```
TITLE[POSTAMBLE];
TOP LEVEL;
%
```

May 18, 1981 1:37 PM
Fix random number generator to use a better algorithm. Modify restart code, and the various subroutines associated with random numbers. Add setRandV, cycleRandV.

February 1, 1980 6:24 PM
Fix goto[preBegin], described below, into goto[restartDiagnostic]. Postamble already defines and uses preBegin.

February 1, 1980 11:52 AM
Fix restart to goto[preBegin]. This allows each diagnostic to perform whatever initialization it wants.

September 19, 1979 9:18 PM
Fix another editing bug in chkSimulating, used the wrong bit to check for flags.conditionOK -- just did it wrong.

September 19, 1979 9:08 PM
Fix bug in chkSimulating wherein an edit lost a carriage return and a statment became part of a comment. Unfortunately, automatic line breaks made the statement look as if it were still there rather than making it look like part of the comment line.

September 19, 1979 4:23 PM
Fix placement errors associated with bumming locations from makeholdvalue and from checksimulating.

September 19, 1979 3:48 PM
Bum locations to fit postamble with current os/microD: reallyDone, checkFlags global, make checkFlags callers exploit FF, eliminate noCirculate label, make others shorter..

September 19, 1979 10:41 AM
change callers of getIM*, putIM* to use FF field when calling them.

September 19, 1979 10:18 AM
Create zeroHoldTRscr which loops to zero hold-- called by routines that invoke resetHold when the hold simulator may be functioning. Make getIM*, putIM* routines global.

September 16, 1979 1:27 PM

Bum code to fix storage full problem that occurs because OS 16/6 is bigger than OS 15/5: remove kernel specific patch locations (patch*).

August 1, 1979 3:28 PM
Add scopeTrigger.

June 17, 1979 4:48 PM
Move IM data locations around to accommodate Ifu entry points

April 26, 1979 11:03 AM
Make justReturn global.

April 19, 1979 5:03 PM
Remove calls to incTask/HoldFreq from enable/disableConditionalTask.

April 18, 1979 3:24 PM
Remove DisplayOff from postamble.

April 18, 1979 11:11 AM
Rename chkTaskSim, chkHoldSim, simControl to incTaskFreq, incHoldFreq, makeHoldValue; clean up setHold.

April 17, 1979 10:51 PM
SimControl now masks holdFreq and taskFreq & shifts them w/ constants defined in Postamble.

April 11, 1979 3:49 PM
Add breakpoint to "done", and fix, again, a bug associated with task simulation. Set defaultFlagsP (when postamble defines it) to force taskSim and holdSim.

March 7, 1979 11:42 PM
Set RBASE to defaultRegion upon entry to postamble. thnx to Roger.

February 16, 1979 2:54 PM
Modify routines that read IM to invert the value returned in link if b1 from that value =1 (this implies the whole value was inverted).

January 25, 1979 10:41 AM
Change **taskCirculate** code to accommodate taskSim wakeups for task 10D, 12B

January 18, 1979 5:13 PM
Modify **checkTaskNum** to use the RM value, currentTaskNum, and modify **taskCircInc** to keep the copy in currentTaskNum.

January 15, 1979 1:25 PM
add **justReturn**, a subroutine that just returns
January 9, 1979 12:07 PM
breakpoint on xorTaskSimXit to avoid midas bug
%

%*+++

TABLE of CONTENTS, by order of Occurence

%*+++
IM[ILC,0];
%*+++

This code presumes R0=0 and uses RSCR, RSCR2, T, and Q. It uses a number of other registers in a different RM region.

When Postamble gets control of the processor at "Done", bits in "Flags", a word in IM determine which of Postamble’s functions will occur when it runs. At the least, Postamble inrements at 32 bit number in IM called Iterations. If flags.taskSim is true, the task simulator started. The task simulator awakens after a software controllable number of clocks has occured. The microcode that wakes up must reset the task simulator before it (the microcode) blocks to cause a task wakeup to occur again. The first time a program runs (ie., the time before it gives control to "done") the task simulator and the hold simulator (discussed below) are inactive. Running the task simulator forces task specific hardware functions to effect the state of the machine.

When flags.holdSim is set, Postamble sets the hold simulator to a non-zero value. The 8 bit "hold value" enters a circulating shift register where occurence of a "1" bit at b[0] causes an external hold. This exercises the hold hardware.

The body of postamble contains a number of procedures for user programs, including routines to read and write IM, a routine to return a random number, and routines to initialize a task’s pc and to notify it.

```
%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
done:
```

```
* June 17, 1979  4:49 PM                        POSTAMBLE CONSTANTS

     set[randomTloc, 620];                      * random number generator may have to
* be moved to "global" call location if extensively used!


     set[flagsLoc, 1000];                       mc[flagsLocC, flagsLoc];
     set[taskFreqLoc, 1400];                    mc[taskFreqLocC, taskFreqLoc];
     set[holdFreqLoc, 2000];                    mc[holdFreqLocC, holdFreqLoc];
     set[nextTaskLoc, 2400];                    mc[nextTaskLocC, nextTaskLoc];
     set[itrsLoc, 3000];                        mc[itrsLocC, itrsLoc];
*    holdValueLoc defined in preamble!
     set[preBeginLoc, 4000];                    mc[preBeginLocC, preBeginLoc];
     set[initTloc, 4400];                       mc[initTlocC, initTloc];

     ifdef[simInitLocC,,mc[simInitLocC, initTloc] ];   * define the bmux constant for the
*    address of the task simulator code. If its already been defined, leave it as is.

*    flags.taskSim defined in preamble!
*    flags.holdSim defined in preamble!
*    flags.simulating defined in preamble!
     mc[flags.testTasks, b13];                  * than 8 flags (since READIM rtns a BYTE)
     mc[flags.conditional, flags.conditionalP];    * allow simulating iff flags.simulating
                                                * AND flags.conditionOK
     mc[flags.conditionOK, flags.conditionOKp];    * enable conditional simulating

%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

This portion of the kernel code encapsulates the microdiagnostic with an outer loop. This outer loop has several features that it implements:

task simulation
hold simulation
task switching

Task simulation refers to the taskSim register in the hardware. It is 4 bits wide; taskSim[0] enables the task simulator and taskSim[1:3] form a counter that determines the number of cycles before a task wakeup occurs.

Hold simulation is similar: holdSim is an 8-bit recirculating shift register in which the presence of a 1 in bit 7 causes HOLD two instructions later.

Task switching determines which task will run the microdiagnostics.

These features are controlled by the flags word in IM. If the appropriate bits are set to one, the associated feature will function. The bits are defined above (flags.taskSim, flags.holdSim, flags.testTasks).

