Keystroke Functional Test (Part A)

CONNECTION TABLE

STIMULUS	MEASUREMENT
I/O MOD	I/O MOD
U4-4 U4-5	U1-4 U4-6

RESPONSE TABLE

SIGNAL	PART PIN	I/O MOD PIN	SIGNATURE
READY	U1-4	4	0015
SRDY	U4-6	26	0015

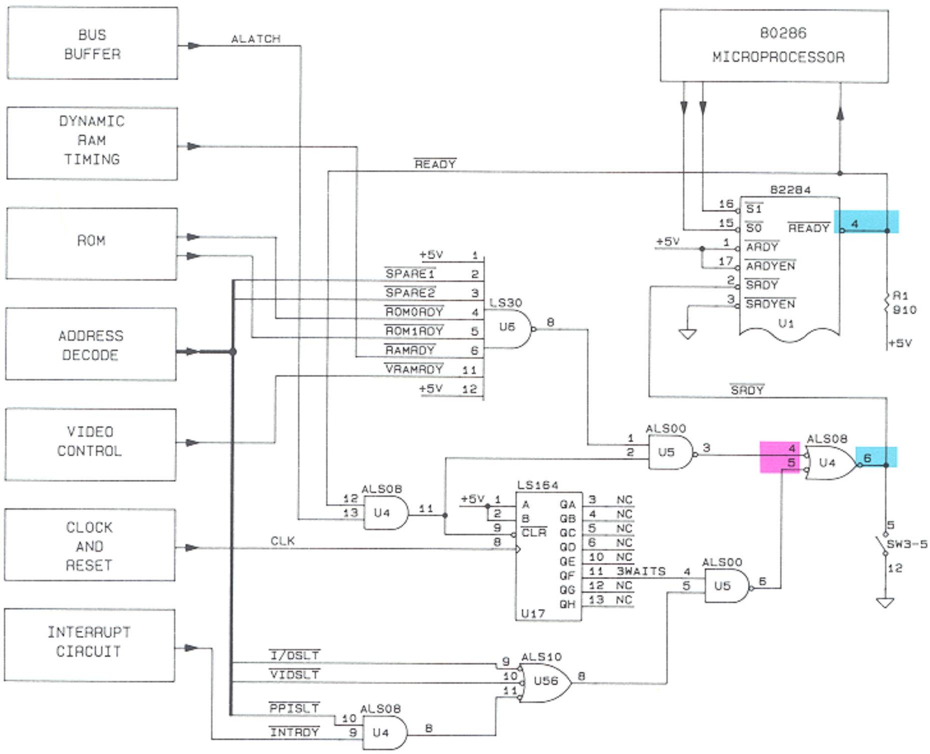


Figure 4-125: Ready Circuit Functional Test (Part A)

Keystroke Functional Test (Part B)

CONNECTION TABLE

STIMULUS	MEASUREMENT
POD TEST ACCESS SOCKET I/O MOD U4-11	I/O MOD U4-6

RESPONSE TABLE #1

SIGNAL	PART PIN	I/O MOD PIN	SIGNATURE	ASYNC LEVEL
SRDY	U4-6	26	0 0 0 0	IO

RESPONSE TABLE #2

SIGNAL	PART PIN	I/O MOD PIN	ASYNC LEVEL	TRANS COUNT
SRDY	U4-6	26	0 1	3

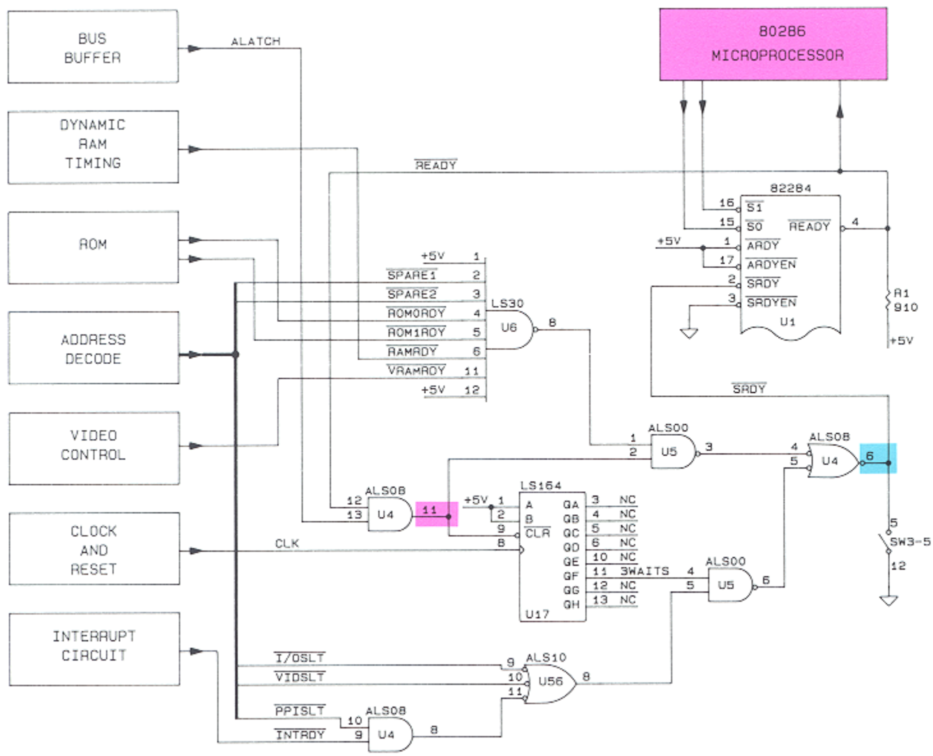


Figure 4-126: Ready Circuit Functional Test (Part B)

Keystroke Functional Test (Part C)

CONNECTION TABLE

STIMULUS	MEASUREMENT
I/O MOD U5-1 U17-9	I/O MOD U5-3

RESPONSE TABLE

SIGNAL	PART PIN	I/O MOD PIN	SIGNATURE
——	U5-3	3	0 0 0 A

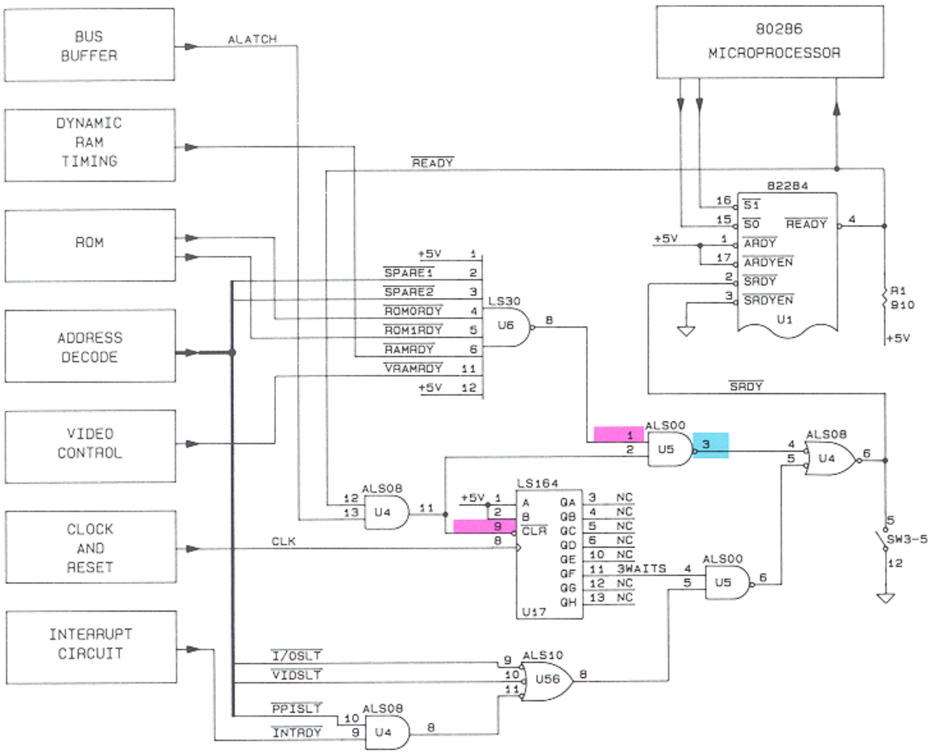


Figure 4-127: Ready Circuit Functional Test (Part C)

Keystroke Functional Test (Part D)

CONNECTION TABLE

STIMULUS	MEASUREMENT CONTROL		MEASUREMENT
I/O MOD	I/O MOD		I/O MOD
U17-9	CLOCK	U1-10	U17-11
	START	U4-11	

RESPONSE TABLE #1

SIGNAL	PART PIN	I/O MOD PIN	ASYNC LEVEL	TRANS COUNT
3WAITS	U17-11	17	01	1

RESPONSE TABLE #2

SIGNAL	PART PIN	I/O MOD PIN	TRANS COUNT
3WAITS	U17-11	17	0

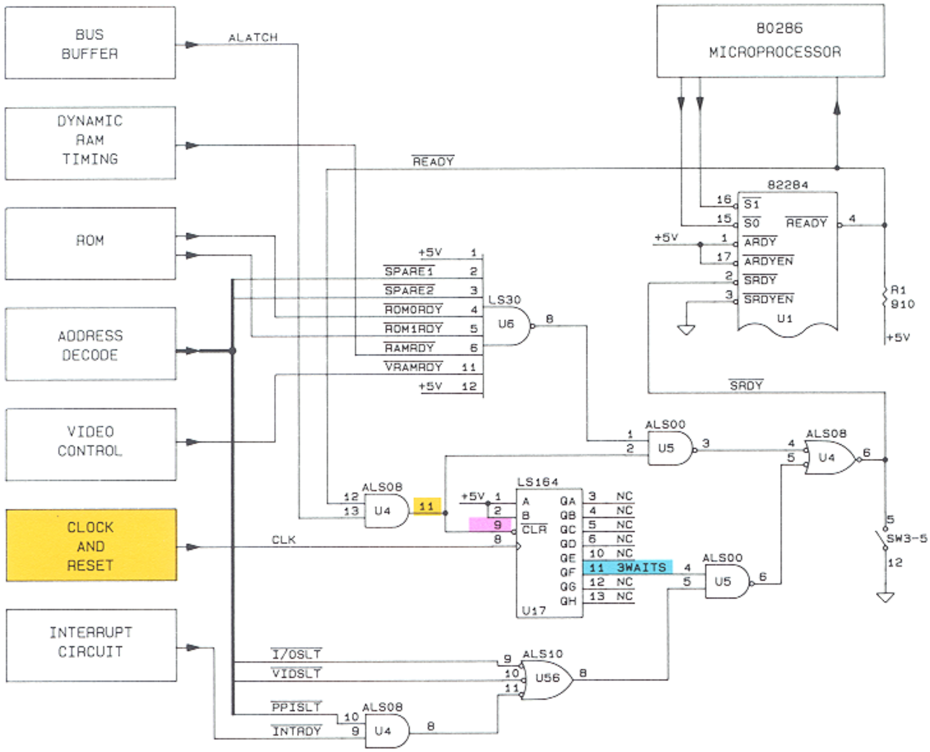


Figure 4-128: Ready Circuit Functional Test (Part D)

Programmed Functional Test 4.14.6.

The *tst_ready* program is the programmed functional test for the Ready Circuit functional block. This program checks the Ready circuit using the *gfi test* command. If the *gfi test* command fails, the *abort_test* program is executed and GFI troubleshooting begins. (See the Bus Buffer functional block for a discussion of the *abort_test* program).

The *gfi test* command executes a number of stimulus programs. The *ready_1*, *ready_2*, *ready_3*, and *ready_4* stimulus programs overdrive nodes in order to break the feedback loop in the Ready circuit. These programs will ask the operator to use a second clip on a second component so that the circuit can be overdriven.

```
program tst_ready

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the READY functional block.                          !
!                                                                         !
! This program tests the READY functional block of the Demo/Trainer.      !
! The gfi test command and I/O module are used to perform the test. The   !
! ready test involves overdriving components to break the feedback loop   !
! in the ready partition.  Two I/O module clips are required; one for     !
! measurement and one for stimulus (overdriving).                         !
!                                                                         !
! TEST PROGRAMS CALLED:                                                   !
!    abort_test (ref-pin)                  If gfi has an accusation       !
!                                          display the accusation else    !
!                                          create a gfi hint for the      !
!                                          ref-pin and terminate the test!
!                                          program (GFI begins trouble-   !
!                                          shooting).                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   if (gfi status "U1-4") = "untested" then
      print "\nl\nlTESTING READY CIRCUIT"

      podsetup 'enable ~ready' "off"
      podsetup 'report forcing' "off"

      if (gfi status "U1-4") = "untested" then gfi test "U1-4"
      if (gfi status "U1-4") = "bad" or (gfi status "U1-2") = "bad" or
                                        (gfi status "U1-3") = "bad" then
         abort_test("U1-4")
      else
         print "READY CIRCUIT PASSES"
      end if
   end if
end program
```

Stimulus Programs and Responses 4.14.7.

Figure 4-129 is the stimulus program planning diagram for the Ready Circuit functional block. The *ready_1, ready_2, ready_3,* and *ready_4* stimulus programs use one clip for measurement and a second clip to overdrive the Ready circuit in order to break the feedback loop in this circuit. *ready_5* and *ready_6* provide stimulus to measure the operation of the I/O ready generator, U17. These two stimulus programs count how many 8 Mhz clocks occur during the wait state generated by U17.

The steps to break the Ready feedback loop to diagnose a fault are shown below:

1. Overdrive inputs U4-4 and U4-5. Then measure outputs U4-6 and U1-4. The 82284 chip (U1) synchronizes the Ready output (U4-6) to the microprocessor read/write cycles. This requires the *ready_1* stimulus program to output the level, allow enough time for the signal to get synchronized, then check the level at the output U1-4.

2. Finish breaking the Ready signal feedback loop by overdriving inputs U4-12 and U4-13, then measure the outputs U4-11, U5-3, and U4-6. In order to measure U5-3 and U4-6, the other inputs U5-1 and U4-5 must be held high so the signals will flow through the AND gates. The *ready_4* stimulus program performs this step.

3. Hold the node with output source U4-11 high. This allows signals from U6 to flow through U5-3 to U4-6. At the same time, holding U4-11 high causes output U17-11 to stabilize at a high state, allowing signals from U56 to ripple through U5-6 to U4-6. Now use the pod to exercise the Ready Circuit inputs that are driven by the Address Decode functional block. The *ready_2* stimulus program performs this sequence for all components that can be forced to use zero wait states. It does this by disabling U17 (all

components except RAM and Video RAM). Since the pod has turned ~READY ENABLE OFF, the pod generates a sync pulse with zero wait states. Because the RAM and Video RAM return wait states, taking signature measurements on RAM and Video RAM will turn out to be unstable. To solve this problem, *ready_2* accesses all components except RAM and Video RAM. Then the *ready_3* stimulus program performs a similar operation, but exercises only RAM and Video RAM. *ready_3* responses are characterized by asynchronous level history and transition counts to allow the RAM and Video RAM wait state signals to be measured.

4. Measure the I/O component wait state generator, U17. The Clear input at U17-9 is toggled low. At the same time a measurement using external Clock (and Start) is made. The External Clock line is connected to the 8 MHz clock CLK and the Start line is connected to the node which includes U17-9. A Stop Count is set and transition counts and level history are measured. The *ready_6* stimulus program uses a Stop Count of four clocks and the response is expected to be low level history and zero transitions, indicating that the wait state output was low for at least four clocks. The *ready_5* stimulus program uses a Stop Count of six clocks. In this case, a response of high and low level history is expected, and a transition count of 1 is expected. These results indicate that the wait state finished within six clock cycles.

Advice for Making GFI Work in the Presence of Ready Faults

When a Ready fault exists, a forcing-line fault condition will be generated. However, the pod must ignore the Ready forcing-line fault condition so that the stimulus program will execute completely. Otherwise, a fault condition would be generated and GFI would terminate. To turn this report off, a SETUP REPORT FORCING ~READY OFF command can be performed. When this is done, the pod will continue to respond to the Ready signal, but will not generate a fault message. If the Ready signal is stuck high, the pod will cause the 9100A/9105A to generate a pod timeout fault condition. To cure this, a SETUP ENABLE ~READY OFF command is performed. At this point, GFI will work properly and Ready problems can be isolated to the failing component or node.

More generally, GFI works best if every stimulus program turns all reporting conditions off. In addition, those stimulus programs that create activity in the kernel area, may need to turn off Enable Ready. All Demo/Trainer UUT stimulus programs related to the address bus, data bus, control signals, address decoding, interrupts, and ready circuitry turn the Ready Enable off at the beginning of the stimulus program and the turn Ready Enable back on at the end of the program.

One more note: the 80286 microprocessor uses a separate bus controller that has no feedback lines to the microprocessor. When the pod disables the Ready input and performs zero wait state operations regardless of the Ready input, the bus controller can get out of synchronization from the pod and may get confused. When this happens, an *enabled_line_timeout* fault condition is generated. The solution is to provide a handler for that fault condition in each stimulus program that enables and disables Ready. The handler for the fault condition should call a program which performs a recovery procedure. The recovery procedure depends on the UUT. Usually, forcing the Ready line active or performing a Reset will recover synchronization. Or, by disabling Ready and then performing a read or write in memory space followed by enabling Ready may recover synchronization of the 80286 pod and the bus controller. Most other microprocessors do not have this problem.

Stimulus Program Planning

PROGRAM: READY_1

OVERDRIVES U4-6 TO CHECK THE SYNCHRONIZED READY OUTPUT

MEASUREMENT AT:

U1-4
U4-6

PROGRAM: READY_2

OVERDRIVES THE NODE AT U4-11 AND ALSO EXERCISES THE READY RETURN LINES (EXCEPT VRAM AND VRAMRDY)

MEASUREMENT AT:

U4-6,8
U5-3,6
U6-8
U56-8

PROGRAM: READY_3

OVERDRIVES THE NODE AT U4-11 AND EXERCISES THE READY RETURN LINES VRAM AND VRAMRDY

MEASUREMENT AT:

U4-6
U5-3
U6-8

PROGRAM: READY_4

BREAKS THE READY FEEDBACK LOOP BY OVERDRIVING THE NODE AT U4-11

MEASUREMENT AT:

U4-11
U5-3

PROGRAM: READY_5

OVERDRIVES THE INPUT TO THE I/O WAIT STATE GENERATOR AND CHECKS THAT THE OUTPUT U17-11 TRANSITIONS FROM LOW TO HIGH WITHIN 7 CLOCKS OF THE INPUT U17-9

MEASUREMENT AT:

U17-11

PROGRAM: READY_6

OVERDRIVES THE INTUT TO THE I/O WAIT STATE GENERATOR AND CHECKS THAT THE OUTPUT U17-11 TRANSITIONS FROM HIGH TO LOW WITHIN 4 CLOCKS AFTER U17-9 GOES HIGH

MEASUREMENT AT:

