indicates an error condition, use Midas to determine the source of the call. The information displayed by Midas will usually be a "virtual" IM location (a label can be obtained by applying the middle mouse button to the value in OLINK). The .DLS listing (produced by MicroD, it is a listing that shows where the Dorado instructions were placed) and the source listing can be used to determine which portion of the diagnostics failed.

## Macro Conventions

To make the code easier to read, a number of macros have been created. With a few exceptions they are fast branch macros. A typical example is, "SKPIF[ALU=0]" which takes the place of "DBLBRANCH[.+2, .+1, ALU=0]". An important exception is the ERROR macro. This macro causes a call to the ERR (global) location of the Kernel code; consequently OLINK points to the calling instruction.

**Common Kernel Macros**

| Macro Name | Effect |
|---|---|
| NOOP | BRANCH[.+1] |
| SKIP | BRANCH[.+2] |
| SKPIF[COND] | DBLBRANCH[.+2,.+1, COND] |
| SKPUNLESS[COND] | DBLBRANCH[.+1,.+2, COND] |
| ERROR | BRANCH[ERR] |

note: "COND" is any fast branch conditional, like R EVEN, ALU >=0, etc.

## Operating Procedures for the Diagnostics

### Files

The names of the two files necessary and sufficient to invoke the kernel diagnostics from Midas are [maxc]<dl>kernel.mb and [maxc]<dl>kernel.midas. The alto executive command, "midas kernel" will cause midas to initialize the display and to load the kernel processor diagnostics. The Midas command "BEGIN;G" causes the diagnostics to run. The diagnostics run until an error occurs or until they are aborted by a command from Midas.

The file [maxc]<dlsources>kernelSources.dm contains the source files necessary to reconstruct the kernel diagnostics. Included in that file are command files to generate a new kernel from the sources and a command file to update the relevant Maxc directories. Kernel diagnostics are currently implemented in five microcode source files and one "preamble" definitions file. The sources are named kernel.mc, kernel1.mc, . . ., kernel4.mc. The "kernel" file is a driver file that causes kernel1 through kernel4 to be assembled.

The Dorado debugging disk contains sufficient files to execute the procedures described below. The file [maxc]<dl>dlDebugDisk.cm can be fetched onto an Alto disk -- a mostly empty disk -- and run to construct a debugging disk; it will have sufficient files to write new diagnostics, modify current diagnostics and run Midas.

## Simple Procedures

In the following procedures **cr** denotes a carriage return typed to the Alto executive, the indented command denotes characters typed to Midas, and the text that follows is a comment.

**Quick checkout**
midas checkout **cr** This is typed to the Alto executive and causes Midas to execute a series of commands that will checkout the Dorado. When it is done, basic features of the machine have been tested.

**Extended kernel checkout**
midas kernel **cr**

BEGIN;G This starts up the kernel diagnostics. They run in an inifinite loop, so any termination of the diagnostic other than operator termination constitutes a bug.

**Extended control checkout**
midas controll **cr**

BEGIN;G This starts up the control diagnostic. It runs in an infinite loop, so any termination of the diagnostic other than operator termination constitutes a bug.

## Changing the Diagnostic Environment

There is a Dorado facility that simulates hold and a facility that simulates task switching. The diagnostics are prepared to utilize these features to check out processor performance in the presence of hold and/or task switching. Furthermore, the software will, upon command, cause the diagnostics to run at different task levels. As described in the Midas documentation, the user may invoke microcode subroutines from Midas by typing "subroutineName()". One parameter may be passed automatically in t by placing its value inside the parentheses. Valuable facilities available "at the Midas level" are described below:

### Task Simulation

The subroutine "**xorTaskSim()**" toggles the value of flags.taskSim. The postamble checks this bit at the end of each iteration, and, if it is set, causes the task simulator to run (task 17). The code that implements this facility chooses a different value for the task simulator at the end of each iteration.

### Hold Simulation

The subroutine, "**xorHoldSim()**" toggles the value of flags.holdSim. As with task timulation, postamble checks the value of this bit at the end of each iteration and chooses a new value for the hold simulator if hold simulation is enabled.

### Task Circulation

The subroutine, "**xorTaskCirc()**" toggles the value of flags.testTasks. Postamble treats it similarily to hold and task simulation. If task simulation is enabled, task circulation occurs only within tasks 0 - 16. If task simulation is disabled, task 17 gets included in task circulation.

## Sample Problems

The following list of edge pins should be shorted to ground. Each one of these shorts will introduce a different bug in the processor, and that bug will be detected by the kernel diagnostics. Beginning users can take this opportunity to use the diagnostics/midas ensemble to find "known" bugs and get a feeling for how some of the regular hardware debugging will proceed. Realize that shorted edge pins are not the most common form of error; this is strictly an introductory exercise. The pieces of the machine that are affected are described in the appendix to this document.

pin 35 on left side
pin 166 on left side
pin 134 on left side
pin 146 on left side
pin 162 on left side

The rest of this document contains an overview of each test. These overviews describe the "name" of the test (the entry point label) and provide other useful information about the test. The term "exhaustive test" is used to indicate that all possible values are used to check a register or function. This implies a microcode loop of some sort. The term "loop variable" usually refers to the RM location that holds the changing test value. For example, an exhaustive test of Q reading and writing Q would require loading and checking it for all possible 16 bit values. This is exactly what the test that checks Q does.

**Remember, this documentation is not a substitute for a microcode listing.**

October 26, 1977 1:03 PM

**Title:** Begin
**Functions Tested:** load control, asel, bsel, "A" and "B" straight through the alu
**Strategy:** Operator checking with single steping. At this point Kernel assumes nothing other than certain values are located in RM. The operator must verify that it is possible to move data from RM to T and back before Kernel can test fast branches.
**Comments:** There are no automatic error indications.

**Title:** aluEQ0FF
**Functions Tested:** ALU=0 fast branch, data from B mux constants
**Strategy:** All 16 possible single one bit values are passed through the alu and tested for zero. E.g., the values 1, 2, 4, 10, etc are tested for equals zero. The bypass logic is avoided.
**Comments:** An error in this test indicates that one of the bit positions is invisible to the =0 logic of the processor. T contains the value on which the "=0" test was performed. Examine the code preceeding the ERROR call to determine which value failed.

**Title:** alu=0RT
**Functions Tested:** ALU=0 fast branch, data from RM and T
**Strategy:** Known values in RM are moved into T and a check for zero is made. Then that value is moved back into RM and a check for zero is made. The bypass logic is avoided.
**Comments:** As with aluEQ0FF, an error indicates that the =0 fast branch logic is not working properly. T contains the value on which the "=0" test was performed.

**Title:** aluLT0RT
**Functions Tested:** ALU < 0 fast branch, data from RM and T
**Strategy:** Known values in RM are moved into T and a check for less than zero is made. Then that value is moved back into RM and a check for less than zero is made. The bypass logic is avoided.
**Comments:** An error indicates that the <0 fast branch logic is not working properly. T contains the value on which the "<0" test was performed.

**Title:** rEven
**Functions Tested:** RM EVEN fast branch, data from RM and T
**Strategy:** Kernel moves known values from RM into T and checks to see if the values were even. The bypass logic is avoided.
**Comments:** An error indicates that the "R EVEN" fast branch logic is not working properly. T contains the value on which the "R EVEN" test was performed.

**Title:** rGE0
**Functions Tested:** RM >=0 fast branch, data from RM and T
**Strategy:** Kernel moves known values from RM into T and checks to see if the values were greater than or equal to zero. The bypass logic is avoided.

**Comments:** An error indicates that the "R >=0" fast branch logic is not working properly. T contains the value on which the "R >=0" test was performed.

**Title:** xorNoBypass
**Functions Tested:** xor (#)
**Strategy:** Kernel successively tests each bit by xoring the bit with itself. RSCR and T are loaded with the current bit and then T is loaded with T xor RSCR. The result should be zero. Noops are used to avoid the bypass logic.
**Comments:** An error indicates that the =0 fast branch logic took a false path (result non zero) after xoring a bit with itself. Examine the code preceeding the error call to determine which bit was used. The non-zero bits in T are in error.

**Title:** xorBypass
**Functions Tested:** xor (#), bypass logic
**Strategy:** Kernel successively tests each bit by xoring the bit with itself. RSCR and T are loaded with the current bit and then T is loaded with T xor RSCR in a fashion that uses the bypass logic. The result should be zero.
**Comments:** An error indicates that the =0 fast branch logic took a false path (result non zero) after xoring a bit with itself. Examine the code preceeding the error call to determine which bit was used. The non-zero bits in T are in error.

**Title:** Aplus1
**Functions Tested:** A + 1
**Strategy:** Selected, known values in RM, and selected B mux constants are incremented by one, and the result is checked for accuracy.
**Comments:** The non-zero bits in T are in error.

**Title:** AplusB
**Functions Tested:** A + B
**Strategy:** Selected, known values in RM, and selected B mux constants are added together, and the result is checked for accuracy.
**Comments:** The non-zero bits in T are in error.

**Title:** Aminus1
**Functions Tested:** A -1
**Strategy:** This is an exhaustive test of A-1 that checks all possible 16 bit values: there is a loop that subtracts one from the current loop variable and then increments that value. The new value should be equal to the original loop variable. Note that this test presumes A+1 works.
**Comments:** The non-zero bits in T are in error, and the original value is in RSCR. The bits in T represent (original value) xor ( (original value -1) +1).