```
%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


     rmRegion[rmForKernelRtn];
     knowRbase[rmForKernelRtn];

     rv[setHoldRtn, 0];
     rv[oldt, 0];                               * save t, rscr, rscr2, rtn link for resetHold
     rv[oldrscr, 0];
     rv[oldrscr2, 0];
     rv[resetHoldRtn, 0];
     rv[xorFlagsRtn,0];
     rv[flagSubrsRtn, 0];
     rv[mainPostRtn, 0];

     knowRbase[rm2ForKernelRtn];                * defined in preamble because of macros
* that reference randV, randX


     knowRbase[defaultRegion];
```

```
* February 1, 1980  6:24 PM                    POSTAMBLE CONTROL CODE

      RBASE _ rbase[defaultRegion], breakpoint;  * set RBASE incase user's is different.
      call[incTaskFreq];
      call[incHoldFreq];
      call[makeHoldValue];
      call[taskCirculate];
      call[incIterations];

      call[checkFlags]t_flags.testTasks;         * bookkeeping is done. switch tasks if required
      skpif[ALU#0];
      branch[preBegin];                          * xit if not running other tasks

taskCircInit:                                    * now that bookkeeping is done, switch tasks if
required
      noop;                                      * for placement.
      call[checkTaskNum];
      rscr _ t;                                  * rscr _ nextTask

      t _ preBeginLocC;                          * link _ t _ preBeginLoc
      subroutine;
taskCirc:
      zeroHold[rscr2];                           * turn off hold-task sim during LdTpc_, wakeup
      link _ t;
      t _ rscr;
      top level;
      ldTPC _ t;                                 * tpc[nextTask] _ preBeginLoc
      call[notifyTask];                          * wakeup nextTask: task num in t
set[xtask, 1];
      block;
set[xtask, 0];

preBegin: noop,                                  at[preBeginLoc];
      call[chkRunSimulators];                    * check for simulator conditions and run if required

reallyDone:                                      * LOOP TO BEGIN
      t_RSCR_a1;
      goto[begin], RSCR2_t;

restart:                                         * restart diagnostics from "initial" state
      rndm0 _ t-t;                               * restart random number generator
      randX _ t-t;

      rscr _ t-t;                                * restart hold/task simulator stuff
      call[putIMRH], t _ holdFreqLocC;
      rscr _ t-t;
      call[putIMRH], t_taskFreqLocC;
      rscr _ t-t;
      call[putIMRH], t _ holdValueLocC;

      rscr _ t-t;                                * restart iterations count
      call[putIMRH], t _ itrsLocC;

      branch[restartDiagnostic];                 * special entry point so each diagnostic
* can perform whatever special initialization that it wants to perform
```

```
* January 18, 1978  1:51 PM

%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

      This code sets the taskSim value with the next value if flags.testTasks is true. Otherwise 0 is
used.

IF flags.taskSim THEN
       BEGIN                                    -- when hardware counts to 17 it awakens
       taskFreq _ (taskFreq + 1) or 10b;        -- simTask
       IF taskFreq > 15 THEN taskFreq _ 12;     -- always wait min=2 cycles
       END
      ELSE taskFreq _ 0;
IF flags.holdSim THEN
       BEGIN
       holdFreq _ holdFreq+1;
       IF holdFreq >376 THEN holdFreq _ 0;
       END;

      ELSE holdFreq _ 0;

%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
incTaskFreq: subroutine;
      t _ link;
      mainPostRtn _ t;
      top level;

      call[checkFlags], t_flags.taskSim;        * see if taskSim enabled
      branch[writeTaskSim, alu=0],t_r0;         * use 0 if not enabled
      t_taskFreqLocC;                           * incrment next taskSim
      call[readByte3];
      t_(r1)+(t);
      t-(156C);                                 * Use [1..156). 156 => max wait,
      skpif[alu<0];                             * 1 = > min wait. Beware infinite hold!
      t_r1;                                     * see discussion at simInit, simSet code
      noop;

writeTaskSim:
      rscr _ t;
      call[putIMRH], t_taskFreqLocC;            * update IM location

taskSimRtn:
      goto[topLvlPostRtn];


incHoldFreq: subroutine;                        * see if holdSim enabled
      t _ link;
      mainPostRtn _ t;
      top level;

      call[checkFlags], t_flags.holdSim;
      branch[noHoldSim, alu=0],t_r0;            * use zero if hold not enabled
      t_holdFreqLocC;
      call[readByte3];
      t_t+(r1);
      t-(377c);                                 * IF holdFreq >376
      skpif[alu<0];
      t_r1;                                     * THEN holdFreq _ 1;
      noop;                                     * here for placement
noHoldSim:
      rscr _ t;                                 * rewrite IM
      call[putIMRH], t _ holdFreqLocC;

holdSimRtn:
      goto[topLvlPostRtn];
```

```
* April 17, 1979  10:51 PM
%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

This code actually controls the task and hold loading. It is responsible for initializing T for the
task at simTaskLevel, and it is responsible for initializing HOLD.

