TEST DESCRIPTION

The first part of these tests will run with no disk present
PrefBegin Checks basic I0 and Muzzle input, and halts on an intentional parity err
Begin start of main diagnostics
CheckState checks that all known muffle bits are in their correct state
TagReg Checks loading of the TAG register - static mode
Ram Tests the format Ram - both addressing and contents
Fifo Tests writing and reading of the Fifo - also test write write-up logic
FifoStep Allows to to step (in passive mode) the controller through Fifo operation
IsDiskSpinning Looks and the disk status and branches to done if not ready and on-line

The rest of the tests require a disk to be present and running
Tag Write Tag register, block, and wait for appropriate wake-up
WakeUps checks basic sector and index wake-ups
SectorCounters Loads Sub-sector counter and checks for the right number of sectors

NOTE: In all testing, the bits that were found to be in error are left in register "Errors"
As an example if Errors contain 200 and the test is for controller state KSTATE, then the signal on th
** the muffle KSTATE "bit 200" (ie EnableRun) would have an unexpected value.

**** - - - - - - - - - - - - - - - - - - - - - - - emulatb code for CheckTAG test
*if the disk task is ever blocked, then this loop will run, and if EMUCount goes to zero then the DSK t
**ask will be woken up via Notify. It is up to the disk micro-codeto notice that there was no actual h
**ardware wake-up.

    set[xtask, 0]; top Level; KnowRBas[Errors];
    set[EMUlocx, 201]; mc[EMUloc, EMUlocx]; *EMU task PC counter

EMULoop:
    RBase < RBase[EMUcount];
    Pd* EMUcount, at[EMUlocx]; /*Test for already zero
    SkpIf[alw0];
    Branch[EMUloop];
    EMUcount + (EMUcount)-1; /*EMU count down loop
    SkpIf[alw0];
    Wakeup[DSK]; /*Notify if just gone to Zero
    Branch[EMUloop];
    set[xtask, IP[DSK]];
** check out the muffler reading scheme and compare the results
*** read by the microcode with what Midas reads

PreBegin:
  T+DiskControl;  *At this point set TIOA with Bmux to disk TIOA
  TIOAT:
  T + noRunEnable;

Out:  Output + T;
  RBase-RBase[Errors];
  Noop;
  T+10C;
  TIOA[DiskMuff];
  Output + T;
  T - TIOASkip;  *Lets check that TIOA got set correctly
  T + T and (177400C);  *mask TIOA
  Errors + T xor (DiskMuff);
  SkpIf[alv=0];

TIOAErr0:
  Error;  *Value in TIOA not equal to DiskMuff (011C)

InNoPck:  Errors + InputNoPe;
  Noop;  
  Noop;
  Noop;

InPck:  Pd + Input;
  Noop;
  Noop;
  Noop;

CheckMuffs:
  call[read2DMuffs], T+ StateMuffs;
  Dmux100<- T;
  call[read2DMuffs], T+ StatusMuffs;
  Dmux101<- T;
  call[read2DMuffs], T+ RAMMuffs;
  Dmux102<- T;
  call[read2DMuffs], T+ TagMuffs;
  Dmux103<- T;
  call[read2DMuffs], T+ FifoMuffs;
  Dmux104<- T;
  T+177400C;  *At this point set TIOA with Bmux 377
  TIOA=T;

PnCk:  Errors - Input;  *There is no input for this TIOA address.
  Noop;  *This should therefore cause an IOin pari err
  Good,PE:01:  Branch[Begin];  *Compare Disk Mufflers against Midas Mufflers
  PCKFailed;
  Error;  *Processor Input Parity test failed!!!
  *No device was selected to drive IOB
**** Check all signals on mufflers and see that they are reasonable at this point
** This means masking off all signals that we can't be sure of.