**Title:** aMinusB
**Functions Tested:** A - B
**Strategy:** Selected values in RM and selected B mux constants are subtracted from each other and the result is compared with the predicted result.
**Comments:** The non-zero bits in T are in error.

**Title:** carryNo
**Functions Tested:** ALU CARRY fast branch
**Strategy:** Selected values in RM and selected B mux constants are used as operands in arithmetic operations. None of the operations should generate a carry.
**Comments:** An error indicates a false carry condition. Note that T contains the result of an arithmetic operation rather than the xor of a predicted and actual result.

**Title:** carryYes
**Functions Tested:** ALU CARRY fast branch
**Strategy:** Selected values in RM and selected B mux constants are used as operands in arithmetic operations. All of the operations should generate a carry.
**Comments:** An error indicates a missing carry condition. Note that T contains the result of an arithmetic operation rather than the xor of a predicted and actual result.

**Title:** carryOps
**Functions Tested:** ALU CARRY fast branch
**Strategy:** Selected values in RM and selected B mux constants are used as operands in arithmetic operations. Successive operations and skips are performed. This is a more complicated version of the carryNo and carryYes tests. Carry may or may not be generated in a particular test. This test interleaves carry generation and testing so that the instruction that has a fast branch clause that tests a preeceding operation also has an arithmetic operation that will be tested with a fast branch clause in the following instruction.
**Comments:** An error indicates an incorrect carry condition. The code must be examined for the particulars; especially note that T contains the result of the next arithmetic operation that would have been checked.

**Title:** freezeBCtest
**Functions Tested:** FREEZEBC function
**Strategy:** Selected fast branch conditions are generated and the FREEZEBC function is used to freeze all the conditions across numerous instructions where each condition is tested. Tests are made to check that FREEZEBC happens soon enough and that it releases appropriately.
**Comments:** An error indicates an incorrect fast branch condition. The code must be examined for the particulars.

October 26, 1977 1:20 PM

**Title:** cntRW
**Functions Tested:** CNT←, ←CNT, data source from B mux
**Strategy:** Test loading CNT from B mux and reading CNT onto the B mux in a loop that generates all possible 8 bit values. Cnt is loaded with the loop variable and then read again. The value read is added to 177400B to construct a "real" negative number (cnt is loaded with a full 16 bit negative number; of course, only the low 8 bits will be used). That value, the constructed negative value, is compared against the loop variable.
**Comments:** An error indicates that the value read from CNT was not the same as the value loaded. The one bits in T are in error, and RSCR contains the value used to load CNT.

**Title:** cntFFrw
**Functions Tested:** CNT←, ←CNT, data source from FF
**Strategy:** Load CNT with selected small constants (FF). The value is read back (177400B is added to the value read back to make it a 16 bit negative number) and checked for correctness.
**Comments:** An error indicates that the value read from CNT was not the same as the value loaded. The one bits in T are in error.

**Title:** cntFcn
**Functions Tested:** CNT←, CNT=0&+1 fast branch
**Strategy:** This is an exhaustive test of the CNT fast branch facility. There is an outer loop (cntFcnOL) that loads CNT with all possible 8 bit values. The inner loop (cntFcnIL) decrements both CNT (by using the fast branch) and a parallel counter for error checking. The inner loop exit happens when CNT=0.
**Comments:** An error in the inner loop indicates that the parallel counter is >= 0. This means that the fast branch exit that should have happened did not occur. An error in the outer loop indicates that the fast branch exit did occur and it should not have occured. RSCR holds the parallel counter and RSCR2 holds the loop variable.

**Title:** NotAtest
**Functions Tested:** NOT A
**Strategy:** This test checks the NOT A alu function of the processor. Selected values are put on the A mux and complemented. The predicted value is xor'd with the actual value and placed in T.
**Comments:** The one bits in T are in error.

**Title:** NotBtest
**Functions Tested:** NOT B
**Strategy:** This test checks the NOT B alu function of the processor. Selected values are put on the B mux and complemented. The predicted value is xor'd with the actual value and placed in T.
**Comments:** The one bits in T are in error.

**Title:** AandBtest
**Functions Tested:** A AND B
**Strategy:** This test checks the A AND B alu function of the processor. Selected values are ANDed together. The predicted value is xor'd with the actual value and placed in T. This test assumes that the sources for the A and B muxes don't matter. E.g., T ← (B←T) AND (A←RM) is identical to T ← (A←T) AND (B←RM).
**Comments:** The one bits in T are in error.

**Title:** AorBtest
**Functions Tested:** A OR B
**Strategy:** This test checks the A OR B alu function of the processor. Selected values are ORed together. The predicted value is xor'd with the actual value and placed in T. This test assumes that the sources for the A and B muxes don't matter. E.g., T ← (B←T) OR (A←RM) is identical to T ← (A←T) OR (B←RM).
**Comments:** The one bits in T are in error.

**Title:** LINKRW
**Functions Tested:** LINK←, NOTLINK
**Strategy:** This is an exhaustive test of loading and reading LINK. The loop label is LINKL, and the value stored into LINK is in RSCR.
**Comments:** The one bits in T are in error.

**Title:** callTest
**Functions Tested:** CALL
**Strategy:** This is a minimal check of the microcode subroutine facility. Registers are set up and a call is made to a local subroutine. The register values are checked, and some registers are changed. Immediately after return from the local subroutine a call is made on a global subroutine. The same sort of register checking occurs. Note that there is no direct way of checking whether the subroutine calls cause wild control transfers.
**Comments:** The one bits in T are in error.

**Title:** QtestRW
**Functions Tested:** Q←, ←Q
**Strategy:** This is an exhaustive test of reading and writing Q. The microcode loop QRWL loads and checks Q for all possible 16 bit values: the predicted value, which is kept in RSCR, is xor'd with the actual value, and the xor is placed in T.
**Comments:** The one bits in T are in error.

**Title:** STKPtestRW
**Functions Tested:** STKP←, TIOA&STKP
**Strategy:** This is an exhaustive test of reading and writing STKP. The microcode loop STKPL loads and checks STKP for all possible 6 bit values: the predicted value, which is kept in RSCR, is xor'd with the actual value and placed in T.
**Comments:** The one bits in T are in error.

**Title:** STKBOUNDRW
**Functions Tested:** STKBOUND←, TIOA&STKP
**Strategy:** This is an exhaustive test of reading and writing STKBOUND. The microcode loop STKBOUNDL loads and checks STKBOUND for all possible 4 bit values: the predicted value is xor'd with the actual value and placed in T.
**Comments:** The one bits in T are in error. RSCR contains the predicted value, and RSCR2 contains a 4 bit mask that is used to isolate stkbound from the values returned by TIOA&STKP.

**Title:** RSTKTEST0
**Functions Tested:** rstk destination function
**Strategy:** This is an exhaustive test of the FF that enables the processor to read one RM location and write another. The code is "expanded in line", ie, there are no loops. Each label denotes the destination address that is being changed in the FF field of the test. Consider RSTKTEST2: This section tests loading the RM address RBase,,2 from the *other* RM addresses (RBase,,0, RBase,,1, RBase,,3, ..., RBase,,7). First a check is made to see if the new value of the destination RM is different from its original value. Then a check is made to see if the current value of the destination RM is the same as the value with which it was loaded
**Comments:** **Caution: the one bits of T are not always in error.** The first test for each rstk location performs T # Q only; there is NO assignment into T. The second test assigns into T.

Kernel 3: SHCtestRW Rlsh Tlsh Rrsh Trsh TRlcyTest RTlcyTest RFWFtest aluRSH aluRCY 1 aluARSH aluLSH aluLCY aluSHTEST

October 19, 1977 1:54 PM

**Title:** SHCtestRW
**Functions Tested:** SHC←, ←SHC
**Strategy:** This is an exhaustive test for reading and writing SHC by generating all possible 16 bit values.
**Comments:** An error indicates that the value read from SHC was not the same as the value loaded. The one bits in T are in error, and RSCR contains the value used to load SHC.

**Title:** Rlsh
**Functions Tested:** SHIFTRMASK, LSH R[i] where 0<=i<=15
**Strategy:** This is an exhaustive test for left shifting RM values. The value 1 is left shifted for all values in [0..15] and the result is checked for accuracy.
**Comments:** An error indicates that some left shift failed. Check the code to see which shift failed. The one bits in T are in error.

**Title:** Tlsh
**Functions Tested:** SHIFTRMASK, LSH T[i] where 0<=i<=15
**Strategy:** This is an exhaustive test for left shifting T values. The value 1 is left shifted for all values in [0..15] and the result is checked for accuracy.
**Comments:** An error indicates that some left shift failed. Check the code to see which shift failed. The one bits in T are in error.

**Title:** Rrsh
**Functions Tested:** SHIFTLMASK, RSH R[i] where 0<=i<=15
**Strategy:** This is an exhaustive test for right shifting RM values. The value 100000 (which is kept in Q) is right shifted for all values in [0..15] and the result is checked for accuracy.
**Comments:** An error indicates that some right shift failed. Check the code to see which shift failed. The one bits in T are in error.

**Title:** Trsh
**Functions Tested:** SHIFTLMASK, RSH T[i] where 0<=i<=15
**Strategy:** This is an exhaustive test for right shifting T values. The value 100000 (which is kept in RHIGH1) is right shifted for all values in [0..15] and the result is checked for accuracy.
**Comments:** An error indicates that some right shift failed. Check the code to see which shift failed. The one bits in T are in error.