U17-11

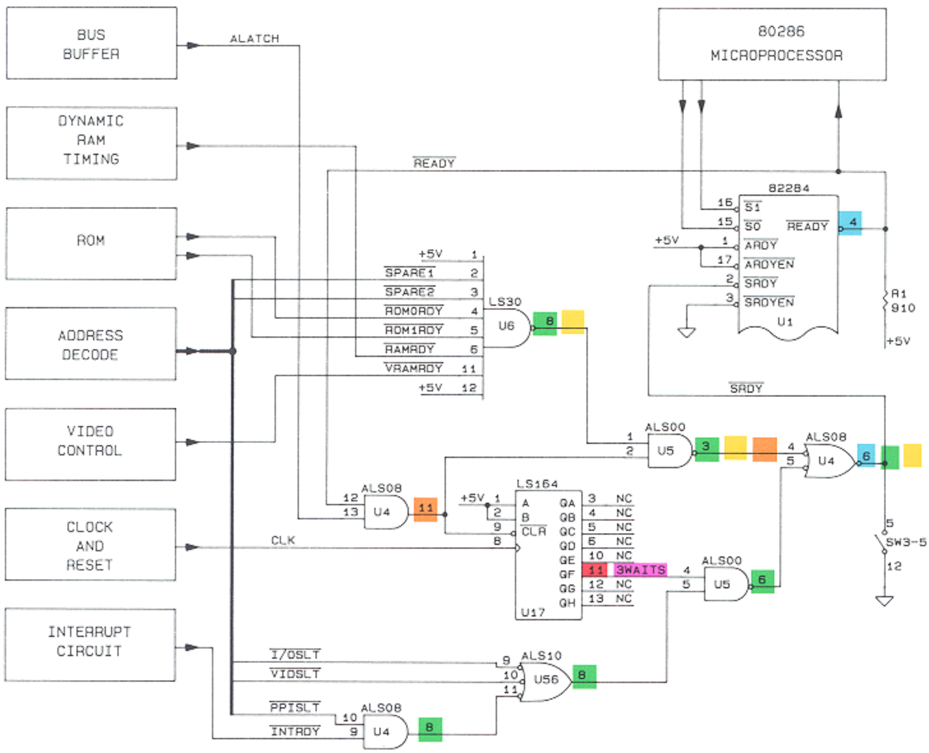


Figure 4-129: Ready Circuit Stimulus Program Planning

```
program ready_1

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM overdrives U4 in ready circuit.                       !
!                  Characterizes U4-6 and U1-4.                          !
!                                                                        !
! Stimulus programs and response files are used by GFI to backtrace     !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                        !
! This stimulus program is one of the programs which creates activity   !
! in the kernel area of the UUT.  These programs create activity with   !
! or without the ready circuit working properly.  Because of this, all  !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                         !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    recover     ()                       The 80286 microprocessor has a!
!                                         bus controller that is totally!
!                                         separate from the pod.  In    !
!                                         some cases the pod can get out!
!                                         of sync with the bus control- !
!                                         ler.  The recover program     !
!                                         resynchronizes the pod and the!
!                                         bus controller.               !
!                                                                        !
! GRAPHICS PROGRAMS CALLED:                                              !
!    (none)                                                              !
!                                                                        !
! Global Variables Modified:                                             !
!    recover_times                        Reset to Zero                  !
!                                                                        !
!                                                                        !
! Local Variables Modified:                                              !
!    measure_dev                          Measurement device             !
!    stimulus_dev                         Stimulus device (overdrives)   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-130: Stimulus Program *(ready_1)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle
handle pod_timout_no_clk
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine measurement device

   if (gfi control) = "yes" then
      measure_dev = gfi device
      measure_ref = gfi ref
   else
      print "Enter reference name of part to measure:"
      print "    (Chose U1, U4, U14 or U15)"
      measure_ref = "" \ input measure_ref
      if measure_ref <> "U14" then
         measure_dev = clip ref measure_ref
      else
         probe ref "U14-63" \ measure_dev = "/probe"
      end if
   end if

! Determine stimulus device

   if measure_ref = "U4" then
      stimulus_dev = measure_dev
   else
      print "\07\1B[2J\1B[201\1B[3;1f        USING \1B[7mSECOND\1B[0m CLIP."
      stimulus_dev = clip ref "U4"
      print "\1B[20h"
   end if
   print "Stimulus Program READY_1"
```

(continued on the next page)

Figure 4-130: Stimulus Program *(ready_1) - continued*

```
! Setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'standby function off'
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   reset device measure_dev
   reset device stimulus_dev
   sync device measure_dev, mode "int"

! Perform Stimulus

   arm device measure_dev
      writepin device "U4", pin 4, level "1", mode "latch"
      writepin device "U4", pin 5, level "1", mode "latch"
      strobeclock device measure_dev
      writepin device "U4", pin 4, level "0", mode "latch"
      writepin device "U4", pin 5, level "1", mode "latch"
      strobeclock device measure_dev
      writepin device "U4", pin 4, level "1", mode "latch"
      writepin device "U4", pin 5, level "1", mode "latch"
      strobeclock device measure_dev
      writepin device "U4", pin 4, level "1", mode "latch"
      writepin device "U4", pin 5, level "0", mode "latch"
      strobeclock device measure_dev
      writepin device "U4", pin 4, level "1", mode "latch"
      writepin device "U4", pin 5, level "1", mode "latch"
      strobeclock device measure_dev
   readout device measure_dev

   clearoutputs device stimulus_dev
   podsetup 'standby function on'
   podsetup 'enable ~ready' "on"

end program
```

Figure 4-130: Stimulus Program *(ready_1) - continued*

STIMULUS PROGRAM NAME: READY_1
DESCRIPTION: SIZE: 94 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
	Response Data						
U4-6	I/O MODULE	0015	1	0	TRANS		
U1-4	PROBE	0015	1	0	TRANS		
U1-4	I/O MODULE	0015	1	0	TRANS		

Figure 4-131: Response File *(ready_1)*

```
program ready_2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM overdrives U4 in ready circuit.                         !
!                  Characterizes U4-6 and U1-4.                            !
!                                                                          !
! Stimulus programs and response files are used by GFI to backtrace        !
! from a failing node.  The stimulus program must create repeatable UUT    !
! activity and the response file contains the known-good responses for    !
! the outputs in the UUT that are stimulated by the stimulus program.      !
!                                                                          !
! This stimulus program is one of the programs which creates activity      !
! in the kernel area of the UUT.  These programs create activity with      !
! or without the ready circuit working properly.  Because of this, all     !
! the stimulus programs in the kernel area must disable the READY input    !
! to the pod, then perform the stimulus, then re-enable the READY input    !
! to the pod.  The 80286 microprocessor has a separate bus controller;     !
! for this reason, disabling ready and performing stimulus can get the     !
! bus controller out of synchronization with the pod.  Two fault           !
! handlers trap pod timeout conditions that indicate the bus controller    !
! is out of synchronization.  The recover() program is executed to         !
! resynchronize the bus controller and the pod.                            !
!                                                                          !
! TEST PROGRAMS CALLED:                                                    !
!    recover    ()                       The 80286 microprocessor has a!
!                                        bus controller that is totally!
!                                        separate from the pod.  In    !
!                                        some cases the pod can get out!
!                                        of sync with the bus control- !
!                                        ler.  The recover program     !
!                                        resynchronizes the pod and the!
!                                        bus controller.               !
!                                                                          !
! GRAPHICS PROGRAMS CALLED:                                                !
!    (none)                                                                !
!                                                                          !
! Global Variables Modified:                                               !
!    recover_times                       Reset to Zero                     !
!                                                                          !
!                                                                          !
! Local Variables Modified:                                                !
!    measure_dev                         Measurement device                !
!    stimulus_dev                        Stimulus device (overdrives)      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-132: Stimulus Program *(ready_2)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle
handle pod_timout_no_clk
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine measurement device

   if (gfi control) = "yes" then
      measure_dev = gfi device
      measure_ref = gfi ref
   else
      print "Enter reference name of part to measure:"
      print "    (Chose U1, U4, U5, U6, U56 or U17)"
      measure_ref = "" \ input measure_ref
      measure_dev = clip ref measure_ref
   end if

! Determine stimulus device

   if measure_ref = "U1" then
      print "\07\1B[2J\1B[201\1B[3;1f        USING \1B[7mSECOND\1B[0m CLIP."
      stimulus_dev = clip ref "U4"
      print "\1B[20h"
   else
      stimulus_dev = measure_dev
   end if
   print "Stimulus Program READY_2"

! Setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   io_byte = getspace space "i/o", size "byte"
   mem_word = getspace space "memory", size "word"
```

(continued on the next page)

Figure 4-132: Stimulus Program *(ready_2) - continued*

```
    reset device measure_dev
    reset device stimulus_dev
    sync device measure_dev, mode "pod"
    sync device "/pod", mode "data"
    old_cal = getoffset device measure_dev
    setoffset device measure_dev, offset (1000000 - 56)

    if measure_ref = "U5" then
       writepin device "U5", pin 2,  level "1", mode "latch"
       writepin device "U5", pin 4,  level "1", mode "latch"
    else if measure_ref = "U4" or measure_ref = "U1" then
       writepin device "U4", pin 11, level "1", mode "latch"
    end if

! Stimulate ICs and capture response.

    arm device measure_dev          ! Start response capture.
       setspace (mem_word)
       read addr $30000             ! IPOLL
       read addr $40000             ! SPARE1
       read addr $50000             ! SPARE2
       read addr $E0000             ! ROM0
       read addr $F0000             ! ROM1
       setspace (io_byte)
       read addr 0                  ! VIDSLT
       read addr $2000              ! I/OSLT
       read addr $4000              ! PPISLT
    readout device measure_dev      ! End response capture.

    if stimulus_dev <> "/probe" then clearoutputs device stimulus_dev
    setoffset device measure_dev, offset old_cal
    podsetup 'enable ~ready' "on"

end program
```

Figure 4-132: Stimulus Program *(ready_2) - continued*

STIMULUS PROGRAM NAME: READY_2
DESCRIPTION: SIZE: 143 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
U4-6	I/O MODULE	0000	1	0	TRANS		
U4-8	PROBE	007E	1	0	TRANS		
U4-8	I/O MODULE	007E	1	0	TRANS		
U5-3	I/O MODULE	0086	1	0	TRANS		
U5-6	I/O MODULE	0078	1	0	TRANS		
U56-8	PROBE	0086	1	0	TRANS		
U56-8	I/O MODULE	0086	1	0	TRANS		
U6-8	I/O MODULE	0078	1	0	TRANS		

Figure 4-133: Response File *(ready_2)*

```
program ready_3

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM toggles ready circuit inputs which generate          !
!                  wait states.                                         !
!                                                                       !
! Stimulus programs and response files are used by GFI to backtrace     !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.   !
!                                                                       !
! This stimulus program is one of the programs which creates activity   !
! in the kernel area of the UUT.  These programs create activity with   !
! or without the ready circuit working properly.  Because of this, all  !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                         !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    recover     ()                         The 80286 microprocessor has a!
!                                           bus controller that is totally!
!                                           separate from the pod.  In    !
!                                           some cases the pod can get out!
!                                           of sync with the bus control- !
!                                           ler.  The recover program     !
!                                           resynchronizes the pod and the!
!                                           bus controller.               !
!                                                                       !
! GRAPHICS PROGRAMS CALLED:                                             !
!    (none)                                                             !
!                                                                       !
! Global Variables Modified:                                            !
!    recover_times                          Reset to Zero               !
!                                                                       !
! Local Variables Modified:                                             !
!    measure_dev                            Measurement device          !
!    stimulus_dev                           Stimulus device (overdrives)  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-134: Stimulus Program *(ready_3)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle
handle pod_timout_no_clk
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine measurement device

   if (gfi control) = "yes" then
      measure_dev = gfi device
      measure_ref = gfi ref
   else
      print "Enter reference name of part to measure:"
      print "    (Chose U1, U4, U5 or U6)"
      measure_ref = "" \ input measure_ref
      measure_dev = clip ref measure_ref
   end if

! Determine stimulus device

   if measure_ref = "U1" then
      print "\07\1B[2J\1B[201\1B[3;1f        USING \1B[7mSECOND\1B[0m CLIP."
      stimulus_dev = clip ref "U4"
      print "\1B[20h"
   else
      stimulus_dev = measure_dev
   end if
   print "Stimulus Program READY_3"
```

(continued on the next page)

Figure 4-134: Stimulus Program *(ready_3) - continued*

```
! Setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'standby function off'
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   io_byte = getspace space "i/o", size "byte"
   mem_word = getspace space "memory", size "word"
   reset device measure_dev
   reset device stimulus_dev
   sync device measure_dev, mode "pod"
   sync device "/pod", mode "data"
   old_cal = getoffset device measure_dev
   setoffset device measure_dev, offset (1000000 - 56)

   if measure_ref = "U5" then
      writepin device "U5", pin 2,  level "1", mode "latch"
      writepin device "U5", pin 4,  level "1", mode "latch"
   else if measure_ref = "U4" or measure_ref = "U1" then
      writepin device "U4", pin 11, level "1", mode "latch"
   end if

! Stimulate ICs and capture response.

   arm device measure_dev           ! Start response capture.
      setspace (mem_word)
      read addr 0                   ! RAM0
      read addr $10000              ! RAM1
      write addr $20000, data 0     ! VRAM (write only)
   readout device measure_dev       ! End response capture.

   clearoutputs device stimulus_dev
   setoffset device measure_dev, offset old_cal
   podsetup 'standby function on'
   podsetup 'enable ~ready' "on"

end program
```

Figure 4-134: Stimulus Program *(ready_3) - continued*

```
STIMULUS PROGRAM NAME: READY_3
DESCRIPTION:                                         SIZE:            112 BYTES

               ------------------- Response Data --------------------
   Node         Learned              Async Clk Counter                         Priority
Signal Src       With        SIG      LVL  LVL  Mode     Counter Range           Pin

U4-6          I/O MODULE              1 0       TRANS   3
U5-3          I/O MODULE              1 0       TRANS   3
U6-8          I/O MODULE              1 0       TRANS   3
```

Figure 4-135: Response File *(ready_3)*

```
program ready_4

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM overdrives U4 in ready circuit.                       !
!                  Characterizes U4-6 and U1-4.                          !
!                                                                        !
! Stimulus programs and response files are used by GFI to backtrace      !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.    !
!                                                                        !
! This stimulus program is one of the programs which creates activity    !
! in the kernel area of the UUT.  These programs create activity with    !
! or without the ready circuit working properly.  Because of this, all   !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;   !
! for this reason, disabling ready and performing stimulus can get the   !
! bus controller out of synchronization with the pod.  Two fault         !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to       !
! resynchronize the bus controller and the pod.                          !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    recover     ()                         The 80286 microprocessor has a!
!                                           bus controller that is totally!
!                                           separate from the pod.  In    !
!                                           some cases the pod can get out!
!                                           of sync with the bus control- !
!                                           ler.  The recover program     !
!                                           resynchronizes the pod and the!
!                                           bus controller.               !
!                                                                        !
! GRAPHICS PROGRAMS CALLED:                                              !
!    (none)                                                              !
!                                                                        !
! Global Variables Modified:                                             !
!    recover_times                          Reset to Zero                !
!                                                                        !
! Local Variables Modified:                                              !
!    measure_dev                            Measurement device           !
!    stimulus_dev                           Stimulus device (overdrives) !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
```

(continued on the next page)

Figure 4-136: Stimulus Program *(ready_4)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle
handle pod_timout_no_clk
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine measurement device

   if (gfi control) = "yes" then
      measure_dev = gfi device
      measure_ref = gfi ref
   else
      print "Enter reference name of part to measure:"
      print "    (Chose U4, U5 or U17)"
      measure_ref = "" \ input measure_ref
      measure_dev = clip ref measure_ref
   end if

! Determine stimulus device

   if measure_ref = "U4" then
      print "\07\1B[2J\1B[201\1B[3;1f         USING \1B[7mSECOND\1B[0m CLIP."
      stimulus_dev = clip ref "U45"
   else if measure_ref = "U5" then
      print "\07\1B[2J\1B[201\1B[3;1f         USING \1B[7mSECOND\1B[0m CLIP."
      stimulus_dev = clip ref "U17"
   else if measure_ref = "U17" then
      print "\07\1B[2J\1B[201\1B[3;1f        USING \1B[7mSECOND\1B[0m CLIP."
      stimulus_dev = clip ref "U4"
   end if
   print "\1B[20h"
   print "Stimulus Program READY_4"
```

(continued on the next page)

Figure 4-136: Stimulus Program *(ready_4) - continued*

```
! Setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   reset device measure_dev
   reset device stimulus_dev
   sync device measure_dev, mode "int"
   sync device stimulus_dev, mode "int"

   if measure_ref = "U4" then
      storepatt device "U4", pin 12, patt "10111"
      storepatt device "U4", pin 13, patt "11101"
      storepatt device "U45", pin 6, patt "00000"
      storepatt device "U45", pin 3, patt "00000"
   else if measure_ref = "U5" then
      storepatt device "U5", pin 1,   patt "11111"
      storepatt device "U17", pin 9,  patt "10101"
   else if measure_ref = "U17" then
      storepatt device "U4", pin 12, patt "10111"
      storepatt device "U4", pin 13, patt "11101"
   end if

! Provide stimulus to UUT using I/O module to overdrive.

   arm device measure_dev
      if measure_ref = "U4" then
         writepatt device "U45,U4", mode "pulse"
      else if measure_ref = "U5" then
         writepatt device "U17,U5", mode "pulse"
      else if measure_ref = "U17" then
         writepatt device "U4", mode "pulse"
      end if
   readout device measure_dev

   podsetup 'enable ~ready' "on"
end program
```

Figure 4-136: Stimulus Program *(ready_4) - continued*

STIMULUS PROGRAM NAME: READY_4
DESCRIPTION: SIZE: 78 BYTES

Node Signal Src	Learned With	SIG	Async LVL	Clk LVL	Counter Mode	Counter Range	Priority Pin
	------------------ Response Data					--------------------	
U4-11	I/O MODULE	0015	1 0		TRANS		
U5-3	I/O MODULE	000A	1 0		TRANS		

Figure 4-137: Response File *(ready_4)*

```
program ready_5

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM characterizes the ready circuit.                       !
!                                                                         !
! Stimulus programs and response files are used by GFI to backtrace       !
! from a failing node.  The stimulus program must create repeatable UUT   !
! activity and the response file contains the known-good responses for    !
! the outputs in the UUT that are stimulated by the stimulus program.     !
!                                                                         !
! This stimulus program is one of the programs which creates activity     !
! in the kernel area of the UUT.  These programs create activity with     !
! or without the ready circuit working properly.  Because of this, all    !
! the stimulus programs in the kernel area must disable the READY input   !
! to the pod, then perform the stimulus, then re-enable the READY input   !
! to the pod.  The 80286 microprocessor has a separate bus controller;    !
! for this reason, disabling ready and performing stimulus can get the    !
! bus controller out of synchronization with the pod.  Two fault          !
! handlers trap pod timeout conditions that indicate the bus controller   !
! is out of synchronization.  The recover() program is executed to        !
! resynchronize the bus controller and the pod.                           !
!                                                                         !
! TEST PROGRAMS CALLED:                                                   !
!    recover     ()                         The 80286 microprocessor has a!
!                                           bus controller that is totally!
!                                           separate from the pod.  In    !
!                                           some cases the pod can get out!
!                                           of sync with the bus control- !
!                                           ler.  The recover program     !
!                                           resynchronizes the pod and the!
!                                           bus controller.               !
!                                                                         !
!    check_meas (device, start, stop, clock, enable)                      !
!                                           Checks to see if the measure- !
!                                           ment is complete using the    !
!                                           TL/1 checkstatus command.  If !
!                                           the measurement times out then!
!                                           redisplay connect locations.  !
!                                                                         !
! GRAPHICS PROGRAMS CALLED:                                               !
!    (none)                                                               !
!                                                                         !
! Local Variables Modified:                                               !
!    done                                   returned from check_meas()    !
!                                                                         !
! Global Variables Modified:                                              !
!    recover_times                          Reset to Zero                 !
!                                                                         !
! Local Variables Modified:                                               !
!    measure_dev                            Measurement device            !
!    stimulus_dev                           Stimulus device (overdrives)  !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

(continued on the next page)

Figure 4-138: Stimulus Program *(ready_5)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
declare numeric done = 0

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine the measurement device.

   if (gfi control) = "yes" then
      measure_dev = gfi device
      measure_ref = gfi ref
   else
      print "Enter reference name of part to measure:"
      print "    (Chose U5 or U17)"
      measure_ref = "" \ input measure_ref
      measure_dev = clip ref measure_ref
   end if
   print "Stimulus Program READY_5"

! Set addressing mode and setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'standby function off'
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   setspace( getspace( "i/o", "byte" ))
   reset device measure_dev
   sync device measure_dev, mode "ext"
   enable device measure_dev, mode "high"
   edge device measure_dev, start "+", stop "count", clock "-"
   stopcount device measure_dev, count 7
```

(continued on the next page)

Figure 4-138: Stimulus Program *(ready_5) - continued*

```
! Prompt user to connect external lines.

   if measure_ref = "U17" then
      connect device measure_dev, start "U4-11", clock "U1-10", common "gnd"
   else
      connect device measure_dev, start "U17-9", clock "U1-10", common "gnd"
   end if

! External lines determine measurement.

   loop until done = 1
      arm device measure_dev
         read addr 0
         done = check_meas(measure_dev,"U4-11", "*", "U1-10", "*")
      readout device measure_dev
   end loop

   clearoutputs device measure_dev
   podsetup 'standby function on'
   podsetup 'enable ~ready' "on"
end program
```

Figure 4-138: Stimulus Program *(ready_5) - continued*

```
STIMULUS PROGRAM NAME: READY_5
DESCRIPTION:                                       SIZE:            69 BYTES

          ------------------- Response Data --------------------
   Node      Learned               Async Clk Counter                      Priority
Signal Src    With          SIG     LVL  LVL  Mode     Counter Range        Pin

U17-11      I/O MODULE              1 0      TRANS   1
```

Figure 4-139: Response File *(ready_5)*