      The code proceeds by constructing the current value to be loaded into hold and placing it in IM
at holdValueLoc. Kernel loads HOLD as its last act before looping to BEGIN.

hold&tasksim_ requires hold value in left byte, task counter value in right byte.
%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
makeHoldValue: subroutine;                      * construct holdValue
      saveReturn[mainPostRtn];

      call[chkSimulating];
      skpif[alu#0];
      branch[simCtrl0];

      t_holdFreqLocC;                           * rscr2 _ holdFreq
      call[readByte3], t_holdFreqLocC;
      rscr2 _ t;

      call[readByte3], t_taskFreqLocC;          * t _ taskFreq

      t_lsh[t, sim.taskShift];                  * position hold and task values
      t_t and (sim.taskMask);
      rscr2 _ lsh[rscr2, sim.holdShift];
      rscr2 _ (rscr2) and (sim.holdMask);
      rscr2 _ (rscr2) and (377c);

      rscr2 _ (t) + (rscr2);                    * taskFreq,,holdFreq
      rscr _ rscr2;
%
 now, save combined taskSim, holdSim values in IM. Last thing done before exiting postamble is to set
HOLD if simulating.
%
simCtrlWHold:                                   * may branch here from simCtrl0
      call[putIMRH], t _ holdValueLocC;         * write holdValue into holdValueLoc
      branch[simCtrlRtn];

simCtrl0:
      branch[simCtrlWHold], rscr _ t-t;         * write zero into holdValueLoc

simCtrlRtn:
      goto[topLvlPostRtn];
```

```
* September 19, 1979  9:09 PM

%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
       IF chkSimulating[] THEN fixSimulator[];

* cause hold or task simulator to run, if required
%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


chkRunSimulators: subroutine;
      saveReturn[chkRunSimRtn];
      call[chkSimulating];
      dblBranch[chkRunsimXit, chkRunSimDoIt, alu=0];
chkRunSimDoIt:                                  * run the simulator
      noop;
      call[fixSim];
      noop;

chkRunSimXit:
      returnUsing[chkRunSimRtn];


* September 19, 1979  9:19 PM
%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
chkSimulating: PROCEDURE RETURNS[weAreSimulating: BOOLEAN] =
       BEGIN
       weAreSimulating _ FALSE;
       IF flags.Simulating THEN
              IF ~(flags.Conditional) OR (flags.Conditional AND flags.ConditionOK) THEN
                     weAreSimulating _ TRUE;
       END;

%*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
chkSimulating: subroutine;
      saveReturn[chkSimulatingRtn];
      call[checkFlags], t_flags.simulating;     * check for taskSim OR holdSim
      branch[chkSimNo, alu=0];
      t _ flags.conditional;                    * We're simulating. check
      call[checkFlags], t _ flags.conditional;
      dblbranch[chkSimYes, chkSimCond, alu=0];
chkSimCond:                                     * conditional simulation. check for ok
      t _ flags.conditionOK;
      call[checkFlags];
      skpif[alu=0];
      branch[chkSimYes];                        * conditionOK is set, do it!
      branch[chkSimNo];
chkSimYes:                                      * run the simulator
      t _ (r0)+1;                               * rtn w/ alu#0
chkSimRtn:
      returnAndBranch[chkSimulatingRtn, t];
chkSimNo:
      branch[chkSimRtn], t _ r0;                * rtn w/ alu=0
```

```
* January 25, 1979  10:44 AM
%
```

This code controls task circulation for the diagnostics: when flags.testTasks is set, postamble causes successive tasks to execute the diagnostic code when the current task has completed. If flags.taskSim is true the diagnostic is using the taskSimulator to periodically awaken the simulator task; consequently, that task (*simTaskLevel*) must not execute the diagnostic -- otherwise the advantage of the simulator for testing the effects of task switching will be lost.

```
IF ~flags.testTasks THEN RETURN;
temp _ getTaskNum[] + 1;                        -- increment the current number
IF flags.taskSim THEN
       IF temp = simTaskLevel THEN temp _ temp+1;
IF temp > maxTaskLevel THEN temp _ 0;
putTaskNum[temp];                               -- remember it in IM
%
taskCirculate: subroutine;
      saveReturn[mainPostRtn];
      call[checkFlags], t _ flags.testTasks;
      branch[taskCircRtn, ALU=0];               * Don't bother if not task circulating.
      noop;

      call[checkTaskNum];                       * Increment the current task number.
      t _ t + (r1);                             * Current value came back in t.
      q _ t;                                    * Remember incremented value in Q.

      call[checkFlags], t _ flags.taskSim;
      skpif[ALU#0], t _ q;                      * Now, see if using task simulator.
      branch[taskCircChk];                      * If not task simulating, check for max size.

      t - (simTaskLevelC);                      * Since we're task simulating, avoid
      skpif[ALU#0];                             * we must avoid simTaskLevel.
      t _ t+1;                                  * Increment over simTask if required.
      noop;

taskCircChk:
      t - (20C);                                * See if tasknum is too big.
      skpif[ALU#0];
      t _ t-t;                                  * We wraparound to zero.

      currentTaskNum _ t;                       * keep it in both RM and IM
      rscr _ t;
      call[putIMRH], t _ nextTaskLocC;
      noop;                                     * for placement

taskCircRtn:
      goto[topLvlPostRtn];
```

```
* January 18, 1978  1:57 PM

incIterations: subroutine;                  * maintain double precision count at incItrsLoc
      t _ link;
      mainPostRtn _ t;
      top level;

      call[getIMRH], t _ itrsLocC;          * increment iteration count at tableloc+1
      rscr _ (t)+1;

      rscr2 _ rscr;                         * copy new itrs
      rscr2 _ (t) #(rscr);                  * see if new b0 # old b0

      rscr2 _ (rscr2) AND (b0);
      skpif[alu#0];
      branch[incItrs2], q _ r0;             * new b0 = old b0. remember in q and write
      t and (B0);                           * see if b0 went from 0 to 1 or 1 to 0 (carry)
      skpif[alu=0], q_r0;                   * skpif old b0 = 0
      q _ r1;
incItrs2:

      call[putIMRH], t _ itrsLocC;          * T = addr, rscr = value
      rscr2 _ q;
      branch[incItrsRtn,alu=0];             * goto incItrsRtn if no carry

incItersHi16:
      link _ t;                             * read hi byte of hi 16 bits
      call[getIMLH];
      rscr _ (t)+1;
      call[putIMLH], t _ itrsLocC;          * T = addr, rscr = value
      noop;                                 * help the instruction placer.