Begin:

    Nboot;

Check.State:
call[clear-disk];
call[read2Muffs], T= StatusMuffs;
    T = T XOR (70C);  *Normalise the bits read
    Errors=1 AND (76377C);  *mask of just the bits that are determinate
    SkpIf[alu=0];

State.Errs:
    Error;  *Compare "1" bit(s) with muffler word KSTATE

call[read2Muffs], T= StatusMuffs;
    T = T XOR (18000C);  *Normalise the bits read
    Errors=1 and (177737C);  *mask of just the bits that are determinate
    SkpIf[alu=0];

Stat.Errs:
    Error;  *Compare "1" bit(s) with muffler word KSTAT

call[read2Muffs], T= FifoMuffs;
    Errors = T xor (2000C);  *Normalise the bits read
    SkpIf[alu=0];

Fifo.Errs:
    Error;  *Compare "1" bit(s) with muffler word KFIFO
*** Write the TAG register with various data.  
** I must avoid values that make the disk to barf 
** read value via the mufflers and compare 
** the value written into the TAG register is immediately 
** available for reading on the mufflers 
** run as DiskTASK only - no blocks/wakeups

TagReg:
    call[clear-disk];  
    Count+ (100000C);  
*initialize counter and TAG value

TagRegL:
    Loop;
    TIOA[DiskTag];
    Output + Count;
    call[read20Mufffs], T+ TAGmuffs;
    T^ T xor (Count);
    SkipIf[alu=0];  *skip if ok

TagRegErr:
*Tag Register not equal to data written

Pd + (Count) and (100000C);
SkipIf[alu=0];
Count+ (Count) rsh 1, Skip[];
Count+ (Count) = 1;
LoopUntil[alu=0; TagRegL];  *skip if around to zero again
**** - - - - - - - - - - - -

* Issue DriveSelect tag instructions to select drives values of 0 to 17
* and see that the controller selects the appropriate drives
* (DrSel is a subroutine that issues a drive select from value in "T"
* and returns with select.0, select.1 in bits 14, 16 of T)

DriveSelect:
  call[clear-disk];  /* initialize controller & clear any wakeups
  call[init-ram];  /* turn on RunEnable
  call[read1Mbuff], T= RunEnableMuff;  /* now check to see if RunEnable is on
  SkpIf[alu=0];  /* skip if RunEnable set

DriveSelE1:
  Error;  /* RunEnable should be set at this point

  Noop;
  call[Set-CKDrive], Count= A0;  /* Select drive 0
  SkpIf[alu=0];

  DriveSelE0:
  Error;  /* didn't select 0

  call[Set-CKDrive], Count= (Count)+1;  /* Select drive 1
  Pd= (T) xor (Count);
  SkpIf[alu=0],Errors=T;

  DriveSelE1:
  Error;  /* didn't select 1

  call[Set-CKDrive], Count= (Count)+1;  /* Select drive 2
  Pd= T xor (Count);
  SkpIf[alu=0],Errors=T;

  DriveSelE2:
  Error;  /* didn't select 2

  Noop;

dS3:
  Noop;
  call[Sel-CKDrive], Count= (Count)+1;  /* Select drive 3 through 16b
  Pd= T xor (3C);
  SkpIf[alu=0],Errors=T;

  DriveSelE3:
  Error;  /* didn't select 3

  Pd= (Count) xor (17C);  /* Do this until we get to drive 16
  LoopUntil[alu=0, dS3];

  Noop;
  call[Set-CKDrive], Count= (Count)+1;  /* Select drive 17b
  SkpIf[alu=0],Errors=T;