```
program ready_6

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! STIMULUS PROGRAM to wiggle all address lines from the uP.              !
!                                                                         !
! Stimulus programs and response files are used by GFI to backtrace      !
! from a failing node.  The stimulus program must create repeatable UUT !
! activity and the response file contains the known-good responses for  !
! the outputs in the UUT that are stimulated by the stimulus program.    !
!                                                                         !
! This stimulus program is one of the programs which creates activity    !
! in the kernel area of the UUT.  These programs create activity with    !
! or without the ready circuit working properly.  Because of this, all   !
! the stimulus programs in the kernel area must disable the READY input !
! to the pod, then perform the stimulus, then re-enable the READY input !
! to the pod.  The 80286 microprocessor has a separate bus controller;  !
! for this reason, disabling ready and performing stimulus can get the  !
! bus controller out of synchronization with the pod.  Two fault        !
! handlers trap pod timeout conditions that indicate the bus controller !
! is out of synchronization.  The recover() program is executed to      !
! resynchronize the bus controller and the pod.                          !
!                                                                         !
! TEST PROGRAMS CALLED:                                                   !
!    recover     ()                       The 80286 microprocessor has a!
!                                         bus controller that is totally!
!                                         separate from the pod.  In    !
!                                         some cases the pod can get out!
!                                         of sync with the bus control- !
!                                         ler.  The recover program     !
!                                         resynchronizes the pod and the!
!                                         bus controller.               !
!                                                                         !
! GRAPHICS PROGRAMS CALLED:                                               !
!    (none)                                                               !
!                                                                         !
! Global Variables Modified:                                              !
!    recover_times                        Reset to Zero                  !
!                                                                         !
! Local Variables Modified:                                               !
!    measure_dev                          Measurement device             !
!    stimulus_dev                         Stimulus device (overdrives)   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

(continued on the next page)

Figure 4-140: Stimulus Program *(ready_6)*

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric recover_times
declare numeric done = 0

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_timeout_enabled_line
   recover()
end handle
handle pod_timeout_recovered
   recover()
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

recover_times = 0

! Let GFI determine the measurement device.

   if (gfi control) = "yes" then
      measure_dev = gfi device
      measure_ref = gfi ref
   else
      print "Enter reference name of part to measure:"
      print "    (Chose U5 or U17)"
      measure_ref = "" \ input measure_ref
      measure_dev = clip ref measure_ref
   end if
   print "Stimulus Program READY_6"

! Set addressing mode and setup measurement device.

   podsetup 'enable ~ready' "off"
   podsetup 'standby function off'
   podsetup 'report power' "off"
   podsetup 'report forcing' "off"
   podsetup 'report intr' "off"
   podsetup 'report address' "off"
   podsetup 'report data' "off"
   podsetup 'report control' "off"
   setspace( getspace( "i/o", "byte" ))
   reset device measure_dev
   sync device measure_dev, mode "ext"
   enable device measure_dev, mode "high"
   edge device measure_dev, start "+", stop "count", clock "-"
   stopcount device measure_dev, count 4
```

(continued on the next page)

Figure 4-140: Stimulus Program *(ready_6) - continued*

```
! Prompt user to connect external lines.

   if measure_ref = "U17" then
      connect device measure_dev, start "U4-11", clock "U1-10", common "gnd"
   else
      connect device measure_dev, start "U17-9", clock "U1-10", common "gnd"
   end if

! External lines determine measurement.

   loop until done = 1
      arm device measure_dev
         read addr 0
         done = check_meas(measure_dev,"U4-11", "*", "U1-10", "*")
      readout device measure_dev
   end loop

   clearoutputs device measure_dev
   podsetup 'standby function on'
   podsetup 'enable ~ready' "on"

end program
```

Figure 4-140: Stimulus Program *(ready_6) - continued*

```
STIMULUS PROGRAM NAME: READY_6
DESCRIPTION:                                      SIZE:            70 BYTES

               ------------------- Response Data --------------------
  Node          Learned             Async Clk Counter                       Priority
Signal Src       With        SIG     LVL  LVL  Mode     Counter Range         Pin

U17-11         I/O MODULE            1 0    0 TRANS   0
```

Figure 4-141: Response File *(ready_6)*

Summary of Complete Solution for Ready Circuit — 4.14.8.

The entire set of programs and files needed to test and GFI troubleshoot the Ready Circuit functional block is shown below. The format below is similar to a 9100A/9105A UUT directory (you could consider the functional block to be a small UUT), but in addition shows the use of each program and the location in this manual for each file.

UUT DIRECTORY
(Complete File Set for Ready Circuit)

Programs (PROGRAM):

TST_READY	Functional Test	Section 4.14.5
READY_1	Stimulus Program	Figure 4-130
READY_2	Stimulus Program	Figure 4-132
READY_3	Stimulus Program	Figure 4-134
READY_4	Stimulus Program	Figure 4-136
READY_5	Stimulus Program	Figure 4-138
READY_6	Stimulus Program	Figure 4-140

Stimulus Program Responses (RESPONSE):

READY_1	Figure 4-131
READY_2	Figure 4-133
READY_3	Figure 4-135
READY_4	Figure 4-137
READY_5	Figure 4-139
READY_6	Figure 4-141

Node List (NODE):

NODELIST	Appendix B

Text Files (TEXT):

Reference Designator List (REF):

REFLIST	Appendix A

Compiled Database (DATABASE):

GFIDATA	Compiled by the 9100A

OTHER FUNCTIONAL BLOCKS AND CIRCUITS 4.15.

The 9100A/9105A provides the capability to handle a number of special circuits or situations. Among these are watchdog timers forcing lines, feedback loops, and in-circuit component testing.

Watchdog Timers 4.15.1.

Watchdog timers usually interfere with testing and troubleshooting. If your UUT has a watchdog timer, your test procedure or program must disable it before performing tests.

Many watchdog timers initiate a master reset when they detect incorrect activity. Others may use a high-priority interrupt line to reset the system.

Whenever possible, physically disable the watchdog timer with a jumper or switch provided for that purpose. If the watchdog timer cannot be disabled at the UUT, the 9100A/9105A may be able to ignore it with the SETUP POD REPORT FORCING SIGNAL ACTIVE OFF keypad command, or disable it with a command like SETUP POD ENABLE READY ON/OFF. Be very careful, however, when doing this. Read the precautions about these commands in Section 4.15.2, "Forcing Lines."

Forcing Lines 4.15.2.

In some situations, forcing lines must be disabled (disconnected from the pod microprocessor) during a test. You can do this with the SETUP POD ENABLE READY ON/OFF keypad command ("READY" is a pod-dependent choice; some pods may call this line by a different name).

Exercise care whenever you disable a forcing line. Write or read commands to circuits that generate wait states through a Ready line may become unpredictable after the Ready line is disabled at the pod.

In addition to disabling forcing lines, you can also ignore them. The SETUP POD REPORT FORCING SIGNAL ACTIVE OFF keypad command will prevent the reporting of forcing lines. In this mode, the pod behaves normally but forcing conditions are not reported by the pod to the 9100A/9105A.

Exercise care with this mode also. The pod's hardware performance is not affected and the pod will continue reacting to the forcing line. If the UUT generates a permanent wait state using a forcing line, the pod will halt and the system will display a timeout message. Other fault-indicating signals on your UUT will also be ignored if the forcing line is disabled. Be sure that your UUT hardware is not affected by the same forcing line.

Breaking Feedback Loops 4.15.3.

Microprocessor-based systems often have several feedback loops. The microprocessor and the components tied to the data and address buses form a large feedback loop. Most of the loops in the system will be broken when the microprocessor is replaced by the pod, because the pod can selectively ignore or report conditions of status and forcing lines. However, there may be additional loops which are not broken by the pod.

Figure 4-125 shows a feedback loop in the Ready functional block of the Demo/Trainer UUT. The READY output (U1-4) is fed back as an input at U4-12.

To test a functional block that contains a feedback loop, drive all of its inputs, including the inputs connected to outputs that form the feedback loop, and measure the outputs. Use the I/O module to overdrive inputs while measuring signature, level, and count at the outputs.

Visual and Acoustic Interfaces 4.15.4.

Some circuits, such as LEDs and beepers, have both electrical characteristics and visual or acoustic characteristics . In general, stimulus programs should ignore the visual or acoustic characteristics and measure only the electrical characteristics. The functional tests should prompt the test operator to verify the visual or acoustical characteristics.

If the functional test fails, use the *gfi test* command. If *gfi test* fails, start GFI troubleshooting. If the functional test fails and *gfi test* passes, the part is bad, since the part operates incorrectly but the electrical signals at the part are good.

In the case of the Parallel I/O functional block on the Demo/Trainer UUT, the functional test includes a prompt to the operator to verify the correct display on the LEDs. If the LEDs fail, the Parallel I/O functional test should perform a *gfi test,* which will run the stimulus programs and check the electrical properties. If *gfi test* passes (when the Parallel I/O functional test failed), it means that the electrical characteristics are good but the display is bad. The LEDs are bad and the operator should be prompted to replace them. If the *gfi test* fails, GFI troubleshooting can begin at the pin where the *gfi test* failed.

In-Circuit Component Tests 4.15.5.

If you wish, you can write TL/1 programs to test individual components rather than using the GFI to do so. These in-circuit component tests use a sequence of ones and zeroes defined with the TL/1 *storepatt* command and executed by the TL/1 *writepatt* command to overdrive the inputs of the component to be tested while measuring the signatures or level histories of its outputs. A test operator runs these tests by using the EXEC key to run the required program.

(This page is intentionally blank.)

Section 5
UUT Go/No-Go Functional Tests

PROGRAMMED GO/NO-GO FUNCTIONAL TESTING 5.1.

The UUT go/no-go test is the third of four modular levels in programming the 9100A, as shown in Figure 5-1. In this third level, the go/no-go test determines whether the UUT is good (passes) or bad (fails). The go/no-go test combines built-in functional test commands with functional tests designed by the programmer.

The go/no-go test is simple because it builds on the tests of functional blocks. It determines only whether the entire UUT is good or bad. It does not determine which functional block is causing a failure.

CREATING A PROGRAMMED GO/NO-GO FUNCTIONAL TEST 5.2.

Suppose a UUT has 14 functional blocks and a functional test is defined for each of them. One way to create a go/no-go test is to perform all 14 functional tests. Some blocks, however, can be tested indirectly by testing other blocks. For example, the bus buffer is assumed to be good if the ROM, RAM, and other blocks pass their tests. Therefore, a second way to create the go/no-go test is to perform functional tests only on functional blocks which cannot be tested indirectly by testing other blocks.

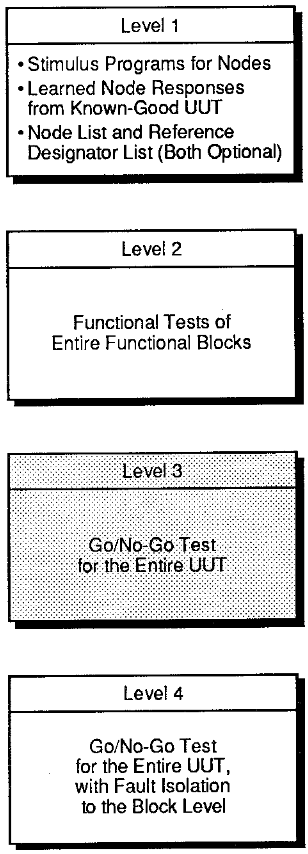


Figure 5-1: UUT Go/No-Go Functional Testing (Level 3)

Figure 5-2 shows the steps used to reach a go/no-go status decision. Care must be taken to ensure that your go/no-go test really does test the UUT for all possible faults.

Figure 5-3 shows the structure of a go/no-go functional test for the Demo/Trainer UUT. For this UUT, only six functional blocks need to be tested for the go/no-go functional test of the UUT: Microprocessor Bus, RAM, ROM, Parallel I/O, Serial I/O, and Video. The microprocessor bus test is run first because it is built-in, fast, and provides excellent diagnostic information. A failure on the microprocessor bus will cause most other circuits to fail, so it is most efficient to check this functional block first.

In the Demo/Trainer UUT, the following functional blocks are tested indirectly by the go/no-go test:

Clock and Reset
Ready Circuit
Interrupt Circuit
Bus Buffer
Dynamic RAM Timing
Address Decode
Video Control
Video RAM

Figure 5-4 is a listing of the go/no-go functional test program for the Demo/Trainer UUT. It calls the functional test for each of the functional blocks which must be tested directly for the UUT go/no-go functional test to be complete. The remaining functional blocks are tested indirectly; if they fail, one of the six blocks that is tested by the go/no-go test will fail also.

EVALUATING TEST EFFECTIVENESS 5.3.

The purpose of the go/no-go test is to determine whether the UUT is good or bad. Two measures are frequently used to evaluate how well a go/no-go functional test performs: node activity and fault coverage. Node activity is important because

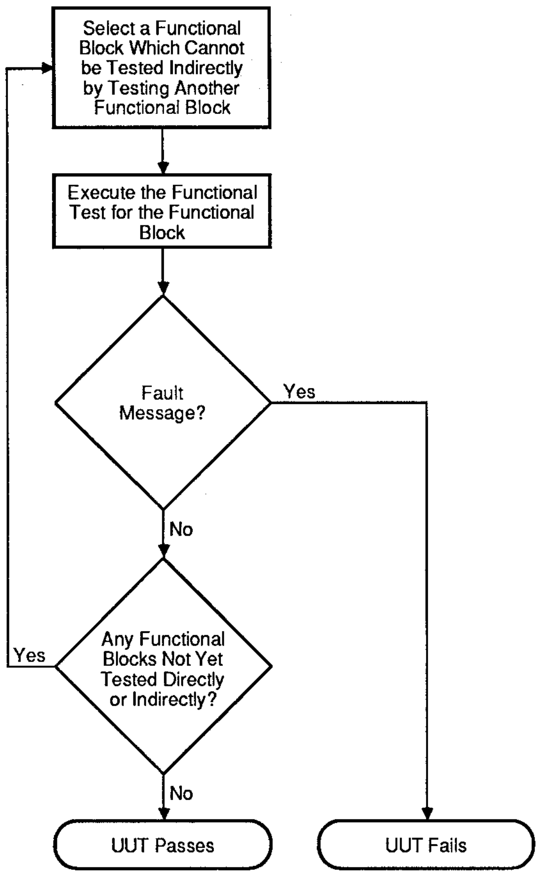


Figure 5-2: Go/No-Go Test Sequence

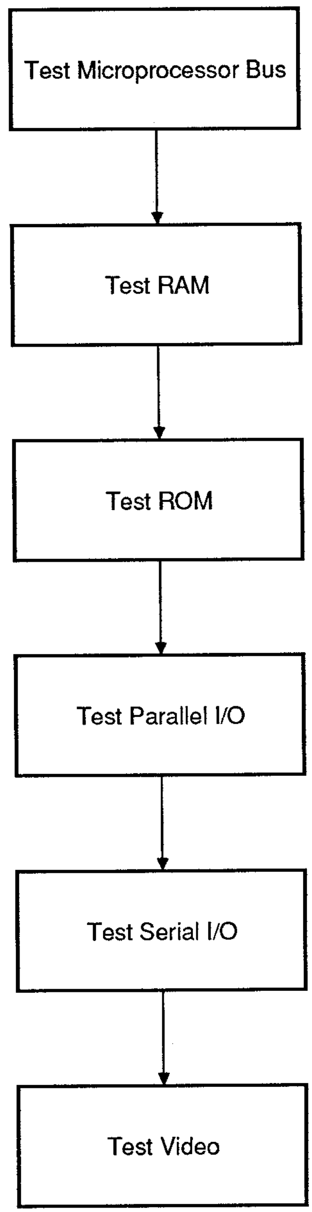


Figure 5-3: Demo/Trainer UUT Go/No-Go Test

```
program go_nogo

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! The Go/No-Go program is the highest level of the functional testing    !
! and fault handlers.  The purpose of the Go/No-Go test is to determine !
! whether the UUT is good or bad.  This program executes six programs    !
! which test the six major functional blocks (Microprocessor Bus, ROM,   !
! RAM, Parallel I/O, Serial I/O, and Video functional blocks).           !
! By testing the six major functional blocks, the remaining              !
! functional blocks are indirectly tested.                               !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    test_bus    ()                    Test the microprocessor bus,     !
!                                      buffered bus, and address        !
!                                      select logic.                    !
!                                                                        !
!    test_rom    ()                    Test the ROM functional block !
!                                      of the Demo/Trainer UUT.         !
!                                                                        !
!    test_ram    ()                    Test the RAM functional block !
!                                      of the Demo/Trainer UUT.         !
!                                                                        !
!    test_pia    ()                    Test the PARALLEL I/O            !
!                                      functional block of the          !
!                                      Demo/Trainer UUT.                !
!                                                                        !
!    test_rs232 ()                     Test the SERIAL I/O              !
!                                      functional block and the         !
!                                      Interrupt Circuit functional     !
!                                      block of the Demo/Trainer UUT.!
!                                                                        !
!    test_video ()                     Test the VIDEO circuit of the !
!                                      Demo/Trainer UUT.                !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SETUP AND SYSTEM INITIALIZATION !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   podsetup 'report power' "on"      ! Turn on reporting functions except
   podsetup 'report intr' "off"      ! interrupt which is tested in the
   podsetup 'report address' "on"    ! SERIAL I/O test (test_rs232).
   podsetup 'report control' "on"
   podsetup 'report data' "on"
   podsetup 'report forcing' "on"

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This is the go/no-go test which runs the major functional tests. !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   gfi clear                         ! CLEAR ALL GFI RECOMMENDATIONS
   connect clear "yes"               ! Clear all connect information.