incItrsRtn:
      goto[topLvlPostRtn];
```

```
* March 20, 1978  1:51 PM                        KERNEL - COMMON SUBROUTINES

resetHold: subroutine;                          * special subroutine called by IM manipulating
* code.  This subr saves t, rscr, rscr2 and causes hold to be initialized to the value in
* holdValueLocC. It restores the RM and T values before returning.
     oldT _ t;
     t _ link;
     resetHoldRtn _ t;
     top level;
     t _ rscr;
     oldrscr _ t;
     t _ rscr2;
     oldrscr2 _ t;                              * link, t, rscr, and rscr2 are now saved.

     t _ HoldValueLocC;                         * READ RIGHT HALF, HoldValueLocC
     subroutine;
     link _ t;
     top level;
     readim[3];                                 * read low order byte
     subroutine;
     t _ link;                                  * low byte in t
     t and (b1);                                * see if the data is inverted. If so, b1 will
     skpif[ALU=0];                              * 1, and we must reinvert the data.
     t _ not(t);
     t _ t and (getIMmask);                     * isolate the byte

     rscr _ HoldValueLocC;
     subroutine;
     link _ rscr;
     top level;
     readim[2];                                 * read hi order byte
     subroutine;
     rscr2 _ link;                              * hi byte in rscr2
     (rscr2) and (b1);                          * see if the data is inverted. If so, b1 will
     skpif[ALU=0];                              * 1, and we must reinvert the data.
     rscr2 _ not(rscr2);
     rscr2 _ (rscr2) and (getIMmask);           * isolate the byte

     top level;
     noop;
     rscr2 _ lsh[rscr2, 10];                    * left shift hi byte
     t _ t and (377C);                          * isolate low byte
     t _ t OR (rscr2);                          * add hi byte
     call[setHold];

     knowRbase[rmForKernelRtn];                 * restore link, t, rscr, rscr2, then return
     RBASE _ rbase[rmForKernelRtn];
     t _ oldrscr;
     rscr _ t;
     t _ oldrscr2;
     rscr2 _ t;
     subroutine;
     link _ resetHoldRtn;
     return, t _ oldt, RBASE _ rbase[defaultRegion];
```

```
* June 23, 1978  10:22 AM
setHold: subroutine;                          * ENTER w/ T = HOLD value
* clobber t, rscr, rscr2

      zeroHold[rscr2];                        * kill hold-task sim before polyphas instrs xqt
      rscr2 _ q;                              * SAVE  Q
      q _ t;                                  * save hold value, then save rtn link
      t _ link;
      setHoldRtn _ t;

      taskingon;
      t _ simInitLocC;                        * defined w/ postamble constants OR in
      * some user specific code (eg., memSubrsA where RM values are defined). This
      * convention allows users to specify their own code to run when the simulator task runs.
      link _ t;                               * cause task taskSimLevel to put
      top level;
      ldTPC _ simTaskLevelC;                  * proper hold value in T for refresh
      notify[simTaskLevel];                   * after task switch occurs. Remember
                                              * taskSim is a counter. refresh it!
      noop;                                   * wakeup will happen soon
      noop;
      rbase _ RBASE[rmForKernelRtn];
      t_ setHoldRtn, rbase _ RBASE[defaultRegion];
      Q _ rscr2;                              * restore Q
      subroutine;
      link _ t;
      return;

* This code actually causes T to be set properly and branches to the code that sets HOLD.

             set[xtask, 1];

simInit:
      t _ q,                                  at[initTloc];
simSet:
      hold&tasksim_t;                         * T init'd at simInit
      noop;                                   * this noop doesn't cause hold to count
simBlock:
      branch[simSet], block;                  * count hold, block
%
Note: if t = 14, then hold = 16 when the simulator blocks. The preempted task will execute one
instruction, then the task simulator will waken the simulator task.
%

      set[xtask, 0];

* November 6, 1978  12:07 PM                  MIDAS SUBROUTINE for testing the task simulator
testTaskSim: subroutine;
      rscr _ link;                            * save return in case we later want it
      top level;
      t _ lsh[t, 10];                         * ENTER w/ T = task sim val NOT shifted
      q _ t;                                  * simInit expects q = hold value

      subroutine;
      t _ initTlocC;
      link _ t;
      top level;
      LDTPC _ simTaskLevelC;
      notify[simTaskLevel];

      noop;
      t _ t - t;                              * t _ 0
      branch[., alu=0], t_t;                  * this shouldn't change
testTaskErr:
      branch[.], breakpoint;

      subroutine;
      link _ rscr;
      return;

fixSim: subroutine;
      t_link;                                 * save return in fixSimRtn
```

```
      fixSimRtn _ t;
      top level;

      call[makeHoldValue];                    * compose holdValue and set hardware
      call[getIMRH], t _ holdValueLocC;
      call[setHold];

      returnUsing[fixSimRtn];

zeroHoldTRscr: subroutine;
      t_4c;
      rscr_a0;
zeroHoldTRscrL:
      Hold&TaskSim_rscr;
      t_t-1, Hold&TaskSim_rscr;
      loopUntil[alu<0, zeroHoldTRscrL];
      return;
```

```
* January 18, 1979  5:18 PM                    * READ/WRITE IM
%
      The subroutines that read and write IM turn OFF hold simulator before touching IM. Before they
return to the caller, the invoke "resetHold" to reset the hold register to the contents of
"holdValueLoc". By convention, the current value of the two simulator registers is kept in
"holdValueLoc" for this express purpose. Zeroing and resetting hold is done because of hardware
restrictions: hold and polyphase instructions don't mix.