**Title:** TRlcyTest
**Functions Tested:** T R LCY[i] where 0<=i<=15
**Strategy:** This is an exhaustive test for left cycling the quantity in T,,RM. There are two phases in this test for each left cycle count; the first phase left cycles the quantity 0,,1 (all zeros except for one "1") and the second phase left cycles the quantity 177777,,177776 (all ones except for one "0"). Note that the RM location R01 is renamed RM2 and it is loaded with the value 177776 (-2). This register and its value is used throughout the test. Note also that NBi where i IN[0..15] stands for logical complement of Bi. E.g., NB0 ==> NOT(B0) ==> NOT(100000) ==> 77777.

**Comments:** An error indicates that some left cycle failed. Check the code to see which cycle failed. RSCR2 holds the predicted result, and the one bits in T are in error.

**Title:** RTlcyTest
**Functions Tested:** R T LCY[i] where 0<=i<=15
**Strategy:** This is an exhaustive test for right cycling the quantity in RM,,T. There are two phases in this test for each right cycle count; the first phase right cycles the quantity 0,,1 (all zeros except for one "1") and the second phase right cycles the quantity 177777,,177776 (all ones except for one "0"). Note that the RM location R01 is renamed RM2 and it is loaded with the value 177776 (-2). This register and its value is used throughout the test. Note also that NBi where i IN[0..15] stands for logical complement of Bi. E.g., NB0 ==> NOT(B0) ==> NOT(100000) ==> 77777.
**Comments:** An error indicates that some right cycle failed. Check the code to see which cycle failed. RSCR2 holds the predicted result, and the one bits in T are in error.

**Title:** RFWFtest
**Functions Tested:** RF←, WF←, ←SHC
**Strategy:** This is an exhaustive test for loading SHC via the RF← and WF← paths. A very simplified mesa version is shown below:

```
FOR Q IN [0..377B] DO
    RF←Q;
    lastSHC←SHC;
    CheckShcValue[lastSHC, Q, performedRF];
    WF←Q;
    lastSHC←SHC;
    CheckShcValue[lastSHC, Q, performedWF];
    ENDLOOP;
```

The things to note about this are that
1) R01 is renamed R4BITMSK (and loaded with 17B),
2) Q is used as the loop index register,
3) RSCR is renamed LASTSHC and holds the value of SHC after the RF← or WF← instruction (since we use and therefore clobber SHC in the process of checking for correctness).

The tests are performed in line -- there is no "CheckShcValue" subroutine. A portion of the hardware manual appendix, which is duplicated below, is most useful when looking at the code that implements this test.

How SHC is loaded

| | shift count | Rmask | Lmask |
|---|---|---|---|
| RF← | 32-pos | 0 | 16-size |
| WF← | B[3],,pos | pos | 16-pos-size |

Note:
pos = BMux[8:11] (when SHC←B happens)
size = BMux[12:15] (when SHC←B happens)

shift count = SHC[3:7]
RMask = SHC[8:11]
Lmask = SHC[12:15]

Kernel 3: SHCtestRW Rlsh Tlsh Rrsh Trsh TRlcyTest RTlcyTest RFWFtest aluRSH aluRCY 3 aluARSH aluLSH aluLCY aluSHTEST

**Comments:** An error indicates that some RF← or WF← did not load SHC correctly. The labels near the ERROR indicate the problem; for example, a failure after RFLMask indicates that the LMask field of SHC was wrong after the RF←Q instruction. Check the code to see which shift failed. RSCR2 holds the predicted result, and the one bits in T are in error, and Q contains the value used in the RF← or WF← instruction.

**Title:** aluRSH
**Functions Tested:** RSH 1
**Strategy:** This test examines selected cases for right shifting the ALU output by one. Note that this shift does not use the hardware shifter -- it is controlled by selective loading of H3.
**Comments:** An error indicates that some right shift failed. In particular, there is a problem in the path between the alu output, the multiplexor that controls loading H3 and T. Check the code to see which shift failed. The one bits in T are in error.

**Title:** aluRCY
**Functions Tested:** RCY 1
**Strategy:** This test examines selected cases for right cycling the ALU output by one. Note that this cycle does not use the hardware shifter -- it is controlled by selective loading of H3.
**Comments:** An error indicates that some right cycle failed. In particular, there is a problem in the path between the alu output, the multiplexor that controls loading H3 and T. Check the code to see which cycle failed. The one bits in T are in error.

**Title:** aluARSH
**Functions Tested:** ARSH 1
**Strategy:** This test examines selected cases for arithmetic right shifting the ALU output by one. Note that this shift does not use the hardware shifter -- it is controlled by selective loading of H3.
**Comments:** An error indicates that some arithmetic (sign preserving) right shift failed. In particular, there is a problem in the path between the alu output, the multiplexor that controls loading H3 and T. Check the code to see which shift failed. The one bits in T are in error.

**Title:** aluLSH
**Functions Tested:** LSH 1
**Strategy:** This test examines selected cases for left shifting the ALU output by one. Note that this shift does not use the hardware shifter -- it is controlled by selective loading of H3.
**Comments:** An error indicates that some shift failed. In particular, there is a problem in the path between the alu output, the multiplexor that controls loading H3 and T. Check the code to see which shift failed. The one bits in T are in error.

**Title:** aluLCY
**Functions Tested:** LCY 1
**Strategy:** This test examines selected cases for left cycling the ALU output by one. Note that this shift does not use the hardware shifter -- it is controlled by selective loading of H3.
**Comments:** An error indicates that some cycle failed. In particular, there is a problem in the path between the alu output, the multiplexor that controls loading H3 and T. Check the code to see which cycle failed. The one bits in T are in error.

Kernel 3: SHCtestRW Rlsh Tlsh Rrsh Trsh TRlcyTest RTlcyTest RFWFtest aluRSH aluRCY 4 aluARSH aluLSH aluLCY aluSHTEST

**Title:** aluSHTEST

**Functions Tested:** RSH1, RCY 1, ARSH 1, LSH 1, RSH 1, LCY 1

**Strategy:** This is an exhaustive test for checking the ALU ouput shift and cycle functions. There is an outer loop that generates all possible 16 bit values. Each of the above functions is tested for all possible values.

**Comments:** An error indicates that some shift or cycle failed. In particular, there is a problem in the path between the alu output, the multiplexor that controls loading H3 and T. Check the code to see which operation failed. The one bits in T are in error, and Q contains the value that was shifted or cycled by one.

October 20, 1977 5:11 PM

**Title:** stkpPush
**Functions Tested:** stack&+1←, stkp←stkp-1, ←stack, stkp←stkp+1
**Strategy:** This is an exhaustive test of the stack push and pop operations. The basic approach is to use a loop to push known values onto the stack. Each time an item is pushed onto the stack, the value of stkp is checked, the item is poped off to see if that works and then the item is pushed again so that it is possible to proceed to the next loop iteration. Q holds the value being pushed onto the stack. The test is arranged so that the Qth location in the stack is loaded with Q. I.e., if Q=5, then the value at stkp=5 should be 5. Note that the act of pushing something onto the stack changes the value of stkp; consequently the test subtracts one from the value of stkp before checking it against Q.
**Comments:** The one bits in T are in error. Each error condition is commented in the code. There are extensive comments in the code that discuss the loop control algorithm.

**Title:** stkpM2
**Functions Tested:** stack&-2
**Strategy:** This is an exhaustive test of the stack -2 function. **IT DEPENDS UPON stkpPUSH!!!** This test presumes that the stack has been backgrounded with the values 0..76 in the same locations. I.e., if stkp=44, the value at the top of the stack is 44 and stack&-2 will position stkp to point to the value 42 as well as change the value of stkp to 42. The two cases that must be tested are when the stkp begins with an even number and when it begins with an odd number. The test proceeds as follows:

Initialization (set stkp, init Q to 74 or 73, init CNT, etc)
Inner Loop
    Read and check stkp
    Read and pop the stack -- check the the expected value was read.
    Check for underflow
Outer loop
    Set stkp to 73 and proceed OR exit if we have already done this.

**Comments:** The one bits in T are in error. Each error condition is commented in the code. There are extensive comments in the code that discuss the loop control algorithm.

**Title:** carry20Test
**Functions Tested:** CARRY20
**Strategy:** This is a selective test of the CARRY20 function. This function causes a 1 to be or'd into the carry-out bit that is used as input to bit 11 in the alu (remember the alu is bit sliced using 10181s). Given that there is not already a carry, this function has the effect of adding 20b to the value in the alu; when the alu operation causes a carry into bit 11 this function has no effect.
**Comments:** The one bits in T are in error.

**Title:** xorCarryTest
**Functions Tested:** XORCARRY
**Strategy:** This is a selective test of the XORCARRY function. This function causes the carry-in bit for bit 15 of the alu to be xor'd. Normally this bit is 0, and when it is one the alu arithmetic functions will see a carry into bit 15. For example, to accomplish twos complement arithmetic, the ALUFM is programmed to provide a "one" for this bit during A-B. By performing xorcarry at the same time as doing A-B the result will be one less than expected, ie, it will effect ones complement arithmetic.
**Comments:** The one bits in T are in error.

**Title:** savedCarry
**Functions Tested:** USEDSAVEDCARRY
**Strategy:** This is a selective test of the useSavedCarry function. This function causes the alu carry bit from the last instruction to be used as the carry-in bit to bit 15 during the current instruction. In absences of this function and others like it, the caryy-in bit for bit 15 is provided by the alufm and is usually zero. This is the bit complemented by the xorcarry function.
**Comments:** The one bits in T are in error.