   execute test_bus()
   execute test_rom()
   execute test_ram()
   execute test_pia()
   execute test_rs232()
   execute test_video()
end program
```

Figure 5-4: Go/No-Go Test for Demo/Trainer UUT each node on the UUT must be exercised for a thorough functional test.

However, activity on each node is not a sufficient evaluation of test effectiveness. In addition, you need to evaluate how well your test detects faults in the UUT. This is done by injecting faults (such as stuck lows, stuck highs, intermittent highs, or intermittent lows) at each node in the UUT while running your functional test to see if the test fails. The 9100A/9105A probe (used as a source) provides a convenient tool for this purpose.

Fault coverage is the percentage of faults that will be detected by the functional test software. It is often measured as the ratio of the number of nodes where injected faults can be detected by a test to the total number of nodes in the UUT. This ratio is usually expressed in percent. If the fault coverage is not high, you can analyze the pattern of faults that are not detected to determine additions to your test program to increase the fault coverage.

EXECUTING UUT SELF-TESTS 5.4.

Self-test routines contained in UUT memory can be executed from the 9100A/9105A by pressing the RUN UUT key at the operator's keypad and entering the UUT's starting address of the routine. These self-test routines can also be run from TL/1 programs by using the *runuut* command. Self-test routines typically save their test results in UUT RAM. The 9100A/9105A can later read the appropriate RAM addresses to get these results.

An I/O module can generate one hardware breakpoint (system interrupt) upon detection of any user-defined combination of logic-highs and logic-lows on selected I/O module lines. This feature may be invoked at the operator's keypad (SET I/O MOD COMPARE WORD command), or through program execution. Once set up for a breakpoint, the I/O module continuously monitors the specified lines while other functions (such as RUN UUT) are performed. When the breakpoint event occurs, RUN UUT execution halts. A breakpoint message will interrupt any current system activity. If a program is being executed, it may redirect the breakpoint message through a fault condition handler, as described in Section 6 of this manual.

A complete functional test for a UUT might begin with the BUS, RAM, and ROM tests, followed by execution of UUT self-test routines. By using RUN UUT breakpoints to detect addresses, data, and other UUT logic levels, the program can integrate the UUT's self-tests with 9100A/9105A functional tests.

Some pods can also generate UUT breakpoints without using the I/O module. For these pods, breakpoint-related softkeys appear when the RUN UUT key is pressed. Consult your pod manual for these pod-specific breakpoint capabilities, if any.

EXECUTING DOWNLOADED MACHINE CODE 5.5.

After part of the UUT RAM has been tested and found to be good, machine code can be downloaded to the tested RAM and executed. The machine code may be downloaded using a series of WRITE commands or the WRITE BLOCK command, which downloads an entire Motorola-format user file.

After the code is downloaded, you can execute it with the RUN UUT command, specifying the code's starting address. Although most testing can be done efficiently through the TL/1 test language, downloading machine code is useful when the code for a test already exists, when the testing must be done at machine-code speeds, or when a feature not supported by the pod must be used as part of the test.

The pod's microprocessor bus cycles are actually done at full UUT speed. The 9100A/9105A, however, is often slower than the UUT. For example, when the system performs a looping READ, each bus cycle is at full UUT speed but individual read operations are not done one immediately after the other.

Section 6
Identifying a Faulty Functional Block

After the go/no-go test determines that a UUT is faulty, the next step is to identify the failing functional block. Doing so before starting to troubleshoot will greatly improve troubleshooting efficiency because troubleshooting can begin *closer to the failure* and will take less time to reach the failing node. In addition, fault detection will be more accurate because the diagnostic test can check for special types of faults, such as bus contention, before troubleshooting begins.

Programs that identify faulty functional blocks are called diagnostic programs. Diagnostic programs, which are a subset of troubleshooting procedures, build on the UUT go/no-go test, functional tests of blocks, and stimulus programs. They are the last of the four modular levels in programming the 9100A, as shown in Figure 6-1. In this fourth programming level, fault condition handlers and *gfi hint* commands are added to the UUT go/no-go test to create a diagnostic program that traps faults and initiates tests of functional blocks that may be responsible for the fault, thereby isolating the block that is causing the UUT to fail. In addition, a failing output of the faulty block is identified as a starting point for backtracing toward the fault that causes the block to fail. At that point, GFI troubleshooting (the GFI key on the operator's keypad) can be used to backtrace to the bad node or component.

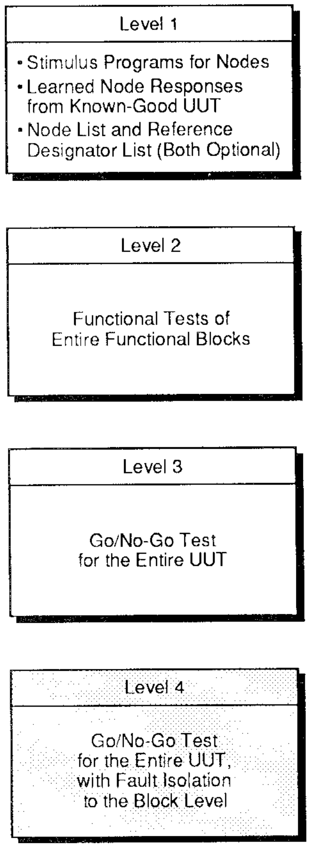


Figure 6-1: Diagnostic Programs (Level 4)

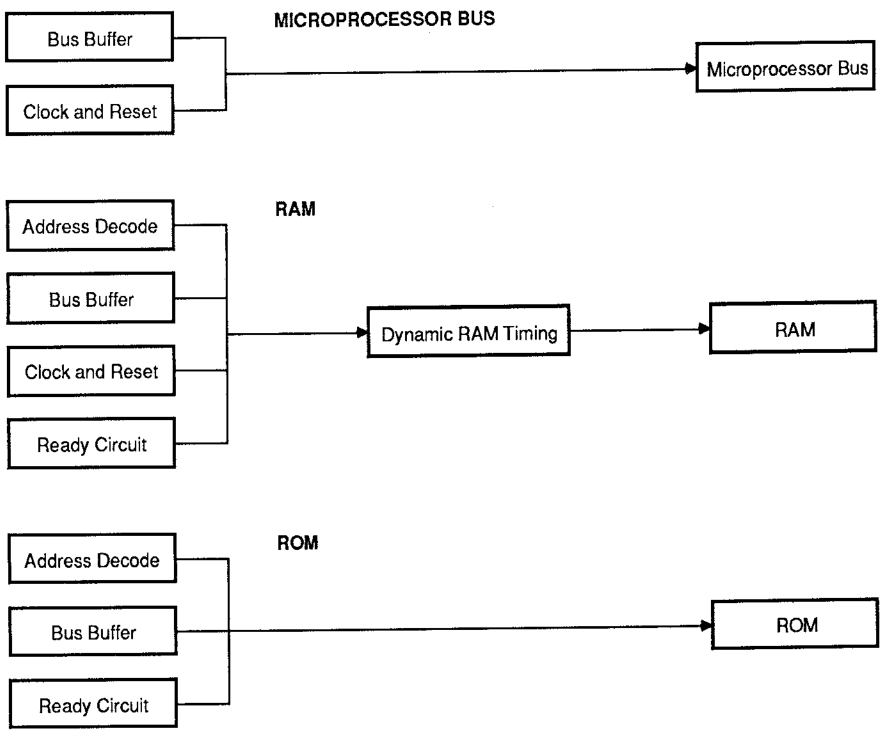
STRATEGY OF DIAGNOSTIC PROGRAMS 6.1.

The first step in developing a diagnostic strategy is to draw a diagram showing the major functional blocks used in the go/no-go functional test. Next, show all other functional blocks that provide input to these major functional blocks. Figure 6-2 shows such a diagram for the Demo/Trainer UUT. The figure shows six sets of functional blocks, one for each major functional block tested by the go/no-go functional test. The blocks on the left provide input to the blocks on the right, and the blocks tested by the go/no-go functional test are on the right side of each set.

The task of the diagnostic program is to select a failing functional block for troubleshooting and to generate an appropriate starting point (or points) where GFI can begin automated troubleshooting. When a major functional block fails, you know that one or more outputs of the block are bad. But it doesn't necessarily mean that the block itself is bad; bad inputs to the major functional block may be causing the block to fail. How do you continue from there to isolate the failing block and select an efficient starting point for GFI?

One diagnostic strategy is to test blocks that provide input to the failing major block. Isolating the block causing a failure involves tracing from the right-hand side toward the left, testing each block in the path until one is found with good inputs and bad outputs. This strategy works best when the string of blocks leading up to a major block is short. Such is the case for most of the sets of blocks in Figure 6-2.

A second diagnostic strategy, helpful when you have a longer string of blocks leading up to a failing major block, is to divide the blocks in half and begin testing a block halfway between the first block in the string and the major block at the end. If the middle block passes, keep dividing the failing string of blocks in half and testing a middle block. If the middle block fails, test the blocks to the left starting at the middle block. This second strategy would be appropriate for the Video set of blocks in Figure 6-2.



(continued on the next page)

Figure 6-2: Inputs to Functional Blocks

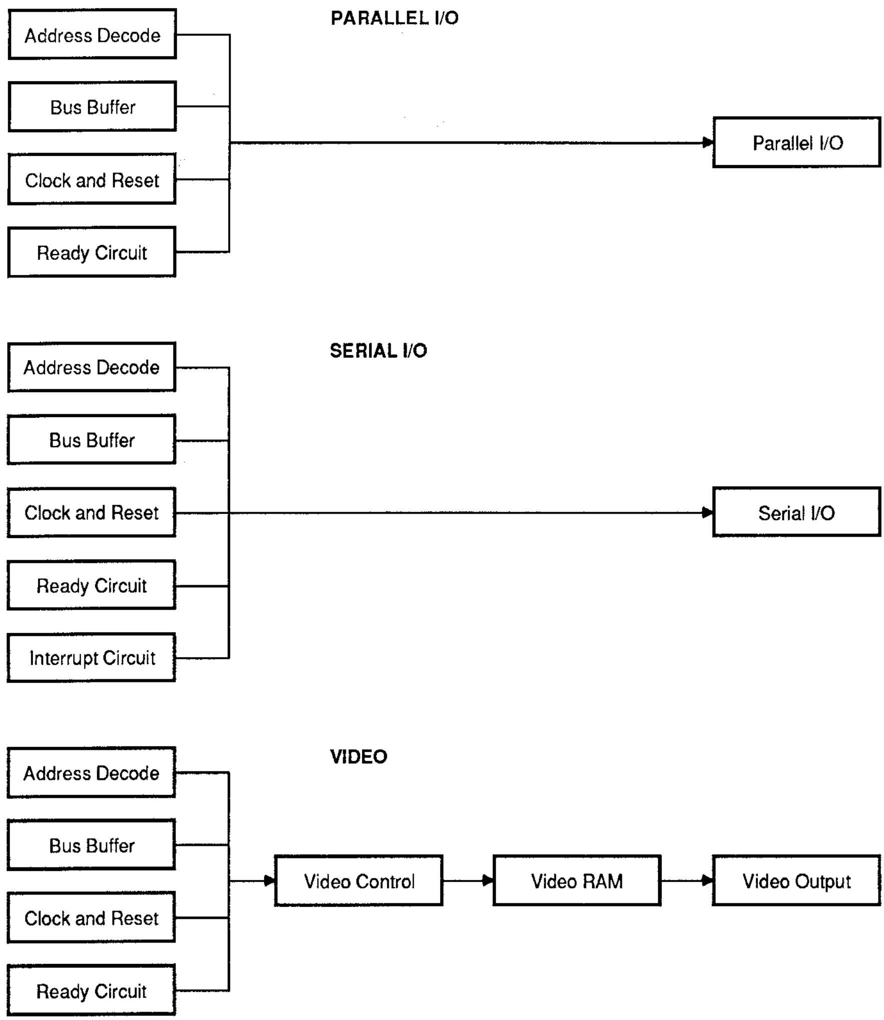


Figure 6-2: Inputs to Functional Blocks- *continued*

Another strategy, used when a fault is likely to be near a failing output pin of the failing major block, is to begin GFI backtracing directly from the failing output pin, without checking the inputs to the major functional block.

Diagnostic programs can speed up troubleshooting by starting GFI closer to the actual problem. On the other hand, isolating the failure to a very small area may require more time than is saved in reduced troubleshooting time. There is a balance between isolating the failure to a very small area and doing no isolation of the failing circuit. Decisions on when to start GFI and when to isolate the failure to a smaller area depend on your UUT and the relative cost of additional programming effort compared to the resulting savings in troubleshooting time.

IMPLEMENTING THE STRATEGY FOR DIAGNOSTIC PROGRAMS 6.2.

Figure 6-3 shows a typical process to implement a diagnostic program strategy. The diagnostic program executes a functional test for each major functional block. If a fault condition is generated during the test, the major functional block is possibly faulty. To verify this suspicion, the inputs to the functional block are checked. If the inputs are all good, then the major functional block is indeed faulty. However, if one of the inputs to a major functional block is not good, the fault probably lies in the functional blocks which provide input to the major functional block. In this case, the input functional blocks become the suspect blocks and their inputs are checked. This process continues until a block is found with all good inputs but a bad output.

When this faulty functional block is identified, appropriate GFI hints are generated to indicate the node (or nodes) where GFI should start troubleshooting.

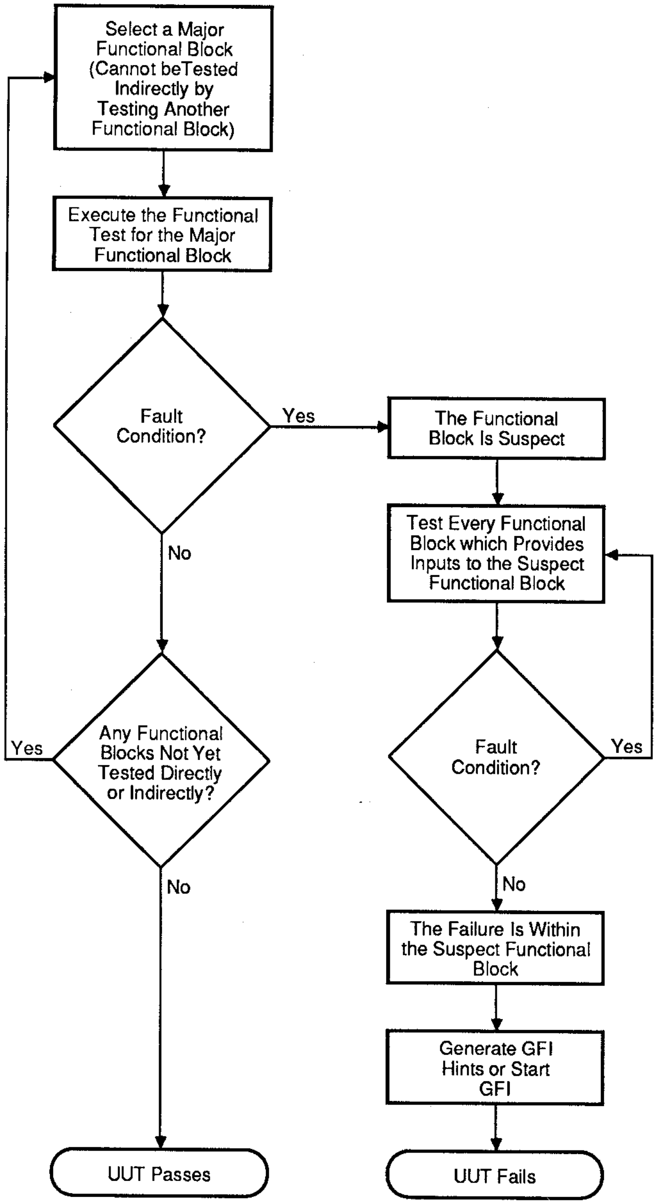


Figure 6-3: Identifying a Faulty Functional Block

DIAGNOSIS USING FAULT CONDITION HANDLERS 6.3.

Fault condition handlers provide the means for communicating 9100A/9105A functional test failure information to the operator for keystroke troubleshooting or to GFI for automated troubleshooting.

What are Fault Condition Handlers? 6.3.1.

A fault condition is generated or "raised" in one of two ways:

- A built-in TL/1 function is run, and the UUT does not respond correctly. For example, a microprocessor address line cannot be driven to logic-high during a read or write operation.
- A *fault* command is executed in a TL/1 program.

A fault condition handler is a TL/1 procedure, called by a fault condition of the same name, that responds in some way to the fault condition. For example, the handler might try to determine the cause of the fault.

Each fault condition has a name. Fault conditions created by built-in functions have defined names and parameters, listed in *TL/1 Reference Manual* appendices. Fault conditions created by your *fault* commands may have any name, including the same name used by the built-in functions.

When a fault condition is raised, the system halts execution of the current program. If your program contains a fault condition handler with the same name as the fault condition, the program statements inside the handler are executed. After the handler is finished, execution of your program resumes where it left off.

If your program does not contain an appropriate fault condition handler, execution of the program terminates and its calling program (if any) is searched for a fault condition handler with the specified fault condition name. This process continues until an appropriate handler is found. If no handler is found, a fault message will appear on the operator's display.

For more information on fault condition handlers, see Section 3.7 of the *Programmer's Manual.*

Using Fault Condition Handlers 6.3.2.

The UUT go/no-go test should test only those functional blocks that cannot be tested indirectly by other blocks. When the go/no-go test detects a failure, the diagnostic program is used to identify the failing block and to identify a failing node as a starting point for troubleshooting.

To use fault condition handlers in a diagnostic program, you need to do two programming tasks for each handler:

1. Use the *fault* command (with an appropriate fault condition that you create) to generate the fault condition if a test (or part of a test) of a functional block fails. For example, if the diagnostic program finds that the functional test of the video output circuitry fails, you might choose to generate a fault condition named *video_output.*

2. Create a handler for this fault condition. The handler should check other input blocks to isolate the failing functional block. It might also do further testing to narrow down the zone of failure within a failing functional block. And the handler will generate the appropriate starting point for GFI by using the *gfi hint* command.

A Diagnostic Test Example 6.3.3.

Suppose the video circuitry is failing. Testing begins with execution of the *go/no-go2* program, listed in Section 6.4 of this manual. This program has many fault condition handlers at the beginning, and it has six *execute* statements at the end that actually execute the go/no-go test. Each of these *execute* statements executes a different functional test program for a major functional block. And each of these functional test programs include the necessary fault condition handlers to generate GFI hints appropriate for the fault condition encountered (a listing for each of these programs is contained in Section 6.5 of this manual). The GFI hints are very important to the troubleshooting process; they are the means by which the 9100A/9105A communicates the results of its functional testing to provide efficient starting points for GFI troubleshooting.

Suppose that the failing video circuitry does not affect any of the six major functional blocks except *test_video2*. In this case, *test_bus2, test_rom2, test_ram2, test_pia2,* and *test_rs232b* all pass, but *test_video2* fails. The *test_video2* test is really the test of the Video Output functional block. If this test fails, a video fault condition is generated (suppose the *video_scan* fault condition is generated). Since the *test_video2* program has a handler for *video_scan,* the program statements inside this handler are executed.

Once the hints to GFI are passed, execution of the video fault condition handler *(video_scan)* ends, the test program *(test_video2)* ends, and the diagnostic program *(go_nogo2)* ends. A message appears on the operator's display saying that GFI hints have been generated, and that GFI should be run.

The diagnostic program is structured so that only one failure is isolated at a time. The problem should be isolated with GFI and fixed when it is detected. It is appropriate to repair an isolated fault before testing any further, since apparent multiple failures often result from one physical problem on a board. For example, a short between two nodes can appear as two failures. After a fault has been repaired, the diagnostic program should be run again to find other faults or to verify that no more faults can be found.

DIAGNOSTIC PROGRAM FOR THE DEMO/TRAINER UUT


```
program go_nogo2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! The Go/No-go program is the highest level of the functional testing   !
! and fault condition handlers.  The purpose of the Go/No-go test is to !
! determine whether the UUT is good or bad.  This program executes six  !
! programs which test the six major functional   blocks (Microprocessor !
! Bus, ROM, RAM, Parallel I/O, Serial I/O, and Video).  By testing the  !
! six major functional blocks, the remaining functional blocks are      !
! indirectly tested.                                                    !
!                                                                       !
! If the Go/No-go test detects a faulty UUT, further fault isolation is !
! performed to isolate which circuit is causing the failure.  The fault !
! condition handlers in the Go/No-go program and the other functional   !
! test programs perform the fault isolation.  The fault condition       !
! handlers included in this program are handlers for those fault        !
! conditions which may occur during any of the six major functional     !
! tests.                                                                !
!                                                                       !
! The major functional test programs include fault condition handlers   !
! for fault  conditions which are only generated within that program.   !
! The first three programs (TEST_BUS, TEST_ROM, and TEST_RAM) use       !
! built-in TL/1 tests and the built-in fault condition handlers that    !
! are documented in the 9100/9105A TL/1 Reference Manual.               !
!                                                                       !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    test_bus2                          Test the microprocessor bus,    !
!                                       buffered bus, and address       !
!                                       select logic.                   !
!                                                                       !
!    test_rom2                          Test the ROM functional block   !
!                                       of the Demo/Trainer UUT.        !
!                                                                       !
!    test_ram2                          Test the RAM functional block   !
!                                       of the Demo/Trainer UUT.        !
!                                                                       !
!    test_pia2                          Test the PARALLEL I/O           !
!                                       functional block of the         !
!                                       Demo/Trainer UUT.               !
!                                                                       !
!    test_rs232b                        Test the SERIAL I/O functional!
!                                       block and the Interrupt         !
!                                       Circuit functional block of     !
!                                       the Demo/Trainer UUT.           !
!                                                                       !
!    test_video2                        Test the VIDEO circuit of the   !
!                                       Demo/Trainer UUT.               !
!                                                                       !
!    recover                            The 80286 microprocessor has a!
!                                       bus controller that is totally!
!                                       separate from the pod.  In      !
!                                       some cases, the pod can get     !
!                                       out of sync with the bus        !
!                                       controller.  The recover        !
!                                       program resynchronizes the pod!
!                                       and the bus controller.         !
!                                                                       !
```

```
! FUNCTIONS CALLED:                                                     !
!    retry_access (access, addr, control) This function is executed when!
!                                         a pod_timeout_recovered fault !
!                                         condition occurs.  This       !
!                                         function repeats the attempted!
!                                         access that failed and        !
!                                         determines if the access can  !
!                                         be sucessfully repeated.      !
!                                                                       !
! Global Variables Modified:                                            !
!    recover_times                        Reset to Zero                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare
   global numeric recover_times           ! Count of executing recover().
end declare

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! GENERAL PURPOSE FAULT CONDITION HANDLERS                             !
!                                                                      !
! The built-in fault conditions "pod_addr_tied", "pod_ctl_tied",       !
! "pod_data_incorrect" and pod_data_tied are generated when the pod    !
! detects a stuck or tied line at the pod socket.  These fault         !
! conditions are not handled because the diagnostic message for these  !
! faults cannot be made better by additional testing.  If one of these !
! fault conditions occurs, the built-in fault message will be displayed!
! and the UUT needs to be repaired.                                    !
!                                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle pod_forcing_active (mask)
   declare string mask
   declare global numeric tlo
   declare string clear_screen = "\1B[2J"
   print on tlo ,clear_screen, "POD Forcing Lines Active fault"
   fault forcing_lines mask mask            ! Redirect fault
end handle

handle pod_interrupt_active (mask)
   declare string mask
   declare global numeric tlo
   declare string clear_screen = "\1B[2J"
   print on tlo ,clear_screen, "POD Interrupt Line Active fault"

! Get the last two characters of the 64 bit string mask and decode to INTR/NMI

   lines = val(mid(mask, len(mask)-3, 2),16)
   if (lines and $10) <> 0 then
      execute tst_intrpt()
   else if (lines and 1) <> 0 then
      fault NMI_active
   end if
end handle

handle pod_misc_fault
   fault bad_power                          ! Redirect fault
end handle
```

```
handle pod_special
end handle

handle pod_timeout_bad_pwr
   declare global numeric tlo
   declare string clear_screen = "\1B[2J"
   print on tlo ,clear_screen, "POD timeout bad power fault"
   fault bad_power                              ! Redirect fault
end handle

handle pod_timeout_enabled_line (mask)
   declare string mask
   declare global numeric tlo
   declare string clear_screen = "\1B[2J"
   print on tlo ,clear_screen, "POD Timeout Enabled line fault"
   fault forcing_lines mask mask                ! Redirect fault
end handle

handle pod_timeout_no_clk
   declare global numeric tlo
   declare string clear_screen = "\1B[2J"
   print on tlo ,clear_screen, "POD Timeout No Clock at POD Pin 31"
   execute tst_clock()                          ! Test Clock and Reset
end handle

handle pod_timeout_recovered (access_attempted, ctl, addr)
   declare string access_attempted
   declare numeric ctl = $E0000000
   declare numeric addr = $E0000000
   declare global numeric tlo
   declare string clear_screen = "\1B[2J"
   declare global numeric repeated_timeouts

   print on tlo ,clear_screen, "pod timeout recovered:  "
   podsetup 'enable ~ready' "off"
   podsetup 'enable hold' "off"
   podsetup 'report forcing' "off"
   repeated_timeouts = repeated_timeouts + 1