      ReadIM[] instructions are followed by a mask operation with getIMmask because of the interaction
between DWATCH (Midas facility) and LINK[0].
%

readByte3: subroutine;                        * CLOBBER T, RSCR!
      zeroHold[rscr];

      rscr _ link;                            * this routine assumes t points to IM
      link _ t;                               * it reads the least significan byte in IM

      top level;                              * read byte 3
      readim[3];
      subroutine;
      t_link;                                 * t = byte3
      t and (b1);                             * see if the data is inverted. If so, b1 will
      skpif[ALU=0];                           * 1, and we must reinvert the data.
      t _ not(t);
      t _ t and (getIMmask);                  * isolate the byte

      top level;                              * reset value of hold and return
      call[resetHold];
      subroutine;
      link _ rscr;
      return;                                 * return w/ byte in t

getIMRH: subroutine, global;                  * CLOBBER T, RSCR, RSCR2!
      zeroHold[rscr];                         * disable task/hold Sim before touching IM

      rscr _ link;                            * ENTER w/ T pointing to IM location
      link _ t;

      top level;                              * read hi byte of right half
      readim[2];
      subroutine;
      rscr2 _ link;                           * rscr2 = high byte
      (rscr2) and (b1);                       * see if the data is inverted. If so, b1 will
      skpif[ALU=0];                           * 1, and we must reinvert the data.
      rscr2 _ not(rscr2);
      rscr2 _ (rscr2) and (getIMmask);        * isolate the byte

      link _ t;                               * read low byte of right half
      top level;
      readim[3];
      subroutine;
      t _ link;                               * t = low byte, rscr2 = hi byte
      t and (b1);                             * see if the data is inverted. If so, b1 will
      skpif[ALU=0];                           * 1, and we must reinvert the data.
      t _ not(t);
      t _ t and (getIMmask);                  * isolate the byte

      rscr2 _ lsh[rscr2, 10];
      t _ t + (rscr2);                        * RETURN w/ T = IMRH

      top level;
      call[resetHold];
      subroutine;
      link _ rscr;
      return;
```

```
getIMLH: subroutine, global;                    * CLOBBER T, RSCR, RSCR2!
      zeroHold[rscr];                           * disable task/hold Sim before touching IM

      rscr _ link;                              * ENTER w/ T pointing to IM location
      link _ t;

      top level;                                * read hi byte of left half
      readim[0];
      subroutine;
      rscr2 _ link;                             * rscr2 = hi byte
      (rscr2) and (b1);                         * see if the data is inverted. If so, b1 will
      skpif[ALU=0];                             * 1, and we must reinvert the data.
      rscr2 _ not(rscr2);
      rscr2 _ (rscr2) and (getIMmask);          * isolate the byte

      link _ t;                                 * read low byte of left half
      top level;
      readim[1];
      subroutine;                               * CLOBBER T, RSCR, RSCR2!
      t _ link;                                 * t = low byte, rscr2 = hi byte
      t and (b1);                               * see if the data is inverted. If so, b1 will
      skpif[ALU=0];                             * 1, and we must reinvert the data.
      t _ not(t);
      t _ t and (getIMmask);                    * isolate the byte

      rscr2 _ lsh[rscr2, 10];
      t _ t + (rscr2);                          * RETURN w/ T = IMLH

      top level;
      call[resetHold];
      subroutine;
      link _ rscr;
      return;

putIMRH: subroutine, global;                    * T = addr, RSCR = value, clobberr RSCR2
      rscr2 _ link;
      link _ t;

      zeroHold[t];                              * disable task/hold Sim before touching IM

      top level;
      t _ rscr;
      IMRHB'POK _ t;

      call[resetHold];
      subroutine;
      link _ rscr2;
      return;

putIMLH: subroutine, global;                    * T = addr, RSCR = value, Clobber RSCR2
      rscr2 _ rscr;
      rscr _ link;
      link _ t;

      zeroHold[t];                              * disable task/hold Sim before touching IM

      top level;
      t _ rscr2;
      IMLHR0'POK _ t;

      call[resetHold];
      subroutine;
      link _ rscr;
      return;
```

```
checkFlags: subroutine, global;                 * CLOBBER T, RSCR, RSCR2
      rscr _ link;                              * this routine assumes t has a bit mask

      zeroHold[rscr2];                          * disable task/hold Sim before touching IM

      rscr2 _ flagsLocC;                        * it reads the flags word in IM
      link _ rscr2;                             * and performs t_tANDflag
      top level;
      readim[3];
      subroutine;
      rscr2 _ link;
      (rscr2) and (b1);                         * see if the data is inverted. If so, b1 will
      skpif[ALU=0];                             * 1, and we must reinvert the data.
      rscr2 _ not(rscr2);
      rscr2 _ (rscr2) and (getIMmask);          * isolate the byte

      top level;
      call[resetHold];
      subroutine;
      link _ rscr;
      return, t _ t AND(rscr2);                 * returnee can do alu=0 fast branch

checkTaskNum: subroutine;                       * enter: T=expected task num,
      rscr_t, RBASE _ rbase[currentTaskNum];    * return: T=current task num, branch condition
      t _ currentTaskNum, RBASE _ rbase[defaultRegion];     * clobber rscr, rscr2

      return, t#(rscr);                         * rtn w/ branch condition, t=current task
* number, rscr = expected task number.
```