  DriveSelE17:
  Error;  /* didn't select 0
***** - - - - - - - - - - - - -
**check that the format ram is being written correctly by writing a random pattern
***write the Ram once; then check those values and write in new ones
Ram:
call[clear-disk]:  *initialize controller & clear any wakeups
Call[SendCommand],[T]*AO:  *make sure write addr is 0 for format RAM
KScr= 132C;  *start reading with some value
KScr2= 123C;  *start writing with some value
Count= 20C;  *counter
RamInitL:  *Write the Ram once to get known data in it
Loop:
  Call[read1Muff], T= RunEnableMuff;  *Check that RunEnable has not been set yet
  SkpIf[alu=0];
RamErr0:
  Error:  *RunEnable should still be cleared at this point
  *The format ram has been written "Count" times
  TIOA[DiskRam];
  Call[MakeRandom], T= KScr2;  *Get a new random number for next write
  KScr2= Output+ I;  *Write new number
  Call[read1Muff], T= IOBP ErrMuff;  *Check that Output didn't cause a parity error
  SkpIf[alu=0];
RamErr1:
  Error:  *Runenable should be set at this point
  *RunEnable should still be cleared at this point
  Count= (Count) - 1;
  LoopUntil[alu>0, RamInitL];
  call[read1Muff], T= RunEnableMuff;
  SkpIf[alu=0];
RamLoop:
  Error:  *Controller detected IOB parity error
  *due to the previous write
  Count= (Count) + 1;
  LoopUntil[alu>0, RamLoop];
"First initialize the controller into an active Write command and check the mufflers
"If the controller doesn't run this test right, you might try the StepFifo routine

**Fifo:**

```c
Noop;
KScr1= 123C;  *Initialize read value
KScr2= 123C;  *Initialize write value
call[clear-disk];
TIOA[DiskTag], T < T-T;  *Make sure Tag register is 0
Output+ T;
call[init-ram];
Call[SetDrive], T=37C;  *Deselect drive so command is not executed
T= 100C;  *Write header command
call[SendCommand], T+ T + (1000C);  *set Debug mode so controller becomes Active
Call[read20Muffs], T= StateMuffs;
T xor (364C);  *Normalize the bits read
Errors* T and (374C);  *mask of just the bits that are determinate
SkpIf[alu=0];
```

**FifoErr0:**

```
Error;  *Controller in wrong state after Write command
*Compare "1" bit(s) with muffler KSTATE
```

**FifoOut:**

```
Noop;
call[read20Muffs], T= FifoMuffs;  *First see that R/W address pointers are zero
Errors* T and (377C);
SkpIf[alu=0];
```

**FifoOutE0:**

```
*The Fifo address counters are wrong
Error;  *Compare should be zero
```

```
Call[MakeRandom], T= KScr2;  *Get a new random number for next write
TIOA[DiskData];
KScr2= Output+ T;  *Write new number [KFIFO = 6000]
call[read1Muff], T= IOBErrMuff;  *Check that Output didn't cause a parity error
SkpIf[alu=0];
```

**FifoOutE1:**

```
Error;  *Controller detected IOB parity error
*due to the previous write
```

```
call[read20Muffs], T= FifoMuffs;  *Check R/W address pointers again
T-T and (6777C);  *CntDone is not determinate
T xor (5000C);
Errors* T xor (21C);
SkpIf[alu=0];
```

**FifoOutE2:**

```
*The Fifo state is wrong KFIFO should be 7021
Error;  *The Fifo address counters should be 21
*OutRegFull should be 1 (2000 bit)
```

```
KScr3= 21C;  *Initialize Read/write address pointer
```

```
****Now start the main loop
FifoLoop:
Count + 16C;
```

****first do 14 writes checking parity, fifo address pointers, and wake-up

**FifoWrite0:**
- Get a new random number for next write
- Output and save the new number
- Check that Output didn't cause a parity error
  - SkpIf[alu=0];
- Error; *Controller detected IOB parity error
  - due to the previous write
- Now update the R/W address register
- Now see that R/W address counters are correct
  - T=(KScr3)+(20C);  
  - T=(KScr3)+(377C);  
- SkpIf[alu=0];

**FifoWrite1:**  *The Fifo address counters are wrong*
- Compare KFIO (bits 0..16) with KScr3
  - T=RSH[KScr3,4];  
  - Errors = T;  
  - Now compute the difference between
  - Words in Fifo = (Waddr-Raddr)&17b
  - T=(KScr3) and (17C);  
  - T=(Errors) - T;  
  - Errors = T and (17C);  
- SkpIf[alu=0];
  - T=(Counts);  
  - Read state of Write Task Wakeup
  - SkpIf[alu=0];  
  - Save value in Errors
  - SkpIf[alu=0];  
  - and skip if it should be
  - Branch[FifoErrCheck];  
  - Branch[FifoWritecont];