! DISABLE all lines that can be enabled, retry access, then turn enable
! lines on until the access cannot be repeated.  The lines that can be
! enabled on the 80286 are Hold and Ready.

   if repeated_timeouts > 10 then
      fault dead_kernel
   else if retry_access(access_attempted, ctl, addr) fails then
      fault dead_kernel
   else
      podsetup 'enable hold' "on"
      if retry_access(access_attempted, ctl, addr) fails then
         fault hold_circuit
      else
         podsetup 'enable ~ready' "on"
         if retry_access(access_attempted, ctl, addr) fails then
            execute tst_decode()
            execute tst_ready()
         else
            print on tlo ,clear_screen
         end if
      end if
   end if
end handle
```

```
handle pod_timeout_setup
end handle

handle pod_uut_power
   fault bad_power                                ! Redirect fault
end handle

handle iomod_dce
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Redirected Fault Handlers !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle forcing_lines (mask)
   declare string mask
   declare global numeric recover_times

! attempt to recover synchronization between pod and bus controller before
! testing the decode, ready or clock circuits.  If the recover procedure
! has been executed at least twice, then go ahead and test decode, ready or
! the clock circuit.

   if recover_times < 2 then
      execute recover()
   else
      lines = val(mid(mask, len(mask)-7, 8),16)
      if (lines and 1) <> 0 then
         execute tst_decode()
         execute tst_ready()
      else if (lines and $10) <> 0 then
         execute tst_clock()                      ! Test Clock and Reset
      end if

! The status lines HOLD, PEREQ, BUSY and ERROR are not used in the
! Demo/Trainer UUT.  Display a message if one of these lines is active
! and wait for the condition to be fixed.

      loop while (lines and $E2) <> 0
         print on t1o ,clear_screen
         if (lines and 2) <> 0 then
            print on t1o ,"HOLD is active; Press RESET to continue"
         else if (lines and $20) <> 0 then
            print on t1o ,"PEREQ is active; Press RESET to continue"
         else if (lines and $40) <> 0 then
            print on t1o ,"~BUSY is active; Press RESET to continue"
         else if (lines and $80) <> 0 then
            print on t1o ,"~ERROR is active; Press RESET to continue"
         end if
         wait time 2000
      end loop
   end if
end handle
```

```
handle bad_power
   declare global numeric t2o
   declare string clear_screen = "\1B[2J"
   declare global string messg
   print on t2o ,messg+"FAULT  DETECTED"
   loop until (readstatus() and $3D00) = 0
      fail($14)
      if (readstatus() and $3C00) = $3C00 then
         print on t1o ,clear_screen, "POD UUT Power"
         print on t1o ,"POWER_UP and press RESET on Trainer UUT"
         wait time 2000
         print on t1o ,clear_screen, "CONTINUING..."
      else
         if (readstatus() and $100) <> 0 then fault 'CAP failure at POD Pin 52'
         if (readstatus() and $400) <> 0 then fault 'POWER failure at POD Pin 30'
         if (readstatus() and $800) <> 0 then fault 'POWER failure at POD Pin 62'
         if (readstatus() and $1000) <> 0 then fault 'GROUND failure at POD Pin 35'
         if (readstatus() and $2000) <> 0 then fault 'GROUND failure at POD Pin 9'
      end if
   end loop
   untested($14)
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Functions                                                              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

function retry_access(ACCESS, ADDR, CTL)

! Retry last access performed using parameters from fault handlers.

   handle pod_timeout_bad_pwr
      fault
   end handle

   handle pod_timeout_enabled_line
      fault
   end handle

   handle pod_timeout_no_clk
      fault
   end handle

   handle pod_timeout_recovered
      fault
   end handle

   handle pod_timeout_setup
      fault
   end handle
```

```
   declare string ACCESS
   declare numeric CTL
   declare numeric ADDR

   if ADDR <> $E0000000 then
      address = ADDR
   else if CTL <> $E0000000 then
      address = CTL
   else
      address = 0
   end if
   if ACCESS = "READ" then
      if read addr address fails then fault
   else if ACCESS = "WRITE" then
      if write addr address, data $A5C3 fails then fault
   end if

end function

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SETUP AND SYSTEM INITIALIZATION !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   recover_times = 0
   execute recover()                   ! Recover synchronization between POD
                                       ! and the 80288 bus controller.

   podsetup 'report power' "on"        ! Turn on reporting functions except
   podsetup 'report intr' "off"        ! interrupts which is tested in the
   podsetup 'report address' "on"      ! SERIAL I/O test (test_rs232b).
   podsetup 'report control' "on"
   podsetup 'report data' "on"
   podsetup 'report forcing' "on"

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This is the go/no-go test which runs the major functional tests. !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   gfi clear                           ! CLEAR ALL GFI RECOMENDATIONS
   connect clear "yes"                 ! Clear all connect information.

   execute test_bus2 ()
   execute test_rom2 ()
   execute test_ram2 ()
   execute test_pia2 ()
   execute test_rs232b ()
   execute test_video2 ()

end program
```

FUNCTIONAL BLOCK TESTS FOR THE DEMO/TRAINER UUT DIAGNOSTIC PROGRAM 6.5.

This section contains the following functional test programs, which are necessary to support the diagnostic program for the Demo/Trainer UUT:

test_bus2	Tests the Microprocessor Bus functional block.
test_pia2	Tests the Parallel I/O function block.
test_ram2	Test the RAM functional block.
test_rom2	Tests the ROM function block.
test_rs232b	Tests the Serial I/O function block.
test_video2	Tests the video circuitry (the Video Control, Video RAM, and Video Output functional blocks).

These programs are much like the programs by the same name found in Section 4 and used in Section 5 of this manual. However, these programs also contain the necessary fault condition handlers and *gfi hint* commands to tell GFI where to start backtracing if the functional block fails.

```
program test_bus2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the Microprocessor Bus.                            !
!                                                                      !
! This program tests the unbuffered microprocessor bus, performs an    !
! access at each decoded address of the buffered bus, and checks the   !
! data bus for bus contention (where a component outputs onto the data !
! bus at incorrect times).  If bus contention is detected then the     !
! program TST_CONTEN is executed.  TST_CONTEN checks for incorrect     !
! enable line conditions on all the components on the buffered data bus.!
!                                                                      !
! TEST PROGRAMS CALLED:                                                !
!    tst_conten (addr, data_bits)         Test for bus contention on   !
!                                         the data bus by checking the !
!                                         enable lines of all devices  !
!                                         on the data bus.             !
!                                                                      !
! Local Constants:                                                     !
!    ZERO_AT_ROM0                         Address of zero data in ROM0 !
!    ZERO_AT_ROM1                         Address of zero data in ROM1 !
!    IO_BYTE                              I/O BYTE address specifier   !
!    MEM_WORD                             MEMORY WORD address specifier !
!                                                                      !
! Local Variables Modified:                                            !
!    x                                    value returned from a read   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  declare numeric ZERO_AT_ROM0 = $E002A ! Location in ROM0 where 0 exists
  declare numeric ZERO_AT_ROM1 = $F0022 ! Location in ROM1 where 0 exists

! Setup Statements

   podsetup 'enable ~ready' "on"
   podsetup 'report forcing' "on"
   IO_BYTE = getspace space "i/o", size "byte"
   MEM_WORD = getspace space "memory", size "word"

! Test the Unbuffered Microprocessor Bus.

   testbus addr 0

! Test the Extended Microprocessor Bus and Address Decoding.

   setspace (MEM_WORD)
   read addr 0                ! RAM BANK 0
   read addr $10000           ! RAM BANK 1
   write addr $20000, data 0  ! VIDEO RAM (write only)
   read addr $30000           ! INTERRUPT POLL
   read addr $E0000           ! ROM BANK 0
   read addr $F0000           ! ROM BANK 1
   setspace (IO_BYTE)
   read addr 0                ! VIDEO SELECT
   read addr $2000            ! RS232 SELECT
   read addr $4000            ! PIA SELECT
```

```
! Test for Bus Contention driving lines low by accessing unused address space

   setspace (MEM_WORD)
   x = read addr $50000                        ! SPARE-2 ADDRESS SPACE
   if x <> $FFFF then
      execute tst_conten( $50000, cpl(x) and $FFFF)
      return
   end if

! Test for Bus Contention driving lines high by reading and writing RAM
! If failure then check for bad RAM by reading zeros from 2 other devices.

   write addr 0, data 0                        ! WRITE and READ RAM addr 0
   x = read addr 0                             ! If fails then check for bad RAM
   if x <> 0 then                              ! by reading 0's at ROM0 and ROM1
      if (read addr ZERO_AT_ROM0) <> 0 then
         if (read addr ZERO_AT_ROM1) <> 0 then
            execute tst_conten( 0, x)
            return
         end if
      end if
   end if

end program
```

```
program test_pia2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the PARALLEL I/O functional block.                 !
!                                                                       !
! This program tests the PARALLEL I/O functional block of the          !
! Demo/Trainer.  The two LEDs and the four pushbutton switches are     !
! tested.  The test operator is prompted to visually inspect the LEDs  !
! as the LEDs count a series of numbers.                                !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    abort_test (ref-pin)                 If gfi has an accusation,     !
!                                         display the accusation;       !
!                                         otherwise create a gfi hint   !
!                                         for the ref-pin and terminate !
!                                         the test program (GFI begins  !
!                                         troubleshooting).             !
!                                                                       !
! TEST FUNCTIONS CALLED:                                                !
!    keys       (key_number)              Test Demo/Trainer pushbutton  !
!                                         key key_number.  Prompt test  !
!                                         operator to push the key.     !
!                                                                       !
!    leds       (led_addr, led_name)      Test Demo/Trainer LED led_name!
!                                         which is driven by the PIA and!
!                                         has the address led_addr.     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare global numeric t1b                ! Term1 buffered output & input
declare global numeric t1i                ! Term1 unbuffered input

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  FAULT CONDITION HANDLERS:                                            !
!     These fault conditions are generated by the this program.  These !
!     handlers perform isolation of the faulty circuit.  The handlers  !
!     which isolate the LED problems perform a GFI test on the LED.    !
!     If all signals are good and the test operator has failed the LED,!
!     then the LED is accused as a bad component.                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle 'PIA LED A failed'
   declare global string rev
   declare string newline = "\nl"

   if gfi test "U32-1" fails then
      abort_test("U32-1")
   else
      if gfi test "U33-1" fails then
         abort_test("U33-1")
      else if gfi test "U33-13" fails then
         abort_test("U33-13")
      else if gfi test "U33-10" fails then
         abort_test("U33-10")
      else if gfi test "U33-8" fails then
         abort_test("U33-8")
      else if gfi test "U33-7" fails then
         abort_test("U33-7")
      else if gfi test "U33-2" fails then
         abort_test("U33-2")
```

```
      else if gfi test "U33-11" fails then
         abort_test("U33-11")
      else if gfi test "U33-6" fails then
         abort_test("U33-6")
      else
         print rev, newline,"LED A IS BAD", newline, "REPLACE LED A"
      end if
   end if
end handle

handle 'PIA LED B failed'
   declare global string rev
   declare string newline = "\nl"

   if gfi test "U46-1" fails then
      abort_test("U46-1")
   else
      if gfi test "U47-1" fails then
         abort_test("U47-1")
      else if gfi test "U47-13" fails then
         abort_test("U47-13")
      else if gfi test "U47-10" fails then
         abort_test("U47-10")
      else if gfi test "U47-8" fails then
         abort_test("U47-8")
      else if gfi test "U47-7" fails then
         abort_test("U47-7")
      else if gfi test "U47-2" fails then
         abort_test("U47-2")
      else if gfi test "U47-11" fails then
         abort_test("U47-11")
      else if gfi test "U47-6" fails then
         abort_test("U47-6")
      else
         print rev, newline, "LED B IS BAD", newline, "REPLACE LED B"
      end if
   end if
end handle

handle 'PIA KEY 1 failed'
   abort_test("U31-14")
end handle

handle 'PIA KEY 2 failed'
   abort_test("U31-15")
end handle

handle 'PIA KEY 3 failed'
   abort_test("U31-16")
end handle

handle 'PIA KEY 4 failed'
   abort_test("U31-17")
end handle
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Functions                                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

function keys(keynum)
   declare numeric keynum                      ! Number of key to test.
   declare string norm = "\1B[0m"              ! Normal video escape string
   declare string rev  = "\1B[0;7m"            ! Reverse video escape string
   declare string entry
   declare string fail = ""
   declare global numeric t1b
   declare global numeric t1i

   mask = setbit(keynum - 1)

   loop until fail = chr($D)                              ! loop until YES key
      print on t1b ,"\nlPress ", rev," UUT KEY ", keynum," ",norm," pushbutton"
      print on t1b ,"Press any 9100 key if test is stuck"
      loop until (poll channel t1i, event "input") = 1
         if ((read addr $4004) and mask) = 0 then return
      end loop
      loop until (poll channel t1i, event "input") = 0    ! Flush input buffer
         input on t1i ,entry
      end loop
      print on t1b ,"\nlPress ",rev," YES ",norm," to fail KEY ",keynum," test,"
      print on t1b ,"Press "+rev+" NO  "+norm+" to continue key test,"
      input on t1i ,fail
   end loop
   print on t1b ,"\nl\nl"
   fault                                    ! Fail Key test (set termination
end function                                ! status of function to fail.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

function leds(led_addr, led_name)
   declare numeric led_addr
   declare string led_name
   declare string key
   declare string norm = "\1B[0m"
   declare string bold = "\1B[1m"
   declare string rev = "\1B[7m"
   declare string clear_screen = "\1B[2J"
   declare string no_auto_linefeed = "\1B[20h"
   declare global numeric t1i
   declare numeric array [0:10] numbers

   numbers [0] = $C0    \    numbers [5] = $92
   numbers [1] = $F9    \    numbers [6] = $82
   numbers [2] = $A4    \    numbers [7] = $F8
   numbers [3] = $B0    \    numbers [8] = $80
   numbers [4] = $99    \    numbers [9] = $98
   NO = chr($7F)        \    YES = chr( $D)

   print norm, clear_screen, "Watch LED ", led_name, " count"
   print "Press ", rev, " ENTER ", norm, " key to start LED counting."
   input  key
   print clear_screen

   for i = 0 to 9
     write addr led_addr, data numbers [i]
     wait time 500
   next
```

```
    write addr led_addr, data $7F
    print  clear_screen, "\1B[201"
    print  "\1B[1;1fDid LED ", led_name, " display ALL segments off, then"
    print  "\1B[2;1fdigits 0 to 9, then only the Decimal Point ?"
    print  "\1B[3;fpress: "+rev+" YES "+norm+" or "+rev+" NO  "+norm
    loop until key = YES or key = NO
       input on tli ,key
       if key = NO then fault
    end loop
    write addr led_addr, data $FF \  print  no_auto_linefeed,clear_screen

end function

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! PARALLEL I/O Test.                                                         !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    tlb = open device "/term1", as "update", mode "buffered"
    tli = open device "/term1", as "input", mode "unbuffered"
    execute pia_init()

    if leds($4000, "A") fails then fault 'PIA LED A failed' \ return
    if leds($4002, "B") fails then fault 'PIA LED B failed' \ return

    if keys(1) fails then fault 'PIA KEY 1 failed' \ return
    if keys(2) fails then fault 'PIA KEY 2 failed' \ return
    if keys(3) fails then fault 'PIA KEY 3 failed' \ return
    if keys(4) fails then fault 'PIA KEY 4 failed' \ return

end program
```

```
program test_ram2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the RAM functional block.                             !
!                                                                          !
! This program tests the RAM functional block of the Demo/Trainer.  The    !
! TL/1 testramfast command is used to test the RAMs.  If the RAMs are      !
! found to be faulty, then one of twelve built-in fault conditions is      !
! generated.                                                               !
!                                                                          !
! TEST PROGRAMS CALLED:                                                    !
!    abort_test (ref-pin)                If gfi has an accusation,         !
!                                        display the accusation;           !
!                                        otherwise create a gfi hint       !
!                                        for the ref-pin and terminate     !
!                                        the test program (GFI begins      !
!                                        troubleshooting).                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  FAULT CONDITION HANDLERS:                                               !
!     Built-in testramfast fault condition handlers                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle ram_addr_fault (data_mask)
   declare numeric data_mask
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM addr line fault detected,  CONTINUING"
   fault ram_component data_bits data_mask
end handle

handle ram_addr_addr_tied (data_mask)
   declare numeric data_mask
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM addr lines tied detected,  CONTINUING"
   fault ram_component data_bits data_mask
end handle

handle ram_addr_data_tied (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM addr-data tied detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle

handle ram_addr_data_tied_unconfirmed (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM addr-data tied detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle
```

```
handle ram_data_data_tied (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM data lines tied detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle

handle ram_data_fault (data)
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM data line fault detected,  CONTINUING"
   fault ram_component data_bits data
end handle

handle ram_data_incorrect (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlBAD RAM data detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle

handle ram_data_high_tied (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM data tied high detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle

handle ram_data_low_tied (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM data tied low detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle

handle ram_cell_cell_tied (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM cells tied detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle

handle ram_cell_low_tied (data_expected, data)
   declare numeric data_expected
   declare numeric data
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRAM cell tied low detected,  CONTINUING"
   fault ram_component data_bits (data xor data_expected)
end handle
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Redirected fault handler                                             !
!                                                                        !
!   The RAM block can fail if a problem exists with the ready circuit.   !
!   So test the ready circuit, then if the ready circuit is good, use    !
!   the data bits parameter passed from the testramfast built-in fault   !
!   handlers to test the failing RAM IC.  If the RAM IC is good then     !
!   test the data bus at the bus buffers.  (Testing the data bus buffer  !
!   will detect any problem in the data bus).                            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle ram_component (data_bits)
   declare numeric data_bits
   declare string array [0:$15] ram_ic

   ram_ic[0] = "U55"    \  ram_ic[1] = "U54"           ! RAMs U55, U54
   ram_ic[2] = "U53"    \  ram_ic[3] = "U52"           ! RAMs U53, U52
   ram_ic[4] = "U51"    \  ram_ic[5] = "U50"           ! RAMs U51, U50
   ram_ic[6] = "U49"    \  ram_ic[7] = "U48"           ! RAMs U49, U48
   ram_ic[8] = "U41"    \  ram_ic[9] = "U40"           ! RAMs U41, U40
   ram_ic[10] = "U39"  \  ram_ic[11] = "U38"           ! RAMs U39, U38
   ram_ic[12] = "U37"  \  ram_ic[13] = "U36"           ! RAMs U37, U36
   ram_ic[14] = "U35"  \  ram_ic[15] = "U34"           ! RAMs U35, U34

! If ready circuit is untested, then check Ready circuit

   if(gfi status "U1-4") = "untested" then
      if gfi test "U1-4"  fails then abort_test("U1-4" )
   end if

! Check highest order ram that is failing, using ram_ic array to get refname.

   if data_bits <> 0 then
      bad_ram_ref = ram_ic[msb(data_bits)] + "-1"
      if gfi test bad_ram_ref fails then abort_test (bad_ram_ref)
   end if

! Check Data Bus buffers.

   if gfi test "U3-2"  fails then abort_test("U3-2" )
   if gfi test "U23-2" fails then abort_test("U23-2")
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST PROGRAM to test RAM CIRCUIT FUNCTIONAL BLOCKS.         !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Setup

   podsetup 'enable ~ready' "on"
   podsetup 'report forcing' "on"
   setspace space (getspace space "memory", size "word")

! Main part of test

   testramfast addr 0, upto $1FFFE, delay 250, seed 1

end program
```

```
program test_rom2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the ROM functional block.                          !
!                                                                       !
! This program tests the ROM functional block of the Demo/Trainer.  The !
! TL/1 testromfull command is used to test the ROMs.  If the ROMs are   !
! found to be faulty, then one of seven built-in fault conditions is    !
! generated.                                                            !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    abort_test (ref-pin)                 If gfi has an accusation,     !
!                                         display the accusation;       !
!                                         otherwise create a gfi hint   !
!                                         for the ref-pin and terminate !
!                                         the test program (GFI begins  !
!                                         troubleshooting).             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT CONDITION HANDLERS:                                           !
!      Built-in testromfull fault condition handlers                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle rom_sig_incorrect (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlBAD signature detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle

handle rom_addr_fault (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRom address line fault detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle

handle rom_addr_addr_tied (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRom address line tied detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle

handle rom_data_high_tied_all (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRom data all high detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle

handle rom_data_low_tied_all (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRom data all low detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle
```