```
* August 1, 1979  3:30 PM                    other, miscellaneous subroutines
notifyTask: subroutine;
      rscr_ link;
      bigBDispatch_t;
      top level;
      branch[dispatchTbl];
set[nloc, 6600];
dispatchTbl:
      branch[nxit], notify[0],                at[nloc,0];
      branch[nxit], notify[1],                at[nloc,1];
      branch[nxit], notify[2],                at[nloc,2];
      branch[nxit], notify[3],                at[nloc,3];
      branch[nxit], notify[4],                at[nloc,4];
      branch[nxit], notify[5],                at[nloc,5];
      branch[nxit], notify[6],                at[nloc,6];
      branch[nxit], notify[7],                at[nloc,7];
      branch[nxit], notify[10],               at[nloc,10];
      branch[nxit], notify[11],               at[nloc,11];
      branch[nxit], notify[12],               at[nloc,12];
      branch[nxit], notify[13],               at[nloc,13];
      branch[nxit], notify[14],               at[nloc,14];
      branch[nxit], notify[15],               at[nloc,15];
      branch[nxit], notify[16],               at[nloc,16];
      branch[nxit], notify[17],               at[nloc,17];
      branch[.], breakpoint,                  at[nloc,20];
      branch[.], breakpoint,                  at[nloc,21];

      subroutine;
nxit:
      link _ rscr;
      return;

topLvlPostRtn:
      RBASE _ rbase[mainPostRtn];
      link _ mainPostRtn;
      return, RBASE _ rbase[defaultRegion];

scopeTrigger: subroutine;
      t _ a0, global;
      TIOA _ t, T_a1;
      return, TIOA_t;

justReturn:  * this subroutine ONLY RETURNS.  Calling justReturn forces the instruction
      return, global;                           *  (logically) after the call to occur in the
physically
* next location after the call.  This is a way of reserving a noop that can ALWAYS be
* safely patched with a "call".
```

```
* April 24, 1978  6:51 PM
      knowRbase[randomRM];
random:
      T_ LSH[rndm0, 11];                              * T_ 2^9 * R
      T_ T+(rndm0);                                   * (2^9 + 2^0)* R
      T_ LSH[T, 2];                                   * (2^11 + 2^2)* R
      T_ T+(rndm0);                                   * (2^11 + 2^2 + 2^0)* R
      T_ T+(33000C);
      T_ rndm0_ T+(31C), Return;                * +13849 (= 33031B)

      goto[random1], t _ rndm0, RBASE _ rbase[randV],   at[randomTloc,0];   knowRbase[randomRM];
      goto[random1], t _ rndm1, RBASE _ rbase[randV],   at[randomTloc,1];   knowRbase[randomRM];
      goto[random1], t _ rndm2, RBASE _ rbase[randV],   at[randomTloc,2];   knowRbase[randomRM];
      goto[random1], t _ rndm3, RBASE _ rbase[randV],   at[randomTloc,3];   knowRbase[randomRM];
      goto[random1], t _ rndm4, RBASE _ rbase[randV],   at[randomTloc,4];   knowRbase[randomRM];
      goto[random1], t _ rndm5, RBASE _ rbase[randV],   at[randomTloc,5];   knowRbase[randomRM];
      goto[random1], t _ rndm6, RBASE _ rbase[randV],   at[randomTloc,6];   knowRbase[randomRM];
      goto[random1], t _ rndm7, RBASE _ rbase[randV],   at[randomTloc,7];

random1:
      return, t _ randV _ (randV)+t;
      knowRbase[defaultRegion];

* code below modified to save/restore/use rndm0 rather than randV.
saveRandState: subroutine;                        * remember random number seed
      RBASE _ rbase[randV];
      oldRandV _ rndm0;
      oldRandX _ randX;
      return, RBASE _ rbase[defaultRegion];

restoreRandState: subroutine;                     * restore remembered random number seed
      RBASE _ rbase[randV];
      rndm0 _ oldRandV;
      randX _ oldRandX;
      return, RBASE _ rbase[defaultRegion];

getRandV: subroutine;
      RBASE _ rbase[randV];
      RETURN, t _ rndm0, RBASE _ rbase[defaultRegion];
setRandV: subroutine;
      RETURN, rndm0_ t;
cycleRandV: subroutine;
      RBASE_ rbase[randV];
      rndm0_ (rndm0)+1, RETURN, RBASE_ rbase[defaultRegion];
```

```
* January 20, 1978  3:04 PM                    'FLAGS' manipulating code

xorFlags: subroutine;                          * T = value to XOR into flags
* CLOBBER RSCR, RSCR2, T
      rscr2 _ t;                               * save bits
      t _ link;
      xorFlagsRtn _ t;
      top level;

      t _ flagsLocC;
      call[readByte3];
      t _ t # (rscr2);                         * xor new bits

      rscr _ t;                                * put new value back into IM
      call[putIMRH], t _ flagsLocC;

      returnUsing[xorFlagsRtn];

xorTaskCirc: subroutine;                       * xor the flags.testTasks bit in FLAGS
* CLOBBER RSCR, RSCR2, T
      saveReturn[flagSubrsRtn];
      t _ flags.testTasks;
      call[xorFlags];
      noop;

      returnUsing[flagSubrsRtn];

xorHoldSim: subroutine;                        * xor the flags.holdSim bit in FLAGS
      saveReturn[flagSubrsRtn];
      t _ flags.holdSim;
      call[xorFlags];

      rscr_a0;                                 * whether off or on, clear holdFreqLoc
      call[putIMRH], t _ holdFreqLocC;         * holdFreq _ 0

      call[fixSim];
xorHoldSimXit:
      noop, breakpoint;

      returnUsing[flagSubrsRtn];

xorTaskSim: subroutine;                        * xor the flags.taskSim bit in FLAGS
      saveReturn[flagSubrsRtn];
      t _ flags.taskSim;
      call[xorFlags];

      rscr_a0;                                 * whether off or on, clear taskFreqLoc
      call[putIMRH], t _ taskFreqLocC;         * taskFreq _ 0

      call[fixSim];                            * fix the holdValueLoc, set hardware

xorTaskSimXit:
      breakpoint, noop;
      returnUsing[flagSubrsRtn];
      top level;

* June 22, 1978  10:15 AM
%
      This code supports the conditional simulation mechanism. disableConditionalTask is a subroutine
that requires no parameters. It clears flags.conditionOK and sets flags.conditional. It also turns off
the hold simulator.