**FifoErrCheck:**
- (Count) xor (3C);
- SkpIf[alu=0];

**FifoWrite2:**  *Write wake-up Error*
- Error; *Value read is in Errors

**FifoWritecont:**
- Count=(Count) - 1;
- LoopUntil[alu=0, FifoWLoop];
- Noop;
- Count += 16C;
****Now do 14 reads checking data and fifo address pointers

FifoRLoop:
  Noop;
  Call[MakeRandom], T=KScr;  *Update random number for reading
  KScr+ T;
  TIOA[DiskData];
  T= Input;
  Errors= (KScr) xor T;
  SkipIf[alu=0];

FifoReadEO:
  Error;  *bad value from Fifo (KScr = good value)
  T + (KScr3) and (17C);  *update the Read portion of R/W addr register
  Pd + T xor (17C);
  SkipUnless[alu=0], KScr3 + (KScr3) + 1;
  KScr3 = (KScr3) - (2DC);
  Noop;
  Call[read2OMuffs], T= FifoMuffs;  *Now see that R/W address counters are correct
  T=T and (377C);
  Errors + T xor (KScr3);
  SkipIf[alu=0];

FifoReadEl:
  Error;  *Compare "1" bit(s) with muffler word KFIFO
  Pd= (KScr) xor (123C);
  SkipUnless[alu=0];  *exit test if we get our original random number
  Branch[FifoDone];
  Count+ (Count) - 1;
  LoopUntil[alu=0,FifoRLoop];  *Keep reading some words
  Branch[FifoLoop];  *Go back and write some more words

FifoDone:
  Call[clear-disk];  *Clear out Debug Mode
  Call[clear-disk];  *Turn off Active
  Branch[IsDiskSpinning];
****  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
**Set the controller up so that the midas STEP command can be used to observe
**the state of the controller as a word is loaded into the controller and progresses
**to the output register

FifoStep:
  Noop;
  call[clear-disk];
  TIOA[DiskTag], T + T-T;
  Output+ T;
  call[init-ram]; *Turn on RunEnable so Fifo can be loaded
  *Fifo is initialized and empty [KFIFO = 2000]
  T + IC;

Set-Pasive:
  *Midas must be in Passive mode for this test
  TIOA[DiskData], Breakpoint; *TIOA = 12
  Noop; *InRegister is loaded [KFIFO = 2000]

S-out: Output+ T; *Write data (see ALUA) [KFIFO = 2000]
S-IR: Noop; *InRegister is loaded [KFIFO = 2400]
S-Fifo: Noop; *Fifo + InRegister
  Noop; *FifoEmpty+0 (no muff) [KFIFO = 2020]
S-OR: Noop; *OutRegister + Fifo
S-In: KScr+Input; *I0B+OutRegister
S-Ck: B + KScr, Breakpoint; *Read data (see ALUA) [KFIFO = 2021]
  T + T lcy 1, Branch[S-out]; *Change data and do again

On successive passes through this loop, KFIFO is incremented by 21 as the two 4 bit address counters incr
**ment modulo 16. The data written and read is 1 left shifted 1 for each pass, and is on the "BMux"
**so you can see in passive mode on the ALUA muffer.
The valid values for instruction "S-Ck" are:
  KFIFO= 3021 3042 3063 3104 3125 3146 3167 3210 3231 3252 3273 3314 3335 3356 3377 3000
  ALUA= 0001 0002 0004 0010 0019 0020 0040 0100 0200 0400 1000 2000 4000 10000 20000 40000 100000