```
handle rom_data_fault (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRom data line fault detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle

handle rom_data_data_tied (addr)
   declare numeric addr
   declare string clear_screen = "\1B[2J"
   print clear_screen
   print "\nlRom data lines tied detected,  CONTINUING"
   fault rom_component addr_bits addr
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Redirected fault condition handler:                                  !
!                                                                        !
!   Use failing address bits parameter passed from testromfull fault     !
!   condition handlers to gfi test the ROM bank that failed.             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle rom_component (addr_bits)
   declare numeric addr_bits

   if addr_bits >= $F0000 then
      if gfi test "U27-1" fails then abort_test("U27-11") \ return
      if gfi test "U28-1" fails then abort_test("U28-11") \ return
   else
      if gfi test "U29-1" fails then abort_test("U29-11") \ return
      if gfi test "U30-1" fails then abort_test("U30-11") \ return
   end if
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST PROGRAM to test ROM CIRCUIT FUNCTIONAL BLOCK            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Setup.

   podsetup 'enable ~ready' "on"
   podsetup 'report forcing' "on"
   setspace space (getspace space "memory", size "word")

! Main part of Test.

   testromfull addr $F0000, upto $FFFFE, addrstep 2, sig $156F
   testromfull addr $E0000, upto $EFFFE, addrstep 2, sig $B61E

end program
```

```
program test_rs232b

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the SERIAL I/O functional block.                   !
!                                                                       !
! This program tests the SERIAL I/O functional block of the Demo/       !
! Trainer.  The two RS-232 ports are tested by setting three Dip        !
! Switches to loop back the two ports (SW4-4, SW4-5 and SW6-4 loop back !
! ports A and B).  The SERIAL I/O functional block also outputs two     !
! interrupt request signals.  This program also checks the interrupt    !
! circuitry.                                                            !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    abort_test (ref-pin)                 Call fail for reference name  !
!                                         then if gfi has an accusation !
!                                         display the accusation else   !
!                                         create a gfi hint for the     !
!                                         ref-pin and terminate the test!
!                                         program (GFI begins trouble-  !
!                                         shooting).                    !
!                                                                       !
!    frc_int     ()                       POD PROGRAM forces repetitive !
!                                         interrupt acknowledge cycles  !
!                                         and returns first interrupt   !
!                                         vector found on data bus.     !
!                                                                       !
!    rd_cscd     ()                       POD PROGRAM returns the 24 bit!
!                                         interrupt cascade address that!
!                                         was found on the address bus  !
!                                         during the last interrupt     !
!                                         acknowledge cycle.            !
!                                                                       !
!    rd_rearm    ()                       POD PROGRAM returns the most  !
!                                         recent interrupt vector and   !
!                                         rearms the pod to respond to  !
!                                         the next interrupt.           !
!                                                                       !
! FUNCTIONS CALLED:                                                     !
!    sync_buffer (address, data)          Synchronize FIFO buffer in    !
!                                         DUART to be last byte received!
!                                         Receive buffer is located at  !
!                                         the value of address.  The    !
!                                         data in data is written to the!
!                                         DUART and then read until it  !
!                                         appears in the FIFO or count  !
!                                         expires.                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare
   string q                             ! used to get input from keyboard
   string rev  = "\1B[0;7m"             ! Reverse Video escape sequence
   string norm = "\1B[0m"               ! Normal Video escape sequence
end declare
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT HANDLERS:                                                    !
!      These fault conditions are generated by the this program.  These !
!      handlers verify the failure using the Probe or I/O Module and    !
!      then pass control to GFI.                                        !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle 'RS232 Port A failed'
   if gfi test "U11-35" fails then abort_test("U11-35")
end handle

handle 'RS232 Port B failed'
   if gfi test "U11-5" fails then abort_test("U11-5")
   if gfi test "U11-11" fails then abort_test("U11-11")
end handle

handle 'Interrupt failed'
   if gfi test "U10-2" fails then abort_test("U10-2")
   if gfi test "U20-9" fails then abort_test("U20-9")
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONS                                                            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

function sync_buffer( address, data )
   declare numeric address
   declare numeric data

! Synchronize FIFO buffer in DUART.  Write and then read until correct data
! is returned or count has expired.

   write addr address, data data      ! Transmit Data 31 on port A
   wait time $200
   cnt = 0  \ x = 0
   loop until x = data or cnt > 3
      x = read addr address
      cnt = cnt + 1
   end loop
end function

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the SERIAL I/O Functional Block.                  !
                          !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Set interrupt acknowledge cycles on and use the 80286
! pod specific programs rd_rearm(), frc_int() & rd_cscd().

   podsetup 'report intr' "off"
   podsetup 'intr_ack on'        ! Enable Interrupt Ack. cycles
   option = getspace space "i/o", size "byte"
   setspace (option)
   execute check_loop()
   execute rd_rearm()            ! Clear interrupts
```

```
! Main part of Test.  Verify DUART port A.

   sync_buffer( $2006, $61 )        ! Synchronize FIFO in DUART for port A
   write addr $2006, data $55       ! Transmit Data 31 on port A
   wait time $200
   if ((read addr $2002) and $F) <> $D then fault 'RS232 Port A failed' \ return
   if (read addr $2006) <> $55 then fault 'RS232 Port A failed' \ return
   write addr $2006, data $55       ! Transmit Data 31 on port A
   wait time $200
   if ((read addr $2002) and $F) <> $D then fault 'RS232 Port A failed' \ return
   if (read addr $2006) <> $55 then fault 'RS232 Port A failed' \ return

! Verify DUART port B and interrupts.

   sync_buffer( $2016, $61 )        ! Synchronize FIFO in DUART for port B
   write addr $201E, data $FF       ! set output port low
   write addr $2016, data $31       ! Transmit Data 31 on port B
   if frc_int() <> $22 then fault 'Interrupt failed' \ return
   if rd_cscd() <> $2016 then fault 'Interrupt failed' \ return
   if (readstatus() and 8) <> 8 then fault 'Interrupt failed' \ return
   if (read addr $2016) <> $31 then fault 'RS232 Port B failed' \ return
   if frc_int() <> $27 then fault 'Interrupt failed' \ return
   write addr $201C, data $FF
   if ((read addr $201A) and 2) <> 0 then fault 'RS232 Port B failed' \ return

end program
```

```
program test_video2

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the VIDEO functional block                          !
!                                                                        !
! This program tests the VIDEO functional block of the Demo/Trainer.     !
! The video test uses the gfi test command to run stimulus programs and !
! to check the outputs of the Video circuit against the stimulus program!
! response files.  The gfi test command returns a passes status if all  !
! the measured results from running the stimulus programs match the     !
! response files.  Otherwise the gfi test command returns a fails       !
! status.                                                                !
!                                                                        !
! TEST PROGRAMS CALLED:                                                  !
!    abort_test (ref-pin)                 If gfi has an accusation,      !
!                                         display the accusation;        !
!                                         otherwise create a gfi hint    !
!                                         for the ref-pin and terminate !
!                                         the test program (GFI begins  !
!                                         troubleshooting).              !
!                                                                        !
!    tst_vidctl ()                        Test program to test the video!
!                                         control functional block       !
!                                         outputs.  Returns passes       !
!                                         termination status if          !
!                                         functional block is good else !
!                                         return fails termination       !
!                                         status.                        !
!                                                                        !
!    tst_vidram ()                        Test program to test the video!
!                                         RAM functional block outputs. !
!                                         Returns passes termination     !
!                                         status if functional block is !
!                                         good else return fails         !
!                                         termination status.            !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   FAULT CONDITION HANDLERS:                                            !
!      These fault conditions are generated by the this program.  These !
!      handlers isolate the failure in the video circuit to the Video   !
!      control section, Video RAM section or the Video output section.  !
!      Once the failing Video subsection has been identified, then GFI  !
!      is started.                                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

handle video_output

   ! IF Video Control section is bad, tst_vidctl will start GFI.

   if tst_vidctl() fails then return

   ! IF Video RAM section is bad, tst_vidram will start GFI.

   if tst_vidram() fails then return

   ! Video Control and Video RAM have passed.  Video Out is bad.  Start GFI.

   abort_test("J3-9")
end handle
```

```
handle video_scan
      gfi hint "J3-8"
      gfi hint "J3-9"
      fault 'gfi hints generated' '  please  run  gfi'
end handle

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the VIDEO Functional Block.                         !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

! Setup and initialization.

   connect clear "yes"
   podsetup 'enable ~ready' "on"
   print "\nl\nl"

! Main part of Test.

   if gfi test "J3-8" fails then fault video_scan \ return
   if gfi test "J3-9" fails then fault video_scan \ return

   if gfi test "U78-11" fails then fault video_scan \ return
   if gfi test "U78-28" fails then fault video_output \ return
   if gfi test "U78-29" fails then fault video_output \ return
   if gfi test "J3-7" fails then fault video_output \ return

end program
```

(This page is intentionally blank.)

Section 7
Troubleshooting

After a failing functional block is isolated with a diagnostic program, Unguided Fault Isolation (UFI) or Guided Fault Isolation (GFI) troubleshooting can be used to backtrace to the bad node or component.

UNGUIDED FAULT ISOLATION (UFI) 7.1.

UFI troubleshooting is valuable when you need experience with stimulus programs before expanding to the GFI environment. It lets you use stimulus programs to determine whether a node is good or bad, without having to enter a node list for the UUT.

UFI is used in a manner similar to GFI: the GFI key on the operator's keypad begins the process. Unlike GFI, UFI is designed to test only output pins. When testing with the probe, the output source for a node can be characterized and the other points on the node (such as inputs) can be probed looking for the same response. However, when testing with the I/O module, only the output pins can be measured because the other pins on the node are connected to I/O module pins different from the pins UFI thinks it should be measuring.

When an operator needs to troubleshoot boards before the GFI database is developed, he can use stimulus programs in UFI mode while waiting for GFI to be completed. However, he needs to understand the UUT since UFI does not recommend the next location to test.

GUIDED FAULT ISOLATION (GFI) 7.2.

The 9100A/9105A's built-in GFI algorithm guides an operator in diagnosing a faulty circuit to the component or node level without assistance from a skilled technician.

Once a functional test or larger diagnostic program has generated a list of suspect nodes, GFI troubleshooting can begin. The GFI key on the operator's keypad starts the process. GFI begins with a bad output and tests the suspect node. Nodes are exercised with a stimulus program and determined to be good or bad by comparing their measured response to responses learned from a known-good UUT.

When a node is bad, GFI tests the inputs which affect that node and recommends which node to test next. If the output of a component is bad and all inputs to the component are good, GFI accuses the component of being bad or the output node of being loaded. The node may be shorted to another node or a defective component may be loading the node. If an input is bad and the output source for that node is good, GFI accuses the node of having an open circuit.

The GFI capability is general enough to troubleshoot most digital circuits. To apply GFI to a particular UUT, however, you will need to supply UUT-specific information to the GFI database for that UUT. The files used for this database are summarized in Section 7.5 of this manual and described fully in the Guided Fault Isolation section of the *Programmer's Manual*.

STIMULUS PROGRAMS 7.3.

Stimulus programs are TL/1 programs used by GFI or UFI to exercise UUT nodes in such a way that responses at the nodes can be analyzed and compared to responses of nodes on a known-good UUT. A typical stimulus program consists of up to 6 main parts:

1. *(As required)* - Initialize the UUT and define the measurement device.

2. *(As required)* - Setup of the pod, probe, or I/O module.

3. Use the *arm* command to start the measurement of the node response.

4. Use any commands necessary to apply the stimulus.

5. Use the *readout* command to end the measurement of the node response.

6. *(As required)* - Restore any conditions altered by the setup step above (step 2).

Stimulus programs should satisfy three very important criteria:

- The program must be independent, initializing the UUT as required. This is because GFI can begin backtracing at any node, and the state of the UUT, prior to running the stimulus, is unknown. The program must also restore any adjustments it makes to the calibration offset.
- During stimulus execution, only one pin should drive a node: that is, during the period between the arm and readout commands, one and only one pin should be a node signal source (data should flow in only one direction).
- There should be at least one stimulus program for each output to the node.

See the "Stimulus Programs" section in the *Programmer's Manual* for more detailed information on stimulus programs.

STIMULUS PROGRAM RESPONSES 7.4.

Both UFI and GFI select the appropriate stimulus programs to exercise a node to be measured and compare the actual response at the node with a stored response from a known-good UUT. These responses may be any of the following (or combinations of them):

- CRC Signature.
- Transition Count.
- Frequency.
- Asynchronous Level History.
- Synchronous Level History.

The information below summarizes each of these response measurements. See the Guided Fault Isolation section of the *Programmer's Manual* for more complete information.

Learning Responses From a Known-Good UUT 7.4.1.

The 9100A editor's LEARN function is used to learn a set of responses measured on known-good UUT nodes. Once a stimulus program is written to exercise a node, a response file can be generated. To do this, the 9100A is commanded to learn responses at a node or set of nodes and the system prompts the operator to connect the measurement device (probe or I/O module) to the component providing the node signal source. The 9100A makes a series of measurements and determines the characteristics. It learns the response with three measurements (early, normal, and late clock or sync events) to make sure the response is stable and that the measurement can be used as a reliable characterization of that node.

Node characterization may use one or more of five characteristics to determine whether the node is good or bad. You can select which of the five should be saved in a response file. GFI and UFI use these saved characteristics to determine whether a node is good or bad.

CRC Signatures 7.4.2.

It is very important to ensure that a CRC signature used in node characterization will properly identify all good UUTs, at all measurement temperatures and power supply levels. A marginal signature occurs when the measured node changes state near the clock transition or when the Start, Stop, or Clock signals are not stable. A marginal signature may appear stable on one UUT and thereby lead to a false sense of security. Other UUTs may yield different signatures because of temperature or power supply variations.

When the 9100A editor learns a signature, it attempts to identify marginal CRCs by collecting signatures with advanced clock edges, normal clock edges, and delayed clock edges. If a signature has the same value for advanced and normal clock edges, it will be suffixed by a "-" sign. If a signature has the same value for normal and delayed clock edges, it will be suffixed by a "+" sign. If all three values agree, the signature is displayed with no qualification.

A variable signature results if the Start, Stop, or Enable signals are irregular, compared to the Clock signal. In addition, since the Start, Stop, and Clock signals are edge-triggered, unstable signatures will result if the Start or Stop signal edge occurs at the same time as the Clock signal edge.

Figure 7-1 shows how to test whether the start/stop interval is stable. Connect the Clock to the clock signal you want to use. Connect the probe or I/O module to a logic-high level and connect the Start and Stop lines to the locations where you would connect these lines when making the signature measurement. If the start/stop interval is stable, a constant number of clocks will occur between the start and stop condition, and the signature will be constant. If the CRC signature is not constant, the start/stop interval is unstable.

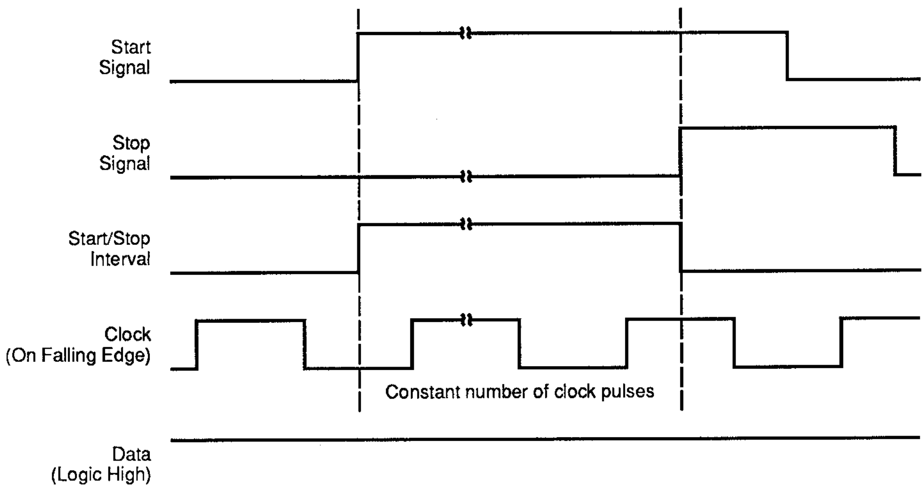


Figure 7-1: Testing for Start and Stop Stability

Unstable signatures may also be caused by Start or Stop signal edges which occur at the same time as the Clock signal edge or by Start or Stop signals which are asynchronous to the selected clock signal. Use an oscilloscope to determine whether a line is irregular or whether a timing problem exists between the Clock signal and the Start or Stop signal.

If unstable signatures are caused by Start or Stop signal edges which occur at the same time as the Clock signal edge, select the other Clock edge (+ or -) and use the *getoffset* and *setoffset* TL/1 commands to adjust the measurement timing.

Other Characterizations 7.4.3.

Some circuits are difficult to characterize by a CRC signature. The node may have regular activity but there might be no signal which can be used as a clock to gather a consistent signature. In many such cases, nodes can be characterized by using transition counts.

The transition count works on asynchronous signals. The transition count can monitor information that the CRC will not detect, such as extra transitions between CRC clocks. The transition count will typically be a range of counts, defined by a minimum and maximum, that represents the extremes of the three measurements taken by the LEARN function. Only low-to-high transitions are counted (not high-to-low). When the measurement is synchronized to the external lines, the data input is gated with the enable line, if used. A count of zero will result if the enable-true window does not overlap the low-to-high transition of the data.

The frequency of a signal may be more important than its CRC or transition count. This is especially true for system clocks. If a system clock is run at 4 MHz rather than 8 MHz, everything on the board could appear to be good. However, when the board is plugged into a system, the board running at 4 MHz may cause a system failure. Frequency is also important for video signals such as horizontal and vertical sync.

Level history is an important characterization parameter when combined with signatures or transition counts. If a faulty node has the correct timing but swings between ground and an invalid level for part of the time, measuring asynchronous level history would detect this fault, which will be missed if only a CRC is measured.

Consider the case where a node that should go high and low is stuck on a faulty UUT. Using both CRC and asynchronous level history to characterize the node will provide more complete information to the technician who repairs the board. The operator can see that the line is stuck when it should be changing.

Level history can be used to detect glitches. If the measurement period is set so that a signal is either high or low during measurement, with no glitches, the level history will show only high or low. If the level history shows both high and low, a glitch has occurred.

Calibration of the I/O Module and Probe 7.4.4.

Whenever the pod performs a microprocessor operation, it generates a synchronization pulse which the 9100A/9105A uses to measure signatures and clocked levels. The synchronization pulse can be generated by several devices, including the pod or an external clock.

In order for the system to measure critical signals reliably, each measurement device (I/O modules and probe) must be calibrated to this synchronization pulse on the system where it will be used, since each measurement device contains its own electronics that affect timing. If your tests must be accurate to within a few tens of nanoseconds on signal edges, calibration should be done.

The procedures for calibration are given in the *Technical User's Manual.* Calibration should be performed for each measurement device and for each synchronization mode of that device on the particular 9100A/9105A system where it will be used. For example, the probe for an 80286-based UUT should be calibrated to EXT, POD ADDR and POD DATA on the 9100A/9105A where the probe will be used.

Calibration is UUT-dependent. For this reason, calibration settings should be saved under the specific directory for that UUT. If calibration is not performed, default calibration values will be used. These default calibration values will only work properly in some UUTs (those which have ample timing margin or which operate at slow speeds).

Adjusting Sync Timing 7.4.5.

The sync pulse that the measurement devices (I/O modules and probe) receive from the 9100A/9105A comes either from the pod or an external clock signal. The pod may provide sync pulses with different timings relative to microprocessor read/write operations, depending on the synchronization mode of the pod. For example, the 80286 pod has POD ADDR and POD DATA sync modes. The sync pulse in POD ADDR mode is earlier than in the POD DATA mode. See the timing diagram in the pod manual for the pod you are using.

Most signals on a UUT can be characterized using the external or pod sync mode. However, in some cases, the sync pulse occurs at a different time than when the signal should be measured.

The *getoffset* and *setoffset* TL/1 commands can be used to adjust the time when a signature or clocked level measurement is made, relative to the sync pulse. Figure 7-2 shows how this offset is implemented in the probe or the I/O module. The data to be measured passes through one delay line and the sync pulse passes through a different delay line. One of the delay lines is variable. By adjusting the variable delay line, the data is measured at a different time relative to the sync pulse.

Section 3 of the *TL/1 Reference Manual* contains details about the *getoffset* and *setoffset* commands, including the approximate timing resolutions of the probe and the I/O module.

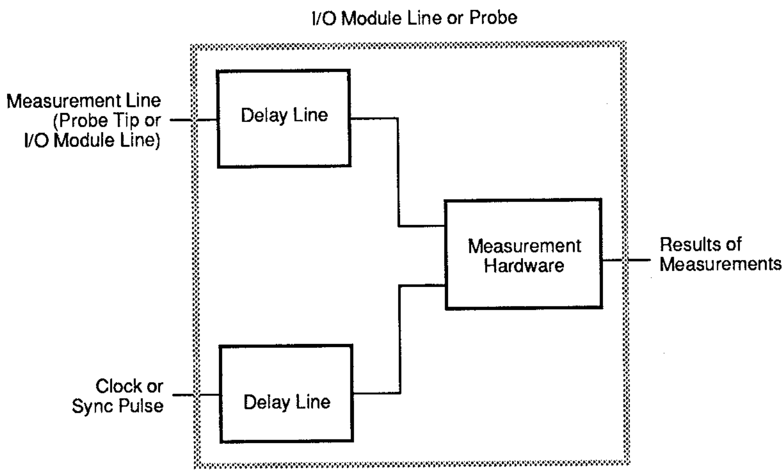


Figure 7-2: Synchronization-Pulse Delay Mechanism

Appendices C and E of the *Technical User's Manual* contain additional timing specifications for the pod, probe, and I/O modules. The *Supplemental Pod Information for 9100A/9105A Users* manual and the pod manuals have more detailed information about pods.

When a program adjusts the sync timing, the original timing should be restored at the end of the program. This can be done by storing the result of a *getoffset* command, adjusting the timing with *setoffset,* and readjusting the timing with *setoffset* at the end of the program with the stored *getoffset* value.

Dynamic RAM circuits usually require sync timing adjustment in order to measure the RAS and CAS signals, which do not necessarily coincide with the POD ADDR or POD DATA sync pulses. The Demo/Trainer UUT stimulus programs for the Dynamic RAM Timing functional block show one way to adjust the sync timing.

THE UUT DESCRIPTION 7.5.

The UUT description, which provides the 9100A/9105A with information used for GFI and UFI, consists of:

- Reference designator list (reflist).
- Part Library (part descriptions). A basic part library is provided with the system.
- Node list (net list or wire list).

The *Programmer's Manual* provides detailed information about this database and how GFI and UFI use it. The following sections are simply a brief overview.

Reference Designator List (REFLIST) 7.5.1.

The reference designator list establishes the relationship between reference designators (such as "U80") and a part or component type (such as 7410). It also specifies the testing device (probe or I/O module) to be used on the component.

A sample Demo/Trainer UUT reference designator list is shown in Appendix A. GFI and UFI both require the reference designator list to determine the device needed to test a component.

No distinction is made between families of components, such as 74LS00 or 74HCT00. The Fluke-supplied part library uses generic names like 7400 and 7432, so when you make entries in a reference designator list you will need to use generic names.

Part Library (Part Descriptions) 7.5.2.

The part library is a group of files (part descriptions) that describe UUT components. A part description specifies each pin to be an input, output, bidirectional, ground, power, or unused. Each output has a list of related inputs which affect that output. The library can be accessed through any UUT directory. A basic part library is supplied by Fluke. You can add part descriptions, including custom designs.

See the Guided Fault Isolation section of the *Programmer's Manual* for examples of part descriptions.

Node List (Net List or Wire List) 7.5.3.

The node list specifies interconnections between reference designators. The list is only necessary for GFI, which uses it to backtrace between components.

A complete node list contains one line for each node in a UUT. The pins on one line are all connected to form a node. Lines may be continued on the next line with the backslash (\) character.

Appendix B contains a node list for the Demo/Trainer UUT. Reviewing this example will be helpful to you when developing you own node lists.

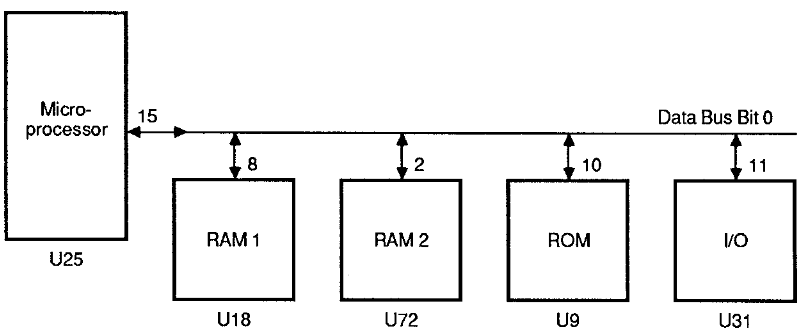
Bus-Master Pins in a Node List 7.5.4.

The 9100A normally determines the flow of data from the node list; it assumes that data can be sent from any pin to any other pin on a given node. However, sometimes two pins are connected together by a node but do not actually communicate with each other; this situation commonly arises in bus-oriented systems with many components connected to a common microprocessor data bus.

In such cases, you need to let GFI know that only some pins (called bus-master pins) can communicate with all the other pins on the same node. This is done by entries in the optional **masters* section of the node list.

The **masters* section is optional, and for most UUT node lists it can be omitted. Where it is needed, it usually contains just a short list of pins, because most nodes have only a single source. It is only for nets such as the one in the following example that the **masters* section becomes important.

Consider the node shown below: It consists of bit 0 of a bidirectional data bus connecting several components to a microprocessor.



Only pin U25-15 can talk to all other input pins on the node and only U25-15 can receive from all other output pins on the node. Either condition would be sufficient to make U25-15 a bus-master pin.

For this reason, pin U25-15 is shown as a bus-master pin in the partial node list below. It is listed in the regular section of the node list and is also included in the optional **masters* section of the node list.