**Title:** multiplyTest
**Functions Tested:** MULTIPLY
**Strategy:** This is a selective test of the multiply function. Multiply has three effects:

1. It causes alu output to be right shifted 1, with CARRY replacing bit 0.
2. It causes Q to be loaded with the quantity alu[15],,Q/2 (i.e., it uses the bit shifted out the alu for Q[0] and right shifts the rest of Q).
3. It causes Q[14] to be OR'd into TNIA[10] as a slow branch.

Note there are eight combinations of Q[14], carry, and alu[15]. All eight situations are tested (tests "A thru H"). In addition, two tests are performed by loading Q with alternating 01 and 10 to make sure that the shifts work properly.

MULCHECK is a subroutine that performs the multiply,
T ← T + (rscr), multiply,
and sets rscr2 to 0 if Q[14] branch did NOT happen, and sets it to one if the branch did happen.
**Comments:** The one bits in T are in error except in the case where there was an incorrect Q[14] branch.

**Title:** divideTest
**Functions Tested:** DIVIDE
**Strategy:** This is a selective test of the divide function. Divide has two effects:

1. It causes alu output to be left shifted 1, with Q[0] replacing bit 15.
2. It causes Q to be loaded with the quantity (Q/2),,carry (i.e., it left shifts Q by one and replaces old Q[15] with the carry from the alu operation).

Note there are four combinations of Q[0] and carry. All these combinations are tested by the code. There is one test with Q loaded with alternating 01 to check the shifting logic.
**Comments:** The one bits in T are in error.

**Title:** CdivideTest
**Functions Tested:** CDIVIDE
**Strategy:** This is a selective test of the Cdivide function. Cdivide is similar to Divide except the alu carry bit is complemented before it is loaded into Q. The are four combinations of Q[0] and carry are tested by the code. There is one test that checks the shifting logic by loading Q with alternating 01 before doing the Cdivide and then checks the shifted bits for correctness.
**Comments:** The one bits in T are in error.

**Title:** slowBr
**Functions Tested:** BDISPATCH
**Strategy:** This is an exhaustive test of the eight way slow dispatch function. Q holds the value that is or'd onto the Bmux.
**Comments:** The one bits in T are in error -- look at RSCR to determine which branch was really taken.

Kernel 4: stkpPush stkpM2 carry20Test xorCarryTest savedCarry multiplyTest divideTest 3 CdivideTest slowBr

December 26, 1978 1:29 PM

## Introduction

MemA contains diagnostics for each memory board in the Dorado. The code is organized by board type and the user can specify that specific diagnostics should or should not run by calling subroutines provided in MemA. MemA will run in an infinite loop until an error occurs. The descriptions below detail the diagnostics. There is no "A" board in the Dorado; MemA is just a name that indicates that All the memory system gets tested by the diagnostic. The pieces of MemA that test each board are described below. Each piece has a name like MemC or MemX. Those names correspond to the C and X boards in the memory system.

## Operating Procedures For MemA

### Geting Files and Starting Up

Load the dump file [ivy]<dl>memA.dm to obtain the newest version of MemA. The alto executive command,

midas MemA **cr**

causes midas to run and to load the diagnostics into the Dorado. "BEGIN;G" causes the diagnostics to run until an error occurs. The user may enable or disable specific board tests by calling the Midas subroutines named below (type "subroutineName()"). The default causes all the tests to run. There are routines that enable the diagnostics for each board, that disable the diagnostics for each board, that enable only a particular board's diagnostics, that disable all the diagnostics and that enable all the diagnostics:

| | | |
|---|---|---|
| addCboardTest | removeCboardTest | onlyCboardTest |
| addXboardTest | removeXboardTest | onlyXboardTest |
| addDboardTest | removeDboardTest | onlyDboardTest |
| addSboardTest | removeSboardTest | onlySboardTest |
| addAllTests | removeAllTests. | |

### Some conventions

Each test checks to see if "its bit" is true before it runs (remember there are numerous tests per board). The bits are kept in the word "memFlags" that is displayed on the screen. The "addBoard" subroutines work by ORing a bit mask into the current value of memFlags and the "removeBoard" subroutines work by ANDing a bit mask into the current value. The user may enable or disable selective diagnostics by setting the value of memFlags explicitly. The bit correspondances appear in the source file "memDefs.mc" and later in this document.

The source code includes a version of the diagnostic that is written in Mesa. This version of the diagnostic provides a "top level" view of how the diagnostic works. The especially peculiar details of the Mesa code usually correspond directly to the way the microcode is implemented. Symbols completely upper case are either Mesa reserved words (like WHILE, ENDLOOP, etc.) or indications of specific Dorado hardware operations (like PIPE2[], DUMMYREF←, FETCH←, etc.).

**sChaos**, a multi-tasking, random memory reference test that is one of the Sboard diagnostics is different from the other diagnostics because the user must enable task simulation (with "xorTaskSim()") for the test to execute.

## Changing the Test Environment

### Error Correction

The left half of the memFlags word contains control information for the diagnostics. One such bit, memState.useTestSyn, determines whether the diagnostics load the testSyndrome register with a zero or 200, the value that enables error correction. There are two subroutines that provide user access to this bit:

| | |
|---|---|
| **ECon** | Turns on memState.useTestSyn |
| **ECoff** | Turns off memState.useTestSyn. |

Changing memState.useTestSyn does not automatically cause error correction to turn on or off. The S board diagnostics examine that bit before setting testSyndrome during S board initialization, which occurs every time through the main test loop.

### Using Only One Column in the Cache

The storage diagnostics have the capacity to set MCR such that the same cache column always will be chosen for the victim in the event of a cache miss. This causes the cache to behave as if it has only one column. The contents of the rm register, sMCRvictim, control this facility. If the value of sMCRvictim is greater than 3, the diagnostics set MCR with zero, the default state (it implies full use of the cache). If sMCRvictim is IN [0..3], it defines the column of the cache that becomes the permanent victim.

The bit memState.usingOneColumn indicates that the storage diagnostic is using only one column in the cache.

### Testing the fast IO system with the fast IO test jig

There is a special board that may be used to test the fast io portion of the memory system. Since the board is optional and won't be present on all Dorados, as a default the diagnostics don't attempt to execute the fastio test. There are two subroutines that may be called from Midas to control running the fastio test: "**FIOtestOn()**" and "**FIOtestOff()**".

## After a Breakpoint at an Error

When an error occurs the user will want more information displayed on the screen. "Middle button" OLINK to determine the label associated with the error. This label should indicate which board diagnostic failed (the first letter of the name usually is the initial of the board). There are Midas command files that cause relevant information to appear for each test type:

showC, showX, showD, and showS.

For example, if an error occured at catColFindDL+6, the user should type, "showC" and then execute the "read-cmmds" menu item to cause Midas to show relevant registers and other data. Another example is XrwPipe3Er that indicates a failure in the map and the user should use "showX". Note that the cache data (D-board) tests use the two letter prefix, "cd". Please bring ambiguous labels to my attention. -- gene.

December 26, 1978 1:06 PM

**Title:** singleStep
**Functions Tested:** decoding for all non-emulator memory references
**Strategy:** Operator checking with single steping. The operator begins by typing "singleStep;G" to Midas. Then the code at singleStep sets mcr.disCF and mcr.disBR, turns on tasking, sets tpc[1] appropriately and then notifies task one. Task 1 runs and immideately hits a breakpoint. Now the operator may single step thru all the non-emulator memory references.
**Comments:** There are no automatic error indications. This sequence of instructions ends in a branch to the top (it is an infinite loop) EXCEPT that a test is made to check the value of T. The singleStep initialization code sets T to zero; if the code at the end of the singlestep sequence finds a non zero value in T, it blocks task 1. This code is useful for debugging new memC boards.

**Title:** emulMem
**Functions Tested:** decoding for all emulator-only memory references
**Strategy:** Operator checking with single steping. The code performs "map ←" and cache "Flush" references.
**Comments:** There are no automatic error indications. This code sequence is an infinite loop with no exit control.

**Title:** cPipeVa
**Functions Tested:** read and write the Pipe VA bits
**Strategy:** Set mcr.disBr, mcr.disCF, and mcr.noRef; then enter a loop that performs dummyRefs to write into the pipe, and read the pipe to see that the proper values were read back. The test uses a cycled one pattern to check each bit in the pipe.
**Comments:** An error at **cPipeVAerr** indicates that the test read something different from what it wrote. T contains the bad bits and rscr contains the value written into the pipe.

MemFlags.cPipeVa controls this test.

**Title:** cBRrwTest
**Functions Tested:** read and write the base registers
**Strategy:** For each base register test each bit position by using a cycled one pattern. The test sets mcr.disCF and mcr.noRef, and uses the subroutine setMbase to set the base registers. The current base register is set by the setBR subroutine and dummyRef is used to read the va from pipe0 and pipe1.
**Comments:** In both error conditions described below, Q contains the number of the base register failed when the error occurs. An error at **cBrLow16er** implies a failure in the low 16 bits of the current base register. The bad bits are in T and the expected bit pattern is in rscr2.

An error at **cBrHi8er** implies a failure in the high 8 bits of the current base register. The bad bits are in T and the expected bit pattern is in rscr.

MemFlags.cBR controls this test.

**Title:** cacheAddr
**Functions Tested:** read and write the cache A-memory
**Strategy:** For each pattern write the entire cache address memory with known values. Then for each row and for each column, write the memory again and check to see that the correct entry appeared in the pipe. Load MCR with dpipeVA←Vic, mcr.FDmiss, mcr.useMcrV, mcr.disCF and mcr.disHold. During the checking pass the target column is loaded with a different pattern than was originally written in the A-memory. UseMcrV selects the column, FDmiss forces a miss and dPipeVa causes the victim's A-memory to appear in the pipe.