****  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
**Now check things that require a disk to be up and spinning

IsDiskSpinning:
  Call[clear-disk]; *Clear out previous states
  call[init-ram]; *Turn on RunEnable
  call[read20Muffs], T+ StatusMuffs;
  T + T and (160000);
  Skip[alu=0];
  Branch[Done]; *Done if the disk is not up and ready
***** - - - - - - - - - - - - - - - - - -
** write the TAG register with incrementing data, thru all values
** read value via the mufflers and compare
** start as DiskTASK then block and wait for Hardware Wakeup
** The emulator has a timeout loop which will generate a Wakeup to be detected as an error
** this checks out clearWs, block, and TAG wakeups

Tag:  
call[clear-disk];  
CountS+5C;  
*read the 5 Tmuffs  
call[ReadMuffs], T- IndexTmuff;  
*address of 1st muffler to read  
SkipIf[alu=0];  
*skip if wakeups cleared  

TagE0:  
Error;  
*wakeups not cleared  
call[init-ran];  
*turn on RunEnable  
call[read1Muff], T= RunEnableMuff;  
*Now check to see if RunEnable is on  
SkipIf[alu#0];  
*skip if RunEnable set  

TagE1:  
Error;  
*RunEnable should be set at this point  
Count+ (IC);  
*shift 1 bit - primarily tests wake-ups  

TagLoop:  
Noop;  
Pu+ (Count) and (70000C);  
*See if the new number should enable Wake-Up  
Branch[NoWkUp,alu=0];  

Noop;  
*Micro placement  
call[SendTag], T- Count;  
*returns via Wakeup  
SkipIf[alu#0],Errors=T;  
*Test to see that return was not due to time out  

TagWkUpEO:  
Error;  
*Was woken up by EMU timeout  
Branch[TagCk];  

NoWkUp:  
call[SendTag], T- Count;  
*returns via time-out  
SkipIf[alu=0],Errors=F;  
*Test to see that return was not due to time out  

TagWkUpEI:  
Error;  
*Was woken up by Hardware  

TagCk:  
T= TAGmuffs;  
*Micro placement  
call[read2OMuffs];  
*Now see if the data is correct  
Errors= T xor (Count);  
SkipIf[alu=0];  
*skip if ok  

TagE2:  
Error;  
*Wrote "Count" into TAG register  
Count+ (Count) lsh 1;  
LoopUntil[alu=0, TagLoop];  
*test for done
********* - - - - - - - - - - - - - - - - - - -
**Try the basic Sector and Index wake-up logic
WakeUps:
call[clear-disk];
call[init-ram]; *turn on RunEnable
EMUcount=A1;

WakeIdx:
Noop;
Call[DoClear], T+ clearSTW;
Block; *block for wake-up -- timeout @ 17ms

CountS+5C;
Call[ReadMuffs], T+ IndexTWmuff;       *See what woke up the task
Pd + T and (SectorTWBit);
LoopUntil[alu=0,WakeIdx], Errors+T;    *Loop until not a Sector wake-up
Pd + T xor (IndexTWBit);
SkpIf[alu=0];
WakeIdxErr:
Error; *No INDEX wake-ups???
EMUcount + A1;

WakeSec:
Noop;
Call[DoClear], T+ clearITW;
Block; *block for wake-up -- timeout after 17ms

CountS+5C;
Call[ReadMuffs], T+ IndexTWmuff;       *See what woke up the task
Pd + T and (IndexTWBit);
LoopUntil[alu=0,WakeSec], Errors+T;    *Loop until not an Index wake-up
Noop;
Pd + T xor (SectorTWBit);
SkpIf[alu=0];
WakeSecErr:
Error; *No SECTOR wake-ups???
**Now put all reasonable values of SubSector counts and check for the right number of Sectors**

SectorCounters:
call[clear-disk];
call[initialize]; *turn on RunEnable

Sectors117:
call[set-sector], T + AO;
call[do-clear], T+ clearATWs;
call[send-command], T+readyAndIndex;
EMUcount + A1,Block; *block for wake-up -- timeout after 17ms

Counts+5C;
call[read-muffs], T+ IndexTWmuff; *See what woke up the task
Pd + T and (IndexTWBit);
loopUntil[alu#0,117loop], Errors + (Errors)+1;
Pd + (Errors) xor (117D);
	if[alu=0];
SectorErr117: "With a value of 0 in the Sub-Sector counters, 
Error; "there should have been a count of 117d sectors.