enableConditionalTask sets flags.conditionOK and flags.conditional, then it calls makeHoldValue to
force the hold simulator into working.
%
disableConditionalTask: subroutine;
      saveReturn[flagSubrsRtn];
      call[checkFlags], t _ (r0)-1;            * use mask = -1 to force a read of all the bits
      rscr _ not (flags.conditionOK);
      rscr _ t and (rscr);
      rscr _ (rscr) or (flags.conditional);
      rscr _ (rscr) and (377C);                * isolate lower byte
```

```
      call[putIMRH], t _ flagsLocC;

      call[makeHoldValue];                    * compose a new hold value from task and
                                              * hold simulator sub values
      call[zeroHoldTRscr];                    * stop hold
      call[resetHold];                        * jam the hold register w/ holdValue
      returnUsing[flagSubrsRtn];

enableConditionalTask: subroutine;
      saveReturn[flagSubrsRtn];
      call[checkFlags], t _ (r0)-1;           * use mask = -1 to force a read of all the bits
      noop;                                   * make placement easier
      rscr _ t or (flags.conditionOK);
      rscr _ (rscr) or (flags.conditional);
      noop;                                   * make placement easier
      call[putIMRH], t _ flagsLocC;           * write the new value

      call[makeHoldValue];                    * compose a new hold value from task and
                                              * hold simulator sub values
      call[zeroHoldTRscr];                    * stop hold
      call[resetHold];                        * jam the hold register w/ holdValue
      returnUsing[flagSubrsRtn];
      top level;


* ERRORs come here!
      branch[err];
SET[ERRLOC,400];
ERR:
      BREAKPOINT,GLOBAL, AT[ERRLOC];
      GOTO[.],BREAKPOINT, AT[ERRLOC,1];
      GOTO[.], AT[ERRLOC,2];

* DATA HELD IN IM

IMdata:
            ifdef[defaultFlagsP,,set[defaultFlagsP,add[flags.taskSim!, flags.holdSim!]]];       *
define default flags if undefined

            data[(Flags: lh[0] rh[defaultFlagsP], at[flagsLoc])];                  * CONTROL FLAGS
            data[(taskFreq: lh[0] rh[0], at[taskFreqLoc])];  * task sim value
            data[(holdFreq: lh[0] rh[0], at[holdFreqLoc])];  * hold sim value
            data[(nextTask: lh[0] rh[0], at[nextTaskLoc])];  * next task value
            data[(holdValue: lh[0] rh[0], at[holdValueLoc])];           * current hold value
            data[(iterations: lh[0] rh[0], at[itrsLoc])];   * iteration count

postDone: noop;
```

```
TITLE[PREAMBLE];
%

%           NAMING CONVENTIONS:
May 18, 1981  1:29 PM   LABELS BEGIN W/ THE OPERATION BEING TESTED:
            Fix getRandom to use a new random number generator
                        Aplus1          cntFcn
February 5, 1981  1:20 PM           aLuLT0
            Adapt to current d1lang.mc
                        lsh => Left shift
September 19, 1979  10:42 AM
            Try undoing zeroHoldFix - found a different bug that probably accounts for the behavior.
                        rsh => Right shift
September 18, 1979  5:55 PM
                        lcy => Left Cycle
            Try to make zeroHoldFix reliable: apparently 3 in a row is not enough.
                        rcy => Right Cycle
June 17, 1979  5:11 PM
            Change holdValue to accommodate ifu entry points
                        Register Read/Write tests are symmetrical with RW:
April 17, 1979  10:45 PM            cntRW
            Add sim.holdMask, sim.taskMask, sim.holdShift, sim.taskShift.
                        shcRW
January 25, 1979  11:05 AM
            Change simTaskLevel to 12B
                        LOOP LABELS ARE SUFFIXED AS INNER (IL) AND OUTER(OL) LOOPS (L).
January 18, 1979  5:10 PM           cntFcnIL                * INNER LOOP
            Add currentTaskNum.     cntFcnOL                * OUTER LOOP
%                                   cntFcnXITIL             * LABEL FOR EXITING INNER LOOP
                                    cntFcnXITOL             * LABEL FOR EXITING OUTER LOOP
                                    Aplus1L                 * ONLY LOOP
%

RMREGION[DEFAULTREGION];
rv[r0,0];               rv[r1,1];               rv[rm1,177777];
rv[r01,52525];          rv[r10,125252];         rv[rhigh1,100000];
rv[rscr,0];             rv[rscr2,0];            rv[rscr3,0];
rv[rscr4,0];            rv[stackPAddr, 0];      rv[stackPTopBits, 0];
rv[klink, 0];           rv[hack0,0];            rv[hack1, 0];
rv[hack2,0];

rvrel[rmx0, 0];         rvrel[rmx1, 1];         rvrel[rmx2, 2];
rvrel[rmx3, 3];         rvrel[rmx4, 4];         rvrel[rmx5, 5];
rvrel[rmx6, 6];         rvrel[rmx7, 7];         rvrel[rmx10, 10];

* Constants from FF

nsp[PNB0,100000];
nsp[PNB1,40000];        nsp[PNB2,20000];        nsp[PNB3,10000];
nsp[PNB4,4000];         nsp[PNB5,2000];         nsp[PNB6,1000];
nsp[PNB7,400];          nsp[PNB8,200];          nsp[PNB9,100];
nsp[PNB10,40];          nsp[PNB11,20];          nsp[PNB12,10];
nsp[PNB13,4];           nsp[PNB14,2];           nsp[PNB15,1];

mc[B0,100000];
mc[B1,40000];           mc[B2,20000];           mc[B3,10000];
mc[B4,4000];            mc[B5,2000];            mc[B6,1000];
mc[B7,400]; mc[B8,200]; mc[B9,100];
mc[B10,40]; mc[B11,20]; mc[B12,10];
mc[B13,4];  mc[B14,2];  mc[B15,1];

mc[NB0,PNB0];
mc[NB1,PNB1];           mc[NB2,PNB2];           mc[NB3,PNB3];
mc[NB4,PNB4];           mc[NB5,PNB5];           mc[NB6,PNB6];
mc[NB7,PNB7];           mc[NB8,PNB8];           mc[NB9,PNB9];
mc[NB10,PNB10];         mc[NB11,PNB11];         mc[NB12,PNB12];
mc[NB13,PNB13];         mc[NB14,PNB14];         mc[NB15,PNB15];

mc[CM1,177777];mc[C77400,77400]; mc[C377,377];mc[CM2,-2];
mc[getIMmask, 377];     * isolate IM data after getIm[]!