```
.
.
.
U8-12 U3-9 U42-21
U25-15 U19-8 U22-2 U9-10 U31-11
U17-4 U28-5 U27-6
.
.
.
*masters
U25-15
```

See the Node List section in the *Programmer's Manual* for more information about bus-master pins.

Choice of Backtracing Path 7.5.5.

If there are two or more stimulus programs available for a node, GFI will attempt to use the program that stimulates all of the node's outputs (and related inputs) before using programs that stimulate only some of the node's pins.

Here are three cases that relate to the AND gate in Figure 7-3. Each case shows the test results from two stimulus programs, A and B, and the conclusion that GFI comes to:

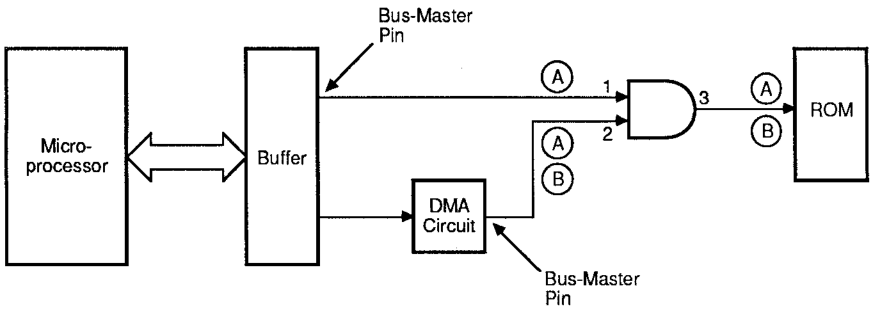


Figure 7-3: Direction-Control Example

Case 1:

	Input 1	*Input 2*	*Output 3*
Stimulus Program A	good	good	bad
Stimulus Program B	–	bad	bad

GFI will accuse the node of being bad because stimulus program A covers all the nodes and is therefore evaluated first. In this case stimulus program B will not be executed.

Case 2:

	Input 1	*Input 2*	*Output 3*
Stimulus Program A	bad	good	bad
Stimulus Program B	–	bad	bad

GFI will test the component connected to input 1, again because stimulus program A covers all the nodes and is therefore evaluated first Therefore, GFI will backtrace to the Bus Buffer.

Case 3:

	Input 1	*Input 2*	*Output 3*
Stimulus Program A	good	good	good
Stimulus Program B	–	bad	bad

GFI will test the component connected to input 2, because stimulus program A finds no problem and the system goes on to evaluate stimulus program B. Therefore, GFI will backtrace to the DMA circuit.

Consider these two problems in Figure 7-3, in which both the microprocessor and the DMA controller are both **master* components:

- If the problem is in the microprocessor, evaluation is the same as for Case 2, above, and GFI troubleshooting traces back to the microprocessor from input 1 of the AND gate.
- If the problem is in the DMA controller, evaluation is the same as for Case 3, above, and GFI troubleshooting traces back to the DMA circuit from input 2 of the AND gate.

While you can effectively steer GFI by designing stimulus programs to cover *all* or only *some* inputs and outputs, you do not usually need to worry about control of the backtracing path; it is only needed in special circumstances.

Normally, you should design stimulus programs that test *all* inputs and outputs of a node or component. If there is no single stimulus program that covers all inputs and outputs, the 9100A/9105A uses these criteria to determine status:

- If ANY stimulus program gives a BAD response on a pin, the pin is considered BAD.
- If ALL stimulus programs give GOOD responses on the pin, the pin is considered GOOD.
- Otherwise, the pin is considered UNKNOWN.

SUMMARY OF GFI COVERAGE 7.6.

The 9100A provides a convenient means to check the completeness of the information you have entered into the GFI database for a particular UUT. When viewing the UUT directory display, you can press the SUMMARY softkey to request generation of a summary of GFI coverage for that particular UUT. The compiled database (GFIDATA or UFIDATA) will be examined and a summary will be generated, displayed on the monitor, and stored in a UUT text file that you specify. If you press the Shift key on the programmer's keyboard and the SUMMARY softkey, the summary will appear on the monitor without sending a copy to a text file.

Creating a Summary of GFI Coverage

The following procedure is used to generate a Summary of GFI Coverage for a UUT:

1. Press the EDIT key on the operator's keypad to enter the Editor (unless you are already in the Editor).

2. Use the EDIT key on the Programmer's Keyboard to enter the name of the UUT so that the UUT directory for this UUT is displayed on the monitor. The UUT directory you have selected must contain a compiled database (either GFIDATA or UFIDATA).

3. Press the SUMMARY Softkey (F8) and the 9100A will issue the prompt shown below to ask for a text file name:

   ```
   Generate GFI Summary to TEXT file ________
   ```

   The Summary of GFI Coverage to be generated will be stored in this text file.

4. Type in the text file name you wish and press the Return key. The 9100A will then begin generating the Summary of GFI Coverage for the UUT and will display the results on the monitor.

When the generation is complete, the following message will appear on the monitor:

```
Press Msgs key to continue
```

When you press the Msgs key on the programmer's keyboard, the UUT directory display will reappear on the monitor. You can use the Edit key on the programmer's keyboard to access the text file you generated.

Statistical Summary

The first part of the Summary of GFI Coverage is a statistical summary of the UUT, based on the GFI database you have provided. Figure 7-4 shows a typical example of such a summary. Each entry in the summary is described below:

- **Summary for /<disk drive>/<UUT>:** In Figure 7-4, HDR is the disk drive and the UUT directory name is EXAMPLE.
- **Parts:** The number of unique part types in the UUT, based on the reference designator list.
- **Reference Designators**: The number of reference designators in the UUT, based on the node list.
- **Connected Pins:** The number of UUT pins that are connected to other pins on the UUT, based on the node list.
- **Unconnected Pins:** The number of UUT pins that are not connected to any other UUT pins, based on the node list.
- **Total Pins:** The total number of pins on the UUT.
- **Programs:** The number of TL/1 programs that can be used by GFI as stimulus programs. This number is equal to the number of response files.
- **Testable Connected Pins:** The number of connected pins that can be tested by GFI. Testable pins have either been characterized with LEARN, or are a member of a node that has been characterized with LEARN.
- **Testable Unconnected Pins:** The number of unconnected pins that can be tested by GFI. Testable unconnected pins have been characterized by LEARN and appear in a response file.
- **Total Testable Pins:** The total number of UUT pins that can be tested with GFI, given the database you have entered.

```
Summary for /HDR/EXAMPLE:

    53  Parts
   167  Reference Designators
  1694  Connected Pins
   225  Unconnected Pins
  1919  Total Pins
    42  Programs

  1688  Testable Connected Pins
    16  Testable Unconnected Pins
  1704  Total Testable Pins

     6  Untestable Connected Pins
   209  Untestable Unconnected Pins
   215  Total Untestable Pins

    99% Test Coverage of Connected Pins
    88% Test Coverage of Total Pins
```

Figure 7-4: Statistical Summary Display for a UUT

- **Untestable Connected Pins:** The number of connected pins that cannot be tested with GFI, due to an incomplete database.
- **Untestable Unconnected Pins:** The number of unconnected pins that cannot be tested with GFI, due to an incomplete database.
- **Total Untestable Pins:** The total number of UUT pins that cannot be tested with GFI, given the database you have entered.
- **Test Coverage of Connected Pins:** The percentage of connected pins on the UUT that can be tested with GFI, given the database you have entered. A figure of less than 100% indicates an incomplete database.
- **Test Coverage of Total Pins:** The percentage of UUT pins that can be tested with GFI, given the database you have entered. This figure is typically less than 100% because a UUT often has unused pins.

Pin Coverage

The second part of the GFI Summary of Coverage display is a matrix showing how component pins are tested with the database you have provided. Figure 7-5 shows a partial example of a pin coverage matrix. The matrix is organized with the reference designators listed vertically (in the left-most column) and with component pin numbers listed horizontally. The number of pins per line will be the number required by the largest component in the list. If more than 35 pins are required, the display will produce a second list of reference designators following the first list and this second set will have pin numbers starting with 36 and continuing up from there.

Each component pin has a one-character symbol that shows how GFI looks at the pin given the database you have provided. The table at the bottom of Figure 7-5 shows the meaning of each symbol that is possible:

```
Pin Coverage:

                              1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
            1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