Sectors30:
call[set-sector], T + AltoCount;
call[do-clear], T+ clearATWs;
call[send-command], T+readyAndIndex;
EMUcount + (1000C),Block; *block for wake-up -- timeout after 150us

Counts+5C;
call[read-muffs], T+ IndexTWmuff; *See what woke up the task
Pd + T and (IndexTWBit);
loopUntil[alu#0,30loop], Errors + (Errors)+1;
Pd + (Errors) xor (30D);
	if[alu=0];
SectorErr30: "With a value of 3 in the Sub-Sector counters, 
Error; "there should have been a count of 30d sectors.

Sectors:9:
call[set-sector], T + TrsCount;
call[do-clear], T+ clearATWs;
call[send-command], T+readyAndIndex;
EMUcount + A1,Block; *block for wake-up -- timeout after 17ms

Counts+5C;
call[read-muffs], T+ IndexTWmuff; *See what woke up the task
Pd + T and (IndexTWBit);
loopUntil[alu#0,9loop], Errors + (Errors)+1;
Pd + (Errors) xor (9D);
	if[alu=0];
SectorErr9: "Wake-Up was NOT an Index 
Error; "WaitRillIndex command probably failed
Errors <- 0;
9Loop:
Call[DoClear], T <- clearATWt;
EMUcount <- 20000C, Block;  *block for wake-up -- timeout after 2000us
CountS=5C;
Call[ReadMuffs], T <- IndexTWmuff;  *See what woke up the task
Pd <- T and (IndexTWbit);
LoopUntil[alu#0,9Loop], Errors <- (Errors)+1;
Noop;
Pd <- (Errors) xor (9D);
SkipIf[alu=0];
SectorErrS:  *With a value of 0 in the Sub-Sector counters,
Error;  *there should have been a count of 9d sectors.

********* - - - - - - - - - - - - - - - - - - - - - -
call[clear-disk];
branch[Done],  *Done with diagnostics
**** subroutine; ****
readMuff:  *muffler number is in T
    TIOA[DiskMuff], Global;
    Output+ T;
    noop; T+ Input;
    Pd+ T, Return; *returns with alu test hanging
**** subroutine; ****
read20Muffs:  *muffler number is in T
    Counts+ 20, Global; *Set count to 20
    Branch[foo], KScrS+ AO; *clear out starting value
**** subroutine; ****
ReadMuffs:  *T plus count in Counts
    KScrS+ AO, Global; *clear out starting value
fbo:  TIOA[DiskMuff], KScr4+T;
loopRdMf:
    Output+ KScr4, KScr4= (KScr4) + 1;
    Counts (Counts) - 1;
    branch[+.3, alu<0], KScrS+ (KScrS) icy 1;
    T+ input;
    KScrS+ (KScrS) OR T, branch[loopRdMf];
    KScrS+ T+ (KScrS) icy 1, return; *Values are returned in KScrS and T
**** subroutine; ****
DoClear:  *Bits to be cleared are in T
    *Noop, Global;
    TIOA[DiskMuff], Global;
    Output+ T; Noop; Noop; *Noops required for wake-ups to clear
    return;
**** subroutine; ****
*** returns when a seek/tagTW is actually seen - or when the EMU task times out
*** An actual hardware wakeup will cause the subroutine to return -1
*** An EMU timeout will return 0
SendTag:
    *TAG value is in T
    KScrL+ Link; ****
    TIOA[DiskTag];
    Output+ T;
    call[readMuff], T+ 10BPerrMuff; *Check that Output didn't cause a parity error
    SkpIf[alu=0];
    OutPerr:
        Link+KScrL, branch[.], breakpoint; *Controller detected IOB parity error
    Noop;
    SndTagL:
        Call[DoClear], T= clearALLbutTagTW;
        EMUcount + 40C, Block; *block for wake-up --3us maximum
        Counts+5C; *block for wake-up
        Call[ReadMuffs], T= IndexTWmuff; *See what woke up the task
        SkpUnless[alu=0], KScrS+T; *Loop back and wait for another wakeup
        Branch[TagTWseen]; *Loop back and wait for another wakeup
    Noop;
    TagTWseen:
        Call[DoClear], T= clearALL;
    subroutine; ****
        Link+ KScrL;
        EMUcount + AO; *stop the Emulator task from counting
        T = KScrS, Return;
**** subroutine; ****
SetDrive:
    TIOA[DiskTag], Global;
    Output+ T; *First set the value in the 12 bit register
    T+ T OR (DrvSelect1AG); *Drive select plus DriveTag bit
    Output+ T; *Now turn on the strobe
T * T and (7777C); *drive select only
Output+ T, Return; *Finally turn off the strobe