**Comments:** This test uses the getPattern / nextPattern subroutines. An error at caBadHi15 indicates the pattern read from the pipe (high 15 bits of the va) was not the pattern expected. T contains the bad bits, rscr contains the high 15 bits from the pipe and rscr2 contains the expected pattern, va contains Mar value the test used, and Q contains the current cache row being tested.

CaBadRow indicates the row bits in pipe1 were not what was expected. The test keeps the value of the current row in q. The subroutine chkPipeRow performs the actual test, and returns an Alu result that gets checked for correctness. This error suggests a fundamental problem with the C board -- a different va was read from the pipe than what the program referenced.

MemFlags.cAmem controls this test.

**Title:** cacheCompr
**Functions Tested:** cache comparators
**Strategy:** Test the cache comparators with data from all the base registers. The outermost loop is the base register loop, next is the pattern loop and after that come the cache row and column loops that actually implement the testing. For each base register, pattern and row the following occurs:

> The entire row is backgrounded with the complement of the current pattern (use mcr.fdMiss, mcr.noRef and invoke the subroutine, "mcrForCol"). In the test loop mcr.fdMiss, mcr.noRef are used with mcrForCol to force the current pattern into the current column. The test sets mcr.disCF, mcr.noRef and Mcr.disHold, sets the current base register to the value of the current pattern and then performs a store reference. This reference causes the selected column to be written with the current pattern (the other columns have been backgrounded with a different pattern). Now the test performs a reference with the current pattern, which should match the current column entry in the cache. The test invokes chkPipe5col to determine if the expected column was chosen by the cache.

**Comments:** An error at ccBadCol indicates the cache chose a different column from the one expected. T contains the bad bits, rscr contains the value of pipe5 and col contains the value of the expected column. Q contains the cache row currently being tested and va contains the current Mar offset.

MemFlags.cComprs controls this test.

**Title:** cFlagsTest
**Functions Tested:** read and write the cache flags like a memory
**Strategy:** For each column and row entry in the cache, test all the possible single bit cache flag values (cycled one bit pattern). The following, enigmatic method for reading and writing the cache flags reflects the fact that the flags weren't designed to be read or written like a memory. Write the cache flags as follows:

> Turn on mcr.fdMiss and mcr.disHold in MCR. Use mcr.useMcrV to select the current column. For this explanation, PRESUME the cflags value has been left shifted by four to align it with the proper position in the cache.
>
> Let va = vaForRow[row] - flagsValue and write the low 16 bits of the current base register with va. Now perform the following reference:
>
> ```
> dummyRef ← flagsValue;
> CFLAGS ← flagsValue; * this write the flags
> ```

To read the cache flags perform the following:

> Note that the current contents of the A-memory must be known and there must not be two hits in the cache when the reference occurs.

Disable hold and use mcr.UseMcrV to force a different column for the victim. Perform the reference,

```
dummyRef ← va;
pipe5 ← PIPE5[];   * read the flags
```

where the PIPE5[] occurs in the instruction following the dummyRef, and va[0:14] are known to be in the A-memory of the cache. Check that col = the column that should have matched and not the column that was chosen as victim! Remember the flags value was returned in the PIPE5[] reference.

**Comments:** In both errors below, Q contains the cache row currently being tested and va contains the current Mar offset. An error at **cfBadCol** indicates the column of the victim was not the column that should have been chosen (there should have been a hit). The bad bits are in T and the value of PIPE5[] is in rscr.

An error at **cfBadFlags** indicates the flag bits that the test read did not match the bits the test wrote. The bad bits are in T and the expected bits are in flagsV. Rscr contains PIPE5[].

MemFlags.cFlags controls this test.

**Title:** cacheAddrTest

**Functions Tested:** test the cache addressing mechanism

**Strategy:** This test checks that the addressing mechanism (as opposed to the bits that hold cache address values) works. The algorithm works as follows:

Zero the A-memory and the cache flags.

Ascend thru the A-memory and the cache flags. Check that the current value is zero and then set both to all ones. If the current value is not zero, an earlier store clobbered this entry. In this case, perform the "findUP" test.

Zero the A-memory and the cache flags.

Descend thru the A-memory and cache flags: check that the current value is zero and then set both to all ones. If the current value is not zero, an earlier store clobbered this entry. In this case perform the "find DOWN" test.

If there were no problems the addressing mechanism works.

FindUP: zero the A-memory and the cache flags.

Ascend thru the A-memory and cache flags: before setting the current location to all ones, check that the earlier, clobbered location is still zero. If it is not zero, the *previous* store clobbered that location.

FindDOWN: same as FindUP excep descend thru the A-memory and cache flags.

**Comments:** An error at **catUpCFerr** indicates that the cache flags were improperly written. The old, clobbered cache address is packed in "va": The three least significant bits contain the column number of the error address, and the remainging bits contain the row number. Hence, if va = 132, the clobbered address is row 13, column 2. Q contains the current row index and col contains the current column index. Note that the previous reference was the one that clobbered the cache entry. Consequently the offending address is "the current address minus one". A column value of zero implies the third column of the preceeding row is the offending address.

An error at **catUpAddrErr** indicates the wrong A-memory was chosen. The same register conventions hold.

An error at **catDownCFerr** or **catDownAddrErr** is similar except the offending is "the current address plus one".

There are numerous subroutines that support this test. SetCAAF sets the current cache A-memory and cache Flags to all ones. ReadCurrentCflags and readCurrentCAmem return the contents of the flags and A-memory, respectively, for the current address. ReadOldCflags and readOldCAmem perform the same function as the preceeding two except that the address is composed from the

packed address contained in va (the old row, old column).
MemFlags.cAddr controls this test.

March 20, 1978 1:07 PM

**Title:** mapRWtest
**Functions Tested:** read and write the map
**Strategy:** Read and write all the bits in the map. To avoid refresh problems this test touches each row of the map at least once every two milliseconds -- this test can be run with refresh disabled. The test uses the following loop structure:

> The outer loop is the pattern loop, followed by a loop that sets va[0:1]. The next loop is the column loop, followed by the row loop where the test performs its testing. By keeping the column loop outside the row loop, the test touches all the rows of the map once before it switches columns. This enables the test to meet the refresh requirements.

The test writes the current pattern into the current column for every row, then it checks the current column for every row to make sure the returned value is correct.

**Comments:** An error at **XrwPipe3Er** indicates that the test read a different value from the map than it wrote. The bad bits are in T and the expected value is in Mpat. The value from the read map operation is in rscr.

An error at **XrwPipe4Er** indicates that the va[0:1] bits read from the pipe4 operation did not correspond to what the test wrote. T contains the bad bits, rscr2 contains the value returned from pipe4 with the pipe4.va01 field right justified. Mva01 contains the current test value for va[0:1].

MemFlags.mRW controls this test.

**Title:** mapAddrTest
**Functions Tested:** map addressing mechanism
**Strategy:** This test follows the same pattern as the addressing tests for the C, D and S boards follow:
**(Ascent Test)** Zero the memory, then ascend the memory as follows:

1) If the current location is non-zero, there was an addressing error (a previous store to an "earlier" location clobbered the current location). Invoke the ascent error routine which finds exactly which store caused this error.
2) Otherwise, set the current location to all ones.
3) Increment the current address: if all addresses have been tested, proceed to the descent addressing test, otherwise proceed to step 1.

**(Descent test)** Zero the memory, then proceed as in the ascent test, except we descend through the memory (decrement) rather than ascend (increment).
**(Ascent error routine)** Denote the "error address" as the clobbered address discovered in the ascent test. The ascent error routine procededs by zeroing memory then ascending the memory as follows:

1) If the error address is non zero, the last store which the ascent error routine made clobbered that location (storing into the "current address-1" clobbered the "error address" that was detected by the ascent test.
2) Otherwise, set the current location to all ones.
3) Increment the current address; if all addresses have been tested, there must have been an intermittent error, otherwise proceed to step 1.

The **descent error routine** is similar to the ascent error routine in the same way.
**Comments:** An error at **xUpErr1** indicates that the **xZeroMap** routine failed to zero the first entry in the map. This is an extra check to simplify error interpretation when there are fundamental problems with the map. An error at **xErrUp2** indicates that the previous store (MpageX-1) clobbered the error location (kept in stack). An error at **xErrUpIntermittent** indicates that the ascent test found that the map entry at stack was clobbered and the ascent error routine was unable to recreate the problem.

An error at **xDownErr2** indicates the previous store (MpageX-1) clobbered the error location (kept in stack). An error at **xErrDownIntermittent** indicates that the descent test found that the map entry at stack was clobbered and the descent error routine was unable to recreate the problem.

MemFlags.mAddr controls this test.

**Title:** mapRWtest2

**Functions Tested:** timing characteristics for the map row

**Strategy:** This time-consuming test requires the operator patch-out a jump at the beginning of the test to the end of the test. The basic idea is to write a map entry and then wait a long time and see if the data is still valid. This test is useful for determining how long the map memory chips hold their data. Refreshing should be disabled when this test runs. The test should not encounter an error.

The outermost loop controls the number of cycles the test waits after writing a map entry. The pattern loop, va[0:1] loop, row loop and then column loop follow. Note that the test writes a single map entry, waits the current amount of time and then checks the entry. The test takes about 20 minutes to run. The subroutine, "testMap", performs the actual testing.