m[noop, BRANCH[.+1]];
m[skip, BRANCH[.+2]];
m[error, ILC[(BRANCH[ERR])]];
m[skiperr, ILC[(BRANCH[.+2])] ILC[(BRANCH[ERR])]];
m[skpif, BRGO@[TS@] JMP@[.+2,#1,#2] ];
m[skpUnless, BRGO@[TS@] DBL@[.+1,.+2,#1,#2] ];
m[loopChk, BRGO@[TS@] DBL@[#1,.+1,#2,#3] ];
m[loopUntil, BRGO@[TS@] DBL@[.+1,#2,#1,#3]];* if #1 then goto .+1 else goto #2
m[loopWhile, BRGO@[TS@] DBL@[#2,.+1,#1,#3]];* if #1 then goto #2 else goto .+1
```

```
* May 18, 1981  1:36 PM

            rmRegion[rm2ForKernelRtn];
            knowRbase[rm2ForKernelRtn];

            rv[chkSimulatingRtn, 0];
            rv[fixSimRtn, 0];
            rv[chkRunSimRtn, 0];
            rv[currentTaskNum, 0];
            rmRegion[randomRM];
            knowRbase[randomRM];

            rv[rndm0, 134134];      rv[rndm1, 054206];
            rv[rndm2, 036711];      rv[rndm3, 103625];
            rv[rndm4, 117253];      rv[rndm5,154737];
            rv[rndm6, 041344];      rv[rndm7, 006712];
* rm below not used for simple random number generator.
            rv[randV,0];            * current value from random number generator
            rv[randX,0];            * current index into random number jump table
            rv[oldRandV,0];         * saved value
            rv[oldRandX,0];         * saved value

            knowRbase[defaultRegion];

            mp[flags.conditionalP, 200];        * bit that indicates conditional simulating
            mp[flags.conditionOKp, 100];        * bit that indicates conditional simulating is ok

            set[holdValueLoc, 3400];            mc[holdValueLocC, holdValueLoc];
            mc[flags.taskSim, b15]; * NOTE: The "flags" manipulation code
            mc[flags.holdSim, b14]; * works only so long as there are no more
            mc[flags.simulating, flags.taskSim, flags.holdSim];
            set[simTaskLevel, 12]; mc[simTaskLevelC, simTaskLevel];

            mc[sim.holdMask, 377]; set[sim.holdShift, 0];
            mc[sim.taskMask, 177400];  set[sim.taskShift, 10];

m[lh, byt0[ and[rshift[#1,10], 377] ] byt1[ and[#1, 377]]
            ];          * assemble data for left half of IM
m[rh, byt2[ and[rshift[#1,10], 377] ] byt3[ and[#1, 377]]
            ];          * assemble data for right half of IM

m[zeroHold, ilc[(#1 _ A0)]
            ilc[(hold&tasksim _ #1)]
            ilc[(hold&tasksim _ #1)]
            ilc[(hold&tasksim _ #1)]
            ];
```

```
* December 11, 1978  3:20 PM
%
            subroutine entry/exit macros
%
m[saveReturn, ilc[(t _ link)]
            top level[]
            ilc[(#1 _ t)]
            ];

m[saveReturnAndT, ilc[(#2 _ t)]
            ilc[(t _ link)]
            top level[]
            ilc[(#1 _ t)]
            ];

m[returnUsing, subroutine[]
            ilc[(RBASE _ rbase[#1])]
            ilc[(link _ #1)]
            ilc[(return, RBASE _ rbase[defaultRegion])]
            ];
m[returnAndBranch, subroutine[]
            ilc[(RBASE_rbase[#1])]
            ilc[(link_#1)]
            ilc[(RBASE_rbase[defaultRegion])]
            ilc[(return, PD_#2)]
            ];
m[pushReturn, subroutine[]
            ilc[(stkp+1)]
            top level[]
            ilc[(stack _ link)]
            ];          * notice that this macro doesn't clobber T !!!
m[pushReturnAndT, subroutine[]
            ilc[(stkp+1)]
            ilc[(stack&+1 _ link)]
            top level[]
            ilc[(stack_t)]
            ];
m[returnP, subroutine[]
            ilc[(link_(stack&-1))]
            ilc[(return)]
            ];
m[pReturnP,
            ilc[(stkp-1)]
            subroutine[]
            ilc[(link_(stack&-1))]
            ilc[(return)]
            ];
m[returnPAndBranch, subroutine[]
            ilc[(link_(stack&-1))]
            ilc[(return, PD_#1)]
            ];
m[pReturnPAndBranch, subroutine[]
            ilc[(stkp-1)]
            ilc[(link_(stack&-1))]
            ilc[(return, PD_#1)]
            ];

*m[getRandom, ilc[(RBASE _ rbase[randX])]
*           ilc[(randX _ (randX)+1, Bdispatch _ randX)]
*           ilc[(call[random], RBASE _ rbase[rndm0])]
*           ilc[(RBASE _ rbase[defaultRegion])]
*           ];          *  Returns random number in T, leaves RBASE= defaultRegion
m[getRandom, ilc[(RBASE_ rbase[rndm0])]
            ilc[(call[random])]
            ilc[(RBASE_ rbase[defaultRegion])]
            ];

knowRbase[defaultRegion];
```