SetSector:
KScrL + Link, Global;
top level; ****
Call[SetDrive];
T * T or (SectorCntEnable);
Call[SetDrive];

subroutine; ****
Link+ KScrL;
Return;

SendCommand; *command is in T
TIOA[DiskControl], Global;
Output+ T, return;

clear-disk:
RBase-RBase[Errors], Global; *Not needed
KScrL + Link;
top level; ****
T = EMUlLoc; *set the EMU PC to its loop
Link+ T;
LDTPC< EMU;
EMUlCount=AO; *Clear EMU counter
T= DiskControl; *Set TIOA with BMux to a disk TIOA
TIOA= T;
T= NoRunEnable; *Turn of RunEnable
subroutine; ****
Link+ KScrL;
Output+ T, Retum;

init-ram: *Init format Ram in Diablo format
TIOA[DiskRam];
T = IC; *header count - 1
Output+ T;
T = JC; *label count - 1
Output+ T;
T = 377C; *data count - 1
Output+ T, T - T;
Output+ T; *unused block = 0
T = 104C; *look for DI Trident sync pattern/read command
Output+ T;
T = 204C; *write command
Output+ T;
T = AC; *init command
Output+ T, T - T;
Output+ T; *reset command is 0
T = 33C; *write delay first block
Output+ T;
T = 6C; *write delay successive blocks
Output+ T;
T = 1IC; *read delay first block
Output+ T;
T = 2C; *read delay successive blocks
Output+ T;
Output+ T, T - T - 1; *head select delay is also 2
Output+ T, T - T - 1; *no. of ECC words = 1 = 1
Output+ T; *the constant 1 - 1 = 0
Output+ T; *must write last word to turn on RunEnable
TIOA[DiskControl]; *no wakeups 'til Ready&Index
T = Ready&Index;
Output+ T;
TIOA[DiskMuff];
T = clearALL; *clearErrors + clearTWs and returns
Output+ T, Return;

*And now my own favorite random number generator
**You give it some number, and it returns a new number
MakeRandom:  *Old value is in T
Pd + T, Global;
Branch[x. +3,alu<0], T + T LSH 1;  *Skip to return
T + T xor (77000C);
T + T xor (273C);   *XOR by value 77213
Return;  *and Return

Set-CKDrive:
KScrL + Link, Global;
top level;  ****
Call[SetDrive] T+Count;
Call[read1Muff], T+DiskSelectMuff;  *muffler number is in T
KScrS=T lsh 1;  *Read the two DriveSelect mufflers
Call[read1Muff], T+DiskSelectMuff+1;  *muffler number is in T
subroutine;
Link~ KScrL;  ****

T+ (KScrS) or T, Return;
top level;  ****
** Disk-Defs.mc - last modified December 10, 1978 6:17 PM
** stored in [ivy]<Dis source>Emu.dm

TITLE[Tricond-Defs.mc - January 9, 1979 3:50 PM];
* m[noop, inc[branch[+1]]];  *Defined in Preamble.mc
m[skipif, BNGO[tsd] BAT[.+2,#1,#2]];