**Comments:** An error at **xTestMapErr3** indicates that the pattern written into the map was not the same pattern read back. Mrow contains the map row, and Mcol contains the map column. Q contains the number of cycles testMap waited after writing the map, Mpat contains the pattern expected, T contains the bad bits, and rscr contains the value read from pipe3.

An error at **xTestMapErr4** indicates that the va[0:1] field of the map was incorrect. T contains the bad bits, rscr2 contains the value of va[0:1] (right justified) as read from pipe4, and Mva01 contains the current value for va[0:1].

March 20, 1978 11:15 AM

**Title:** cDataTest1
**Functions Tested:** Read and write the bits of the cache data memory
**Strategy:** This test writes and reads the cache data memory. The outer loop is the pattern loop (cycled 1s followed by va). Within the pattern loop, the test writes the entire cache and then reads the entire cache, checking for an error.
**Comments:** An error at **cd1rerr1** indicates that two reads of the cache data don't match: the test performed a regular and immediate read of MD. CData contains the value returned from a regular read and T contains the value returned from MDI. Va contains the virtual address that the test checked.

An error at **cd1Rerr2** indicates that the data returned by the memory system did not match the data written into the cache. T contains the bad bits, CData contains the pattern written into the cache and rscr contains the value returned by the memory system.

MemFlags.dRW controls this test.

**Title:** cdaTest
**Functions Tested:** tests the cache data addressing
**Strategy:** The algorithmic description of **cacheAddrTest** for MemC applies to this test.
**Comments:** **CdaUpErr** indicates that there was an addressing error. Va minus one is the address that clobbered the contents of the address at r1.

An error at **cdaUpNoFind** indicates there was a transient error that caused the the address in r1 to be clobbered. The test was unable to replicate the addressing problem.

An error at **cdaDownError** indicates that there was an addressing error such that va plus one is the address that clobbered the address in r1.

An error at **cdaDownNoFind** indicates there was a transient error that caused the the address in r1 to be clobbered. The test was unable to replicate the addressing problem.

MemFlags.dAddr controls this test.

**Title:** cdHoldTest
**Functions Tested:** Check that the memory holds the processor when appropriate.
**Strategy:** Fetch a value we know under different timing circumstances to make sure the memory system returns the value properly. These test try to retrieve a known value with zero, one or two noops between fetch← and ←md. The test checks both md and mdi. The test turns off wake-ups.
**Comments:** An error at **cdHoldErr0**, **cdHoldErr1**, or **cdHoldErr2** indicates that the ←MD test received a value different from the one it expected. T contains the bad bits, rscr contains the expected value and r1 contains the address.

An error at **cdHoldMdiErr0**, **cdHoldMdiErr1**, or **cdHoldMdiErr2** indicates that the ←MD test received a value different from the one it expected. T contains the bad bits, rscr contains the expected value and r1 contains the address.

MemFlags.dHold controls this test.

October 3, 1978 10:24 AM

**Title:** sDtest

**Functions Tested:** read and write memory storage boards

**Strategy:** Read and write all the bits all the bits on the storage boards, and read the pipe to check the error correction bits. The basic idea is to write all the words in the memory with the current pattern and then to read all the words in the memory and to check that the data read is the same as the data written.
There are three classes of patterns:

Cycled one bit
Virtual address as data
Random numbers

The diagnostic checks all the words in a munch before it reads the pipe to see if there's been a failure in the error correction bits or if there's been a successfully corrected failure.

**Comments:** An error at **sVaRerr** indicates that the pattern read from the memory is not the same as the pattern written. T contains the bad bits, sva contains the virtual address that failed, rscr contains the current pattern and rscr2 contains the value returned by the memory system.

This test presumes the cache works since no effort is made to guarantee that the bits checked come from the storage board rather than being left over in the cache (writing the entire memory should flush the cache many times).

Data errors caught by the code that reads the pipe require the user to use Midas to fully understand what has occured. The "showS" command file places the Midas pseudo registers PFAULT 20, PVA 20, PERRS 20, PREF 20, and PMAP 20 on the screen (see Midas documentation). These registers represent the "current" pipe values. Unless the operator has specifically intervened, PROCSRN will be zero and the registers described above will display information relevant to the emulator. Applying the "middle button" to the values in those registers will provide important information relating to the failure. The diagnostic has some idea what has occured and errors occur at different labels depending upon what the diagnostic thinks happened.

An error at **sVaChkBitErr** indicates the diagnostic believes there was a failure in the check bits in a quadword for the current munch. **sVaDblErr** is the location that indicates the diagnostic believes there was a double error, and **sVaSingleErr** indicates there was a single error that was undetected by the preceeding data check (this occurs when the error corrector has been enabled). **sVaUnknown** represents the detection of a problem too complicated for the diagnostics to comment upon.

MemFlags.sRW controls this test.

**Title:** sAddrTest

**Functions Tested:** check storage addressing

**Strategy:** The algorithmic description of **cacheAddrTest** for MemC applies to this test.

**Comments:** An error at **sAddrUpFindL** + 11 indicates that an addressing error was found. The bad value is in rscr2, and the address of the clobbered location is in rscr,,t. The address that caused the clobber to occur is in sVaHiX,,sVa.

An error at **sAddrDownFindL** + 12 has the same implication and register conventions as the preceeding error label.

MemFlags.sAddr controls this test.

**Title:** sChaos
**Functions Tested:** multi tasking, random references

**Strategy:** This test backgrounds two rows of the cache with munches that are chosen at random. The dirty bit for each munch is also selected at random. The default task and the task simulator execute one of four possible unique tests based upon random selection. Each test contains a reference (fetch or store) followed by a delay (chosen at random), followed by another reference. Chaos backgrounds memory with its own address (mem[sva] ← sva), and all stores performed by the tests follow that pattern. An error occurs when selected registers don't have the same contents or when the contents of a memory location don't correspond to the contents of the corresponding RM location.

**Comments:** Chaos, unlike the other storage board diagnostics requires more than just its "memFlags" bit to be set: Chaos requires that task simulation be enabled before it runs. Enable task simulation (use **xorTaskSim**) to run Chaos as an S board diagnostic.

Each task performs two references separated by a delay. Regardless of which test chaos selected, certain pairs of registers should contain the same values, and storage should contain the same values for those addresses. If this is not true there has been a hardware failure. The following should be true:

```
mem[rm00] = rm00 = rm01
mem[rm02] = rm02 = rm03
mem[rm10] = rm10 = rm11
mem[rm12] = rm12 = rm13
```

The RM name provides some information about the use of the RM location. For example, rm10 is the first rm location used by the simulator task, and rm02 is the third RM location used by the default task. The pattern of usage is always the same.

If the pattern is a fetch:

```
FETCH ← rm?0
(delay)
rm?1 ← MD;
or,
FETCH ← rm?2
rm?3 ← MD
```

If the reference is a store:

```
STORE ← rm?0, MD ← rm?0
or,
STORE ← rm?2, MD ← rmd?2
```

**What To Do If An Error Occurs**

Type "showChaos" and select the "read-cmmds" menu item. This command file causes Midas to display all the "RMxx" registers and two IM locations, **chaos0Stageit** and **chaosSimStageit**. Middle button the value fields of these two IM locations to determine which two tests chaos invoked to cause the current error. Examine the code listings.

The tests are named in a consistent fashion:

chaosTsk<i>Tst<j> denotes the task and test number. If I is zero, the test was run by the default task, otherwise it should be one and it denotes the simulator task. J denotes the test number within the task. Tests are numbered on the basis of the two reference patterns:

| | |
|---|---|
| test 0 | fetch-delay-fetch |
| test 1 | fetch-delay-store |
| test 2 | store-delay-fetch |
| test 3 | store-delay-store |

The implementation actually uses a two bit field to generate the test number where zero names a fetch reference and one names a store reference.

MemFlags.sChaos controls this test.

**Title:** fioTest

**Functions Tested:** fast IO portion of Storage system

**Strategy:** Use the fast io jig to simulate a fast IO device. This test requires the presence of the *optional* io test jig; consequently, the default condition in the diagnostics is that the fioTest does not execute. The subroutines **"FIOtestOn()"** and **"FIOtestOff()"** will enable and disable this test.

The code checks that the subtask mechanisms in the processor and memory system work properly, then it checks that storage transports work properly. Remember that subtask[0:1] gets or'd into both (RBASE.3, rstk.1) and (memBase[2:3]). The diagnostic uses two control loops, one for the task and another for the subtask. The emulator level code sets tpc for the current 'test' task and notifies that task. The rest of the test then executes as the current, non-emulator task; it sets the fio jig to the current task and subtask value, performs various tests and blocks when it is done. Then control returns to the emulator which processes the next subtask or task as required. The diagnostics initialize BR[i] to contain i*1000B (eg., BR[3] = 3000B) so that references made through the various membases can be checked.

**Comments:** The diagnostic sets rbase and membase to zero when it operates under the influence of the fio jig. Thus a reference to rmx0 may reference the r-register at 0, 4, 10b or 14b depending upon the subtask value in the fio jig. An address denoted *subtask relative i* refers to the i-th location *as seen by the subtask.* The diagnostic computes the effective address of subtask relative locations when it checks the results of some operation that was performed while "under the influence". Diagnostic checking occurs while the fio jig is disabled.

An error at **fioSTiErr** (for i = 0,1,2,3) indicates that the diagnostic expected to reference the subtask relative, zero-th RM location for the current, i-th subtask, and it failed. The diagnostic works by setting RBASE to zero and then performing "t ← rmx0" where the low three address bits of rmx0 are zero. T contains the value read from Rm. The first 20B locations in RM are backgrounded with their address (ie., RM[0] = 0, RM[1] = 1, etc), so the value read should indicate which rm locaction was sourced. An error at **fioSTiErr2** indicates that subtask i expected to clobber a specific rm location; it failed. In particular, the diagnostic performed, "rmx0 ← 77B" and the value in subtask relative rmx0 was not 77B. Note that the current value of the subtask determines which RM location should have been clobbered. AtestTaskX, and AsubTaskX are the RM locations that contain the current task and subtask being tested.

**fioSerr1** indicates that the sequence,

```
IOfetch ← 0;
IOfetch ← 20B;
IOstore ← 0;
IOstore ← 20B;
```

somehow clobbered one of the first *subtask relative* 32 words in storage. The initialization for storage is mem[i] ← i. At this error location, T contains the effective address being checked, sva contains the subtask relative address, and rscr contains the data fetched from memory. Remember that current value of AsubTaskX determines which subtask was active and that determines which base register was used.

The two labels **fioSerr2a** and **fioSerr2b** suggest that the IOfetches performed by the diagnostic did not fetch a dirty munch in the cache.
The diagnostic performed the following sequence,

```
mem[0] ← NOT(mem[0])
mem[20] ← NOT(mem[20])
IOfetch ← 0;
IOfetch ← 20;
IOstore ← 0;
IOstore ← 20
```

T contains the address that failed, rscr contains the value fetched, and rscr2 contains the expected value.

An error at **fioSerr3** indicates that an IOstore failed to overwrite a dirty munch in the cache. T contains the address with the "old" and incorrect value, rscr contains the bad value (T contains the expected value as well as address).

MemState.FIOtest controls this test. Note this is MemState, not MemFlags. MemState is the upper half of the MemFlags word.

March 2, 1978 8:05 AM

### Introduction

Preamble is a microcode file inserted at the beginning of all diagnostics. It provides macro and constant definitions that are used throughout all the diagnostics. It also predefines a few RM locations. In general preamble provides constants for each bit position in a 16-bit word, loop and skip macros, and subroutine entry and exit macros. These and others are described below.

## RM definitions: defaultRegion, randomRM, rm2ForKernelReturn

Kernel's main RM region is called defaultRegion. Preamble declares the following register name and value pairs in defaultRegion:

| | |
|---|---|
| R0 | 0 |
| R1 | 1 |
| RM1 | 177777 |
| R01 | 52525 |
| R10 | 125252 |
| RHIGH1 | 100000 |
| RSCR | 0 |
| RSCR2 | 0. |

Note that the various pieces of diagnostic code can and do declare different regions and sometimes redefine the name of one of the RM locations in defaultRegion.

The region named randomRM provides constants for the random number generator that is a part of the postamble package. The region rm2ForKernelRtn, which appears in preamble as well, declares two more registers for the random number generator. The rest of its storage is devoted to saving return link values.

Micro, the assembler for Dorado microcode, requires that register names be declared before they are used in the program. Since there are macros in preamble that use the "random number" RM names, the regions declaring those names are declared in preamble. Preamble declares defaultRegion to make it available to all microprograms.

## Constants and Parameters

A parameter is a name that has a numerical value. Assembler details of no interest here force the programmer to distinguish between parameters and B-mux constants. Consequently there is a collection of parameter and constant (B-mux) declarations in preamble. They define all the bits (b0, b1, etc), the compliment of all the bit positions (nb0, nb1, etc), and a collection of useful constants:

| | |
|---|---|
| CM1 | 177777 (minus 1) |
| CM2 | -2 |
| C77400 | 777400 |
| C377 | 377 |

## Loop, Skip, and other Branch Macros

In the following descriptions, "cond" refers to a fast branch condition like "alu=0" or "r odd".

| | |
|---|---|
| **noop** | branch[.+1] |
| **skip** | branch[.+2] |
| **error** | branch[err] |
| **skiperr** | branch[.+2]<br>branch[err] |
| **skpif[cond]** | branch[.+2, condition] used after a test of some sort |
| **skpunless[cond]** | dblbranch[.+1, .+2, cond] where "cond" is a fast branch condition. |

This means IF cond THEN goto .+1 ELSE goto .+2.

**loopuntil[cond, label]** This is just a branch condition. Diagnostics use this sort of macro to emphasize the presence of a loop: IF cond THEN goto .+1, ELSE goto label.

**loopwhile[cond, label]** IF cond THEN goto label ELSE goto .+1.

## Subroutine Entry and Exit Macros

**saveReturn[rmLocation]** place the return link in T and store the link into rmLocation. The macro goes through T first because that makes it possible to store the link value into a different rbase region from the current one. The assembler is left in "top level".

```
t ← not(notLink);
top level[];
rmLocation ← t;
```

**saveReturnAndT[rmLocationForRtn, rmLocForT]** First this macro stores the value of T into rmLocForT, then it saves the return link as saveReturn saves it. The assembler is left in "top level".

```
rmLocForT ← t;
t ← not(notLink);
top level[];
rmLocation ← t;
```

**returnUsing[rmLoc]** This macro fetches the value of rmLoc, stores it into link and returns. Note that the macro presumes rmLoc is located in a different rbase from the current one. Consequently it first sets rbase to the rbase of rmLoc, loads link, and then returns while setting the value of rbase to defaultRegion. The assembler is left in subroutine mode.

```
subroutine[];
RBASE ← rbase[rmLoc];
link ← rmLoc;
return, RBASE ← rbase[defaultRegion];
```

**returnAndBranch[rmLoc, registerName]** This variant on returnUsing performs the assignment "registerName ← registerName" in the return instruction. This enables the caller of the subroutine to perfrom a fast branch test at the point of return. RegisterName must be defined in the default region. The assembler is left in subroutine mode.

```
subroutine[];
RBASE ← rbase[rmLoc];
link ← rmLoc;
RBASE ← rbase[defaultRegion];
return, registerName ← registerName;
```

## Miscellaneous Macros

**zeroHold[registerName]** This macro zeros registerName and then sets the "hold register" three times. Hold implements task simulation and hold simulation. Because of problems associated with task switching its necessary to set hold three times before the programmer can be certain it is zero.

**getRandom[]** This macro returns a random number in T and leaves the assembler in the defaultRegion. The interface to the random number generator is rather peculiar, and this macro makes the calling conventions invisible.

January 11, 1978 6:19 AM

## Introduction

Postamble is a microcode file that is inserted at the end of all diagnostics. It provides the main, "outer" loop mechanism for all diagnostics along with several other functions. It also provides a small number of common subroutines for all diagnostics. Postamble controls the use of the hold and task simulators in the Dorado processor. The hold simulator causes periodic HOLDs and the task simulator periodically awakens a task determined by jumpers on the backplane. If the operator enables hold and task simulation, postamble causes the values used in those simulators to change each time the diagnostics complete a pass. Postamble implements task circulation, the situation whereby the diagnostics run as task 0, then as task 1, etc.

The operator controls the simulator and the task circulation by modifying the data word "flags" that is kept in IM. Postamble uses flags as a bit mask to control the simulators and task circulation.

**Flags Value Operation**

| | |
|---|---|
| 1 | use task simulator |
| 2 | use hold simulator |
| 4 | circulate tasks |

The value 7 will cause postamble to use both simulators and to circulate tasks.

**Label:** rmCheck
**Action:** guarantee "known" rm values are correct
**Strategy:** This code assures that the known rm values have are still valid before the next pass is made. Either a hardware bug or a software bug could cause the value of one of the known rm registers to be wrong.
**Comments:** This test occurs upon intering postamble.

**Label:** chkTaskSim
**Action:** generate next task simulator value.
**Strategy:** This code generates the next value for the task simulator. It generates zero if task simulating is turned off.
**Comments:** Calls the subroutine checkFlags and readByte3. This code always updates taskFreq, the IM location that holds (right justified) the value placed into the task simulator.

**Label:** chkHoldSim
**Action:** generate next hold simulator value.
**Strategy:** This code generates the next value for the hold simulator. It generates zero if hold simulating is turned off.
**Comments:** Calls the subroutine checkFlags and readByte3. This code always updates holdFreq, the IM location that holds (right justified) the value placed into the hold simulator.

**Label:** simControl
**Action:** construct task and hold simulator value.
**Strategy:** This code generates the next value for the hold/task simulator. Note that it does not load the simulator.
**Comments:** Calls the subroutine readByte3. This code always updates holdValue, the IM location that holds the value actually placed into the hold/task simulator.

**Label:** simInit, simSet, simBlock
**Action:** load task and hold simulator.
**Strategy:** This code runs at the highest priority task and actually loads the simulator. Postamble initializes this code at the simInit entry where the current value for the simulator is placed into T. SimSet is the location where the postamble actually loads the simulator and simBlock is the location where it blocks waiting for the next simulated task awakening to occur.
**Comments:** This code is not task 0 code, and it initializes itself by using the current value of Q for the next value for the hold/task simulator. The value is kept in T during normal operation.

**Label:** taskCirculate
**Action:** cause different tasks to execute the diagnostics.
**Strategy:** If task circulating is enabled, this code causes the diagnostics to run at the next task level. Task levels are chosen in turn, ie., task-0 runs the diagnostics once, then task-1 runs them, etc. Note that there is a conflict with the task simulator. If the simulator is active, the diagnostics should not execute at the same task level that the simulator awakens since the code at that task level is responsible for restarting the task simulator. **TaskCirculate assumes that the task simulator awakens task-15!** When simulating is true, task circulation stops after task-14 and then starts again at zero. When simulating is false, it stops after task-15 and starts again at zero.
**Comments:** This code calls checkFlags, checkTaskNum, and notifyTask.