***** - - - - - - - - - - - - - - - - TASK[N(DSK, 14);
*** TIOA values for disk

Device[DiskControl, 10];
Device[DiskMuff, 11];
Device[DiskData, 12];
Device[DiskRam, 13];
Device[DiskTag, 14];

***** - - - - - - - - - - - - - - - - *** Rm register names

RmRegion[disk rms];
rv[Errors, 0];
rv[KScr, 0];
rv[KScr2, 0];
rv[KScr3, 0];
rv[KScr4, 0];
rv[Count, 0];
rv[CountS, 0];
rv[KScr5, 0];
rv[KScr6, 0];
rv[EMUcount, 0];
rv[Dmux100, 0];
rv[Dmux101, 0];
rv[Dmux102, 0];
rv[Dmux103, 0];
rv[Dmux104, 0];

***** - - - - - - - - - - - - - - - - *** Muffler constants

mc[StatusMuffs, 20];
cm[NotReadyBit, StatusMuffs, 5];
cm[writeErrorMuff, StatusMuffs, 16];
cm[readErrorMuff, StatusMuffs, 17];
cm[StateMuffs, 0];
cm[IndexTWWmuff, StateMuffs, 1];
cm[SectorTWWmuff, StateMuffs, 2];
cm[SeekTWWmuff, StateMuffs, 3];
cm[TagTWWmuff, StateMuffs, 3];
cm[ReadTWWmuff, StateMuffs, 4];
cm[WriteTWWmuff, StateMuffs, 5];
cm[NreableMuff, StateMuffs, 10];
cm[DiskSelectMuff, StateMuffs, 16];
cm[DiskSelectMuff+, DiskSelectMuff, 1];
cm[IOBPErrMuff, StateMuffs, 34];
cm[IndexTWBit, 29];
cm[SectorTWBit, 10];
cm[OtherTWBit, 4];
cm[ReadTWBit, 2];
cm[WriteTWBit, 1];
cm[117D, 165];
cm[30D, 36];
cm[3D, 11];
cm[RamMuffs, 40];
cm[TagMuffs, 60];
cm[FifoMuffs, 100];
cm[IW#addrMuffs, 110];
cm[T#AddrMuffs, 114];

***** - - - - - - - - - - - - - - - - *** Disk Command constants

mc[clearErrors, 400];  *= muf fAddr.07
mc[clearOTWs, 1000];  *= muf fAddr.05 - clea rOtherTWs
mc[clearSTW, 2000];  *= muf fAddr.05 - clearSectorTW
mc[clearITW, 4000];  * = maffAddr.04 - clearIndexTW
mc[clearATWs, 7000];  * = maffAddr.04, .05, .06
mc[clearALL, 7400];  * = maffAddr.04, .05, .06, .07
mc[clearAllbutTagTW, 6400];  * = maffAddr.04, .05, .06, .07
mc[forceCompError, 10000];  * = maffAddr.03
mc[clearSetCompError, 20000];  * = maffAddr.02
mc[NoRunEnable, 2000];  * = Control.05
mc[DebugMode, 1000];  * = Control.06
mc[ReadyAndIndex, 400];  * = Control.07
mc[DrvSelectTAG, 100000];  * = TAG.00
mc[CylSelectTAG, 40000];  * = TAG.01
mc[HeadSelectTAG, 20000];  * = TAG.02
mc[Control1Tag, 10000];  * = TAG.03
mc[SectorCntEnable, 40];  * = TAG.04
mc[ATSyncPat, 1];  * disk sync pattern to write in Alto mode
mc[DSyncPat, 201];  * disk sync pattern to write in D1 mode
mc[ATSyncRead, 4000];  * Alto sync pattern to be read
mc[D1SyncRead, 0];  * D1 sync pattern to be read
mc[NormalRead, 104];  * control RAM entry for read command

***** - - - - - - - - - - - - - - - - - - - - - - - - - - - -
* alto sub-sector count = 1
MC[AltoCount, lshift[3,6]];  * driveSelectTAG + (Tri sub-sector count - 1)
mc[TriCount, 100000, lshift[14,6]];