**Label:** readByte3
**Action:** subroutine: t=IM addr; return byte 3 of IM
**Uses Registers:** rscr, t
**Strategy:** This subroutine reads the last byte (byte3, least significant byte) of IM location at "t".
**Comments:** This code disables hold simulation while it runs.

**Label:** getIMRH
**Action:** subroutine: t=IM addr, return right half of IM
**Uses Registers:** rscr, rscr2, t
**Strategy:** This subroutine reads and returns in t the contents of the right half of IM pointed to by T at time of entry.
**Comments:** This code disables hold simulation while it runs.

**Label:** getIMLH
**Action:** subroutine: t=IM addr, return left half of IM
**Uses Registers:** rscr, rscr2, t
**Strategy:** This subroutine reads and returns in t the contents of the left half of IM pointed to by T at time of entry.
**Comments:** This code disables hold simulation while it runs.

**Label:** checkFlags
**Action:** subroutine: t=flag bit mask, return w/ fast br condition, t = flags that are on and in the mask.
**Uses Registers:** rscr, rscr2, t
**Strategy:** This code reads the flags word and masks its contents with the contents of t. This is done so that the returnee can do a fastbranch skip to see if any of the flags in the mask were on. T contains the flags that were in both the mask and the flags word. Eg.:

```
t ← flags.taskSim;
call[checkFlags];
skpif[alu#0];
branch[notSimulating];
```

**Comments:** This code disables hold simulation while it runs.

**Label:** checkTaskNum
**Uses Registers:** rscr, rscr2, t, **r1**
**Action:** subroutine: t=expected val for current task, return t= real value of current task, returnee can do fast branch for equality
**Strategy:** This code returns in T the task number of the task that postamble believes is currently executing. The expected task number is compared with the real task number so the returnee can do a fastbranch skip to see if they are the same: E.g.:

```
t ← 3c;
call[checkTaskNum];
skpif[alu#0];
branch[notInTask3];
```

**Comments:** This subroutine calls readByte3.

**Label:** notifyTask
**Action:** subroutine: notify the task indicated by T
**Uses Registers:** rscr
**Strategy:** Use a bigBdispatch to notify the task indicated by the contents of T.
**Comments:** Return link kept in rscr.

**Label:** setHold
**Action:** subroutine: set the task/hold simulator registers w/ value of T
**Uses Registers:** q, t, setHoldRtn
**Strategy:** This subroutine causes the task at simTaskLevelC (a constant defined in postamble, it should be the task number of the task awakened by the task simulator) to initialize the task/hold simulator registers.
**Comments:** This code enables and disables the task/hold simulator.

**Label:** displayOff
**Action:** subroutine: Turns off interim display
**Uses Registers:** t
**Strategy:** This code turns off the interim display. It is for interactive use with midas while doing hardware or microcode debugging. The code sets a breakpoint at its exit so that the user can "call" the subroutine from midas to turn off the display.
**Comments:** "Midas subroutine".

April 13, 1978 7:45 PM

# Introduction

Any microcode subroutine may be called from Midas or used in patches to a diagnostic. This is an alphabetically ordered list of subroutines that have proven to be useful in various circumstances. They assume that RBASE = defaultRegion. Unless otherwise specified, single parameters are passed in T and single results are returned in T.

**addAllTests()** This enables all the memory diagnostics.

**add?BoardTest()** (Replace ? by C, X, S, or D) This enables the set of tests associated with the appropriate memory board.

**cacheAforVa(va)** Returns the midas style cache address that corresponds to va.

**chkRunSimulators()** Checks the values of flags.taskSim, flags.holdSim, flags.taskCirculation, flags.Conditional, and flags.conditionOK. This routine turns on hold simulation or task simulation if the proper bits are enabled. It does the same for task circulation. If flags.Conditional is true, none of the simulators will run unless flags.conditionOK is true also.

**clearCacheFlags()** Sets all the cache flags to vacant.

**cRowForVa(va)** Returns the cache row index for va.

**dirtyWriteLoop** This midas oriented code is seful during S-board checkout. Begin execution at "dirtyWriteLoop". After a breakpoint occurs, the map has been initialized. At that point the user must initialize sva to the address of interest. The code sets the cache A memory for the appropriate row to four different values that are sva+1000, +2000, etc. The code sets Cflags to vacant. Then a store followed by a long wait occurs. At the end of the wait, the test loops to the point where it sets the cache A memory.

**disableConditionalTask()** Clear flags.conditionOK, set flags.conditional and cause the conditional tasks to stop running.

**displayOff()** Turns off the interim display.

**doScheckOut** This midas oriented code is useful during S-board checkout. After initialization it enters an infinite loop that reads and checks two storage locations. The code has a patch location (**scodF**) that may been changed to force an infinite loop of write/read transports. There are numerous registers whose initial values may be manipulated to obtain the desired behavior.

**ECoff()** Clear memState.useTestsyn. This causes the storage diagnostics to run without error correction.

**ECon()** Set memState.useTestSyn. This causes the storage diagnostics to run with error correction enabled.

**enableConditionalTask()** Sets flags.conditional and flags.conditionOK. Routine also causes the appropriate tasks to run if they are enabled.

**iDboard()** Initializes the Dboard for the memD diagnostics.

**iSboard()** Initializes the storage boards. This presets the map, clears the cache flags and sets testSyndrome according to memState.useTestSyn.

**longWait(nCycles)** Returns after waiting nCycles.

**l?** (Replace ? by 1 through 5.) Code patch locations allocated in memA. The length of the location is denoted by the numerical suffix. The last location contains a branch to the top (eg., l4+3 contains "branch[l4]"). The other locations contain noops.

**notifyTask(taskNum)** Notify task "taskNum".

**only?BoardTest()** (Replace ? by C, X, S, or D) This disables all memory diagnostics and then enables only the set of tests associated with the appropriate memory board.

**patch?** (Replace ? by 1 through 5.) Code patch locations allocated in postamble. The length of the location is denoted by the numerical suffix. The last location contains a branch to the top (eg., patch4+3 contains "branch[patch4]"). The other locations contain noops.

**presetCache(hi8, low16, CFLAGS)** Sets each cache A entry with hi8,,low16 and sets the cache flags to CFLAGS. Note that the low 9 bits of low16 will be ignored. Think of this routine as taking a 24-bit virtual address and writing the appropriate cache entry w/ that address and with CFLAGS. Parameter Conventions:

T = hi 8
rscr = low 16
rscr2 = CFLAGS

**presetMap()** Resets the map, then initializes it so that virtual addresses correspond to real addresses (ie., virtual page 0 maps to real page 0, etc).

**resetHold()** Sets the hold simulator to the value in holdValueLoc in IM.

**removeAllTests()** This disables all memory board diagnostics.

**remove?BoardTest()** (Replace ? by C, X, S, or D) This disables the set of tests associated with the appropriate memory board.

**resetMap()** Kick-starts the map automata and initializes the tag bits.

**setMbase(brIndex)** Sets current memBase to brIndex. This routine clobbers and then restores the value in r0.

**setTestSyn(val)** Sets testSyndrome to val.

**sGetConfig()** Initializes the registers used by various loop control mechanisms that need to know how much memory is available. This routine tells the diagnostics what the memory configuration looks like.

**sMCRvictim** This Rm location causes the storage diagnostics to run with MCR set up to restrict all victimes to a particular column. This makes the cache one column wide. If sMcrVictim > 3 the default of allowing the entire cache is used.

**testMap(column, row, pattern, cycles, callResetMapFlag)** This routine writes a pattern into the map at column,row, waits nCycles and then read the map to see if the pattern is still in the map. If callResetMapFlag is true (#0) then the map is reset just before writing the pattern into the map. Parameter conventions:

Mcol = column
Mrow = row
Mpat = pattern
rscr = nCycles
rscr2 = callResetMapFlag

**testTaskSim(taskSimVal)** Midas subroutine for testing the task simulator. Causes the hold register to be loaded with taskSimVal and then enters an infinite loop examining t. If t is non zero the simulator task has clobbered the current task's T. If a breakpoint occurs anywhere else, there has been a wild transfer of control.

**vacateCacheRow(va)** Set CFLAGS to vacant in the cache row that corresponds to va.

**vaForRow(cacheRowIndex)** Returns a virtual address that will hit the cache row, chacheRowIndex.

**xBoardLoop** This routine is a midas oriented piece of code that enters an infinite loop that writes the map twice, then reads it twice. Mwait contains the length of time the code waits after manipulating the map, MapAddr1, and MapAddr2 contain the two addresses read and written. MwriteVal contains the value written into the map. This code is used during initial X-board checkout.

**xorHoldSim()** Toggles the value of flags.holdSim, the bit that enables the use of the hold simulator.

**xorTaskSim()** Toggles the value of flags.taskSim, the bit that enables the use of the task simulator.

**xortaskCirc()** Toggles the value of flags.taskCirculation, the bit that enables task circulation in the diagnostics (causes the diagnostics to run at each task level).

**zeroMemory()** Zero the entire memory.

## APPENDIX 1: Sample Problems

pin 35 on left side

This pin should interfere with Amux select. The initial single stepping portion of the diagnostics will fail.

pin 166 on left side

The left mask of the shifter will fail.

pin 134 on left side

The right mask of the shifter will fail.

pin 146 on left side

The least significant byte of the alu will always be #0.

pin 162 on left side

The function that replaces the ALU carry with the saved carry will fail.