     C15    I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     C16    I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     C17    I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
      J5    I * * I I * * * I * * I I I I I I I I I I I I I I I I I I I I I I I I
      J6    I I I I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
      Q1    O I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
      Q2    O I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     R10    O I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     R11    I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     R12    I O . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
      S1    I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
      S2    I I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
     U10    I I I I I B I I B I I B I I B B I I B I . . . . . . . . . . . . . . .
     U11    * I * I I I I * I I I * O O O * O B B B B I * O B B B B * * * * O * I
     U12    O O I O I O O I O I I O I O I I . . . . . . . . . . . . . . . . . . .
     U13    I O I O I O G O I O I O I P . . . . . . . . . . . . . . . . . . . . .
     U14    O * * O O * * * I * * O O O O O O O O O O O O O O O O O I I I O O O I
```

Symbol	*Meaning*
I	The pin is testable as an input only.
O	The pin is testable as an output only.
B	The pin is testable as both an input and an output.
P	The pin is testable as a power pin.
G	The pin is testable as a ground pin.
*	The pin is not testable (because it has no associated stimulus program or no known-good response stored for this pin).
.	There is no such pin in the database.

Figure 7-5: Pin Coverage Display for a UUT

FAULT CONDITION EXERCISERS 7.7.

When the 9100A/9105A detects a fault, and a fault condition handler is not defined for the fault condition raised, a fault message will appear on the operator's display. At this point, the operator can press the LOOP key on the operator's keypad to repeatedly reproduce the fault so that it can be isolated manually. To do so requires that a fault condition exerciser exist for the fault condition that was raised. If the exerciser exists, it is invoked continually until the operator presses the STOP key on the operator's keypad.

A fault condition exerciser is a software block designed specifically to reproduce a fault condition in a UUT. Two types of exercisers are available: built-in exercisers and user-defined exercisers.

When a fault condition is raised by a built-in stimulus function (such as read, write, ramp, toggle, or rotate) or a built-in test function (such as *testbus, testramfast, testramfull,* or *testromfull*), the 9100A/9105A has a pre-defined sequence of commands that exercise the fault when the LOOP key is pressed. These are called built-in fault condition exercisers. In addition, you as a programmer can write your own fault condition exercisers for fault conditions that you define or to replace the built-in fault condition exercisers. When one of these fault conditions is encountered and the LOOP key is pressed, the fault condition exerciser with the matching name is invoked.

If a fault condition exerciser for the displayed fault condition is found when the LOOP key is pressed, the fault condition exerciser is invoked repeatedly to stimulate the UUT. This allows the probe to be used to examine node responses in the circuit and to trace faulty circuit operation to its cause.

REPAIR AFTER TROUBLESHOOTING 7.8.

When GFI terminates, it will often display one of the following messages:

- Open circuit.
- Bad IC or output loaded.

When GFI reports an *open circuit*, it has found an input which is bad even though the signal source on that node is good. To repair the node:

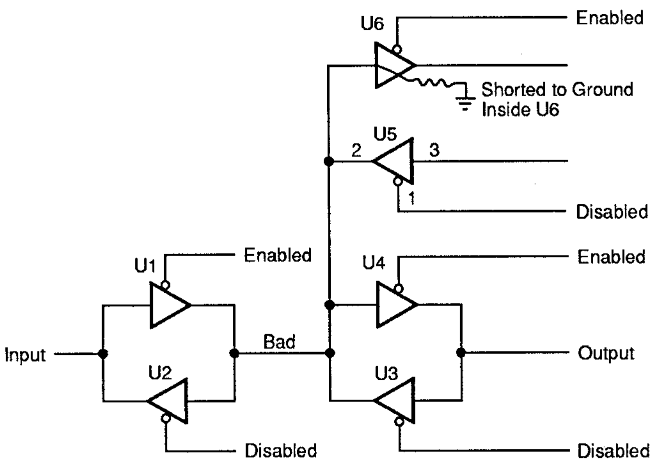
1. Retest both ends of the node to make sure the output was properly probed.
2. Confirm the open circuit with an ohmmeter.
3. Trace along the node with the ohmmeter until the open point is found.
4. If the node is connected properly, check for:
   - An error in the node list entry for the failed node.
   - Marginal measurements due to the frequency or timing of signals on the node. Ringing may be occurring on the node, or the time between the sync and the signal transitions may be marginal. Change the stimulus setup or the sync timing to correct the problem (see Section 8.5 on adjusting sync timing).

When GFI reports a *bad IC* or *output loaded*, it has found all good inputs and one or more bad outputs. In this case, determine whether the part is bad or the output is loaded. To do this, test the component by overdriving its inputs with the I/O module while measuring level history or CRC signatures.

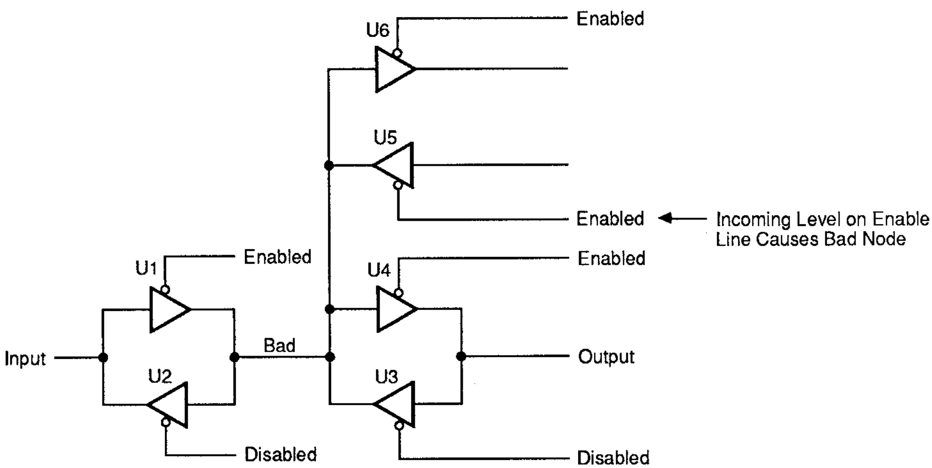
In doing so, determine whether:

- The level history showed that the line went to a high and low state. If so, the node is only loaded part of the time, or the component is bad.
- The node is loaded. If the component is good but the node is bad, the node must be loaded. The cause of a loaded node can be:
    - A short to another node, the power supply, or ground.
    - A damaged IC loading the node. Example 1 in Figure 7-6 shows a bad input at U6 causing node A to be loaded.
    - Another output source is also driving that node. Check the enable and control lines of any other devices that can drive the node. Example 2 in Figure 7-6 shows node A to be loaded because both U1 and U5 are attempting to drive the node at the same time. U1 is operating as it should but the U5 enable-line state is incorrect and U5 is also driving the same node.

Operators should be provided with a procedure for tracing short circuits. For example, a milliohmmeter can be used to determine the point at which a node is shorted. To do this, attach one lead of the milliohmmeter to the faulty node. With the other lead, look for low resistance paths.



Example 1: Bad IC



Example 2: Bus Contention

Figure 7-6: Node Loading

Section 8
Glossary

If you cannot find a term in the glossary, search the index for a reference to that term.

Active Edge
A signal transition used to initiate action.

Address Space
A section of memory reserved for a particular use, such as the stack. The term "decoded address space" implies memory residing in physically separate chips (selectively enabled by a "decoder"), such as a frame buffer, character generator, or the control registers inside a peripheral chip.

Aliasing
A condition where a component address responds to more than one combination of address bus bits.

Assert
To cause a signal to change to its logical "true" state.

Asynchronous
Not synchronized to the microprocessor or not synchronous to any clock signal.

Automated Test
An automated activity that verifies the correct operation of a circuit by comparing its output to the expected output.

Automated Troubleshooting
An automated process of locating a fault on a UUT.

Backtracing
A procedure for locating the source of a fault on a UUT by checking logic along a logical path from bad outputs to bad inputs until the point where no bad inputs are found.

Bus
A group of functionally similar signals.

Bus Contention
A situation where two or more bus devices are trying to put different data onto the same bus.

CAD
An acronym for Computer-Aided Design. CAD systems let the user create, manipulate, and store designs on a computer.

Comment
Text in a program that is not executed. A comment in a TL/1 program or a node list must begin with an exclamation point (!).

Component
A passive or active part on a UUT.

Control Line
A signal that comes out of a microprocessor and is used to control the UUT.

CRC Signature
CRC is an acronym for Cyclic Redundancy Check. A CRC signature is a compression of a long data stream into a 16-bit number.

Cursor
A symbol on a display (usually a box or an underscore) that indicates where a typed character will appear.

Data Bus
A set of signal paths on which parallel data is transferred between two or more devices.

Device
1. Refers to the probe, an I/O module, a reference designator, or the pod. 2. Also used with I/O operations to specify a port or a disk drive.

DIP
An acronym for Dual In-line Package. A DIP has an equal number of pins on each of its long sides. See also SIP.

Directory
A collection of related sets of data (files, for example) on a disk.

Drivability
Testing whether lines can be driven to the appropriate active high or active low level.

Dynamic Coupling
Data in one memory location is affected by combinations of data in other memory locations.

Edge
The transition from one voltage level to a different voltage level.

Exerciser
See Fault Condition Exerciser.

External Synchronization
Synchronizing a node response measurement using signals external to the pod.

Fault
A defect in a UUT that causes circuitry to operate in a manner that is inconsistent with its design.

Fault Condition
A recognition by the 9100A/9105A that a fault exists on the UUT.

Fault Condition Exerciser
A group of statements that attempts to repetitively reproduce the conditions that generate a fault condition. (Sometimes called just an "exerciser.")

Fault Condition Handler
A group of statements that is executed when a particular fault condition occurs. (Sometimes called just a "handler.")

Fault Condition Raising
The generation of a fault condition either from detecting a fault on a UUT or from using a TL/1 *fault* statement.

Feedback Loop
A circuit in which one or more outputs is routed to the circuit's input.

Forcing Line
Input to the microprocessor that forces it to a particular known state.

Functional Test
An activity that verifies the correct operation of a circuit by comparing its output to the expected output.

GFI
See Guided Fault Isolation.

GFI Summary
A record of the components that have been tested by GFI.

Go/No-Go Test
A pass/fail test; either a unit passes or it doesn't.

Guided Fault Isolation
An algorithm that uses backtracing to troubleshoot a UUT.

Handler
See Fault Condition Handler.

Hexadecimal
Pertaining to the base 16 numbering system. (Often abbreviated as "hex.")

I/O
An abbreviation for Input/Output. The transfer of data to and from devices other than the local memory of the microprocessor system.

I/O Module
An option for the 9100A/9105A that allows simultaneous stimulus or response for multiple points on a UUT.

Level History
A character string that represents a record of the logic levels measured at a point over a period of time. "1", "X", and "0" represent high, invalid, and low states, respectively.

Library
A directory that contains a collection of only a particular type of file. The 9100A/9105A uses four libraries: a part library, a program library, a pod library, and a help library.

Mask
A value where each logic "1" represents a bit that is to be acted on.

Monitor
A 24-line, 80-column video display that connects to the rear panel of the 9100A/9105A.

Node
A set of points that are all electrically interconnected.

Node List
A file containing a description of the interconnection of all pins on a UUT.

Operator
1. A symbol that acts on one or more values or expressions to produce another value. 2. A person who uses the 9100A/9105A for testing or troubleshooting.

Operator's Display
Three-line display on the mainframe of the 9100A/9105A.

Operator's Interface
The operator's display and the operator's keypad.

Operator's Keypad
The set of keys on the front panel of the base unit of the 9100A/9105A.

Overdrive
To put a logic state on a signal line by applying more power than the normal driver for that node. This is how the 9100A/9105A injects signals into the UUT.

Part Description
A file that describes a component on a UUT.

Part Library
A library of part descriptions.

Pod Library
A library of pod descriptions, each of which contains a pod database and pod-related TL/1 programs.

Pod Synchronization
Synchronizing a node response measurement using signals generated by the pod to indicate the sampling time.

Priority Pin
A pin that the GFI program will test first if a particular node is bad.

Probe
A hand-held device that can stimulate and measure any single point on the UUT.

Program Library
A library of programs that can be called by any program in the userdisk.

Programmer's Interface
The monitor and the programmer's keyboard.

Programmer's Keyboard
The keyboard that connects to the side panel of the 9100A.

Raise
See Fault Condition Raising.

Reference Designator
A one to ten character string naming a component on the UUT.

Related Input Pin
An input pin on a component that affects an output pin on that same component.

Response File
A file containing data generated by executing a specific stimulus program to a UUT and recording the responses from its execution.

RUN UUT Test
A feature that allows the normal operation of a UUT using its own program.

Signature
See CRC Signature.

SIP
An acronym for Single In-line Package. See also DIP.

Softkey
A key that has its function determined by software.

State Machine
A circuit which produces output signals in response to input signals and its own internal state. Typically used to generate a sequence of control signals, as in a bus interface.

Stimulus Program
A program that exercises a circuit while the activity on circuit nodes are recorded to see if the circuit produces the same response as a known-good circuit.

String
A group of characters enclosed in double-quote characters (") and manipulated as a single entity.

Synchronous
Coordinated to the transitions of a clock signal.

Termination Status
An indication of whether a UUT passed a test.

Timeout
A condition in which an expected event has not occurred within the expected time period.

Toggle
Change to the complementary logic state.

Transition Count
A record of the number of times the logic level at a node changes from low to high within a period of time.

Troubleshooting
A process of locating the area of a UUT that is causing a fault.

Userdisk
1. A diskette containing test programs and information about a particular UUT. 2. The current disk drive that is used as a source for UUT programs and data.

UUT
Unit Under Test. A physical item, i.e., a board or a system to be tested.

UUT Directory
A set of files that contain information about a particular UUT.

Wait State
A bus cycle which is too short for a slow chip is lengthened by the insertion of one or more clock cycles, called wait states.

Watchdog Timer
A circuit which produces a signal, typically a reset or high-priority interrupt, if a timeout condition is met. For example, an excessive number of wait states may trigger a watchdog timer.

Wildcard
A symbol that represents any sequence of characters. The 9100A/9105A uses the asterisk character (*) for this purpose.

Window
An area of the monitor reserved for certain information to be displayed.

(This page is intentionally blank.)

Appendix A
Demo/Trainer UUT Reflist

NAME: REFLIST
DESCRIPTION:

SIZE: 3,555 BYTES

REF	PART	TESTING DEVICE
R72	RESISTOR	PROBE
R73	RESISTOR	PROBE
R4	RESISTOR	PROBE
R79	RESISTOR	PROBE
R78	RESISTOR	PROBE
R61	RESISTOR	PROBE
R62	RESISTOR	PROBE
R63	RESISTOR	PROBE
R64	RESISTOR	PROBE
R65	RESISTOR	PROBE
R70	RESISTOR	PROBE
C4	CAPACITOR	PROBE
C5	CAPACITOR	PROBE
C8	CAPACITOR	PROBE
C9	CAPACITOR	PROBE
C13	CAPACITOR	PROBE
C15	CAPACITOR	PROBE
C16	CAPACITOR	PROBE
C17	CAPACITOR	PROBE
U74	2016	I/O MODULE
U85	2016	I/O MODULE
U72	2674	I/O MODULE
U78	2675	PROBE
U11	2681	PROBE
U77	27128	I/O MODULE
U30	27256	I/O MODULE

U29	27256	I/O MODULE
U28	27256	I/O MODULE
U27	27256	I/O MODULE
Q1	TRANSISTOR	PROBE
Q2	TRANSISTOR	PROBE
C1	CAPACITOR	PROBE
R35	RESISTOR	PROBE
R1	RESISTOR	PROBE
R77	RESISTOR	PROBE
R80	RESISTOR	PROBE
R15	RESISTOR	PROBE
R14	RESISTOR	PROBE
R16	RESISTOR	PROBE
R13	RESISTOR	PROBE
R17	RESISTOR	PROBE
R12	RESISTOR	PROBE
R18	RESISTOR	PROBE
R11	RESISTOR	PROBE
R27	RESISTOR	PROBE
R25	RESISTOR	PROBE
R24	RESISTOR	PROBE
R28	RESISTOR	PROBE
R29	RESISTOR	PROBE
R23	RESISTOR	PROBE
R30	RESISTOR	PROBE
R19	RESISTOR	PROBE
R68	RESISTOR	PROBE
R69	RESISTOR	PROBE
R20	RESISTOR	PROBE
R21	RESISTOR	PROBE
R22	RESISTOR	PROBE
R34	RESISTOR	PROBE
R33	RESISTOR	PROBE
R3	RESISTOR	PROBE
R5	RESISTOR	PROBE
R6	RESISTOR	PROBE
R7	RESISTOR	PROBE
R8	RESISTOR	PROBE
R32	RESISTOR	PROBE
R31	RESISTOR	PROBE
R26	RESISTOR	PROBE
R9	RESISTOR	PROBE
R2	RESISTOR	PROBE
U34	4164	I/O MODULE
U35	4164	I/O MODULE
U36	4164	I/O MODULE
U37	4164	I/O MODULE
U38	4164	I/O MODULE
U39	4164	I/O MODULE
U40	4164	I/O MODULE

U41	4164	I/O MODULE
U48	4164	I/O MODULE
U49	4164	I/O MODULE
U50	4164	I/O MODULE
U51	4164	I/O MODULE
U52	4164	I/O MODULE
U53	4164	I/O MODULE
U54	4164	I/O MODULE
U55	4164	I/O MODULE
R67	RESISTOR	PROBE
C6	CAPACITOR	PROBE
C7	CAPACITOR	PROBE
R71	RESISTOR	PROBE
R10	RESISTOR	PROBE
R66	RESISTOR	PROBE
U14	80286	PROBE
J5	CONN68	PROBE
U1	82284	I/O MODULE
U15	82288	I/O MODULE
U31	8255	I/O MODULE
U58	7400	I/O MODULE
U24	7400	I/O MODULE
U5	7400	I/O MODULE
U64	7402	I/O MODULE
U57	7404	I/O MODULE
U19	7404	I/O MODULE
U4	7408	I/O MODULE
U63	7408	I/O MODULE
U56	7410	I/O MODULE
U21	74138	I/O MODULE
U8	74138	I/O MODULE
U9	74138	I/O MODULE
U3	74245	I/O MODULE
U23	74245	I/O MODULE
U44	7474	I/O MODULE
CR1	DIODE	PROBE
J2	CONN_RS232	PROBE
J3	CONN_VIDEO	PROBE
J6	CONN_KEYBD	PROBE
U73	74157	I/O MODULE
U83	74157	I/O MODULE
U84	74157	I/O MODULE
U65	74257	I/O MODULE
U66	74257	I/O MODULE
U33	7SEGLED	PROBE
U47	7SEGLED	PROBE
U61	7400	I/O MODULE
U70	7400	I/O MODULE
U71	7400	I/O MODULE
U62	7404	I/O MODULE

U59	74109	I/O MODULE
U80	7410	PROBE
U81	7410	PROBE
U7	74112	I/O MODULE
U25	74112	PROBE
U26	74125	I/O MODULE
U20	74148	I/O MODULE
U13	7414	I/O MODULE
U43	74164	I/O MODULE
U17	74164	I/O MODULE
U75	74175	I/O MODULE
U68	74244	I/O MODULE
U69	74244	I/O MODULE
U32	74244	I/O MODULE
U46	74244	I/O MODULE
U6	7430	I/O MODULE
U79	7430	I/O MODULE
U60	7431	I/O MODULE
U45	7432	I/O MODULE
U86	74373	I/O MODULE
U87	74373	I/O MODULE
U10	74373	I/O MODULE
U2	74374	I/O MODULE
U16	74374	I/O MODULE
U22	74374	I/O MODULE
U76	74374	I/O MODULE
U42	74390	I/O MODULE
U67	74590	I/O MODULE
U12	MAX232	PROBE
J4	PWRCONN	PROBE
U18	OSCILLATOR	PROBE
U82	74175	PROBE
U88	7486	PROBE
Y1	XTAL	PROBE
S4	KEYSWITCH	PROBE
S3	KEYSWITCH	PROBE
S2	KEYSWITCH	PROBE
S1	KEYSWITCH	PROBE
S6	KEYSWITCH	PROBE
DS1	LED	PROBE
Z1	NETWORK10	I/O MODULE

Appendix B
Demo/Trainer UUT Node List

NAME: NODELIST
DESCRIPTION: SIZE: 16,492 BYTES

```
U23-11  U41-2  U69-17  U30-11  U28-11  U41-14  Z1-10
U23-12  U40-2  U40-14  U69-15  U30-12  U28-12  Z1-9
U23-13  U39-2  U39-14  U69-13  U30-13  U28-13  Z1-8
U23-14  U38-2  U38-14  U69-11  U30-15  U28-15  Z1-7
U23-15  U37-2  U37-14  U69-8  U30-16  U28-16  Z1-6
U23-16  U36-2  U36-14  U69-6  U30-17  U28-17  Z1-5
U23-17  U35-2  U35-14  U69-4  U30-18  U28-18  Z1-4
U23-18  U34-2  U34-14  U69-2  U30-19  U28-19  Z1-3
U58-8 U34-15  U35-15  U36-15  U37-15  U38-15  U39-15  U40-15  U41-15
R69-1  R72-1  U88-8
U84-6  U72-32
R14-1  U46-12
R13-1  U46-14
R12-1  U46-16
R11-1  U46-18
R18-1  U46-3
R17-1  U46-5
R16-1  U46-7
R15-1  U46-9
U32-11 U31-40
R27-1  U32-9
R25-1  U32-12
R24-1  U32-14
R23-1  U32-16
R19-1  U32-18
R30-1  U32-3
U32-13  U31-39
R28-1  U32-7
U32-15  U31-38
```

```
R29-1  U32-5
U32-8  U31-1
R27-2  U33-7
R19-2  U33-1
U2-5  U66-6  U21-2  U30-26  U29-26  U28-26  U27-26
U84-3  U72-31
U84-10  U72-33
U16-15  U65-5  U84-11  U11-4  U72-38  U31-8  U30-9  U29-9  \
  U28-9  U27-9
U11-36  Y1-1   C8-1
U11-37  Y1-2   C9-1
U16-19  U61-9  U21-4  U62-9  U62-11
U70-11  U81-5
U22-9  U61-10  U57-13  U62-13
U65-1  U66-1  U60-7
U3-18  U48-2  U48-14  U68-2  U10-19  U11-21  U72-15  U31-27  \
  U29-19  U27-19
U2-6  U66-3  U21-1  U30-2  U29-2  U28-2  U27-2
R25-2  U33-8
R24-2  U33-10
R23-2  U33-13
R29-2  U33-11
U32-6  U31-2
U32-2  U31-4
U32-4  U31-3
U46-11  U31-22
R12-2  U47-13
R17-2  U47-11
R13-2  U47-10
R14-2  U47-8
R15-2  U47-7
R11-2  U47-1
U58-2  U8-14
U61-1  U62-12
U61-4  U62-10
U43-11  U61-12  U67-11  U67-13  U44-1  U44-13  U59-13
U61-6  U68-1  U68-19  U74-21
U61-3  U69-1  U69-19  U85-21
U70-3  U71-2
U70-6  U71-4
U70-8  U71-5
U56-10  U21-15  U72-2
U75-5  U83-10  U72-29
U68-3  U74-9  U77-6
U68-5  U74-10  U77-5
U68-14  U74-15  U77-24
U76-6  U78-4
U76-5  U78-36
U88-9  U78-29
U87-13  U77-16
```

```
U87-8  U77-15
U87-17  U77-18
U87-7  U77-13
U87-18  U77-19
U75-2  U77-10
U75-7  U77-9
U75-10  U77-8
U86-19  U78-19
U87-14  U77-17
U87-3  U77-11
U87-4  U77-12
U86-15  U78-17
U86-16  U78-16
U86-12  U78-25
U86-6  U78-18
U69-9  U86-13  U85-13
U69-18  U85-17  U77-26
U72-23  U78-11
U69-12  U86-8  U85-14
U80-8  U81-13
U80-10  U81-10  U82-6
U80-12  U81-1
U80-2  U70-5  U71-12  U81-4  U82-15
U80-4  U70-9  U81-11  U82-10
U80-6  U81-2
U80-11  U79-11  U82-14
U80-5  U79-4  U62-2
U80-3  U70-13  U79-5  U81-9  U82-3  U73-1  U83-1  U84-1  U62-5
U70-12  U76-11  U79-3  U86-11  U87-11  U72-16  U78-33
U71-13  U79-6  U81-3  U82-11
U22-5  U21-6
U83-9  U74-3  U85-3
U3-11  U55-2  U55-14  U68-17  U10-2  U11-28  U72-8  U31-34  \
  U29-11  U27-11
U32-17  U31-37
U2-15  U65-6  U73-11  U30-24  U29-24  U28-24  U27-24
U16-5  U66-5  U83-11  U30-5  U29-5  U28-5  U27-5
U46-13  U31-23
U46-8  U31-21
U2-9  U65-13  U30-23  U29-23  U28-23  U27-23
U27-22 U6-5  U45-3   U28-22
U34-4  U35-4  U36-4  U37-4  U38-4  U39-4  U40-4  U41-4  U48-4  \
  U49-4  U50-4  U51-4  U52-4  U53-4  U54-4  U55-4 U64-8  U63-8
U34-5  U35-5  U36-5  U65-4  U67-15 U37-5  U38-5  U39-5  U40-5   \
  U41-5  U48-5  U49-5  U50-5  U51-5  U52-5  U53-5 U54-5  U55-5
U34-6  U35-6  U36-6  U65-9  U67-2 U37-6  U38-6  U39-6  U40-6  \
  U41-6  U48-6  U49-6  U50-6  U51-6  U52-6  U53-6  U54-6  U55-6
U34-7  U35-7  U36-7  U65-7  U67-1  U37-7  U38-7  U39-7  U40-7  \
  U41-7  U48-7  U49-7  U50-7  U51-7  U52-7  U53-7  U54-7  U55-7
U34-3  U35-3  U36-3  U37-3  U38-3  U39-3  U40-3  U41-3  U48-3  \
```

```
 U49-3  U50-3  U51-3  U52-3  U53-3  U54-3  U26-8  U55-3
U34-12  U35-12  U36-12  U65-12  U67-3  U37-12  U38-12  U39-12  \
  U40-12  U41-12  U48-12  U49-12  U50-12  U51-12  U52-12  U53-12 \
  U54-12  U55-12
U34-11  U35-11  U36-11  U66-4  U67-4  U37-11  U38-11  U39-11  \
  U40-11 U41-11  U48-11  U49-11  U50-11  U51-11  U52-11  U53-11 \
  U54-11  U55-11
U34-10  U35-10  U36-10  U66-7  U67-5  U37-10  U38-10  U39-10  \
  U40-10  U41-10  U48-10  U49-10  U50-10  U51-10  U52-10  U53-10 \
  U54-10  U55-10
U34-13  U35-13  U36-13  U66-9  U67-6 \
  U37-13  U38-13  U39-13  U40-13  U41-13  U48-13  U49-13  U50-13   \
  U51-13  U52-13  U53-13  U54-13  U55-13
U58-11  U48-15  U49-15  U50-15  U51-15  U52-15  U53-15  U54-15   \
  U55-15
U6-2  U8-11
U6-3  U8-10
U5-10  U11-9  U72-3  U31-36  U15-11
U57-2  U5-13
U83-12  U74-4  U85-4
U6-11  U81-8
U16-2  U66-11  U83-5  U30-4  U29-4  U28-4  U27-4
U43-9  U56-12
U2-12  U65-10  U73-5  U30-21  U29-21  U28-21  U27-21
U56-1  U44-9  U64-12
U34-9  U35-9  U36-9  U66-12  U67-7  U37-9  U38-9  U39-9  U40-9  \
  U41-9  U48-9  U49-9  U50-9  U51-9  U52-9  U53-9 U54-9  U55-9
U56-9  U21-14  U11-39
U46-15  U31-24
U46-17  U31-25
U76-15  U78-38
U75-15  U77-7
U69-14  U86-7  U85-15
U45-1  U45-4  U56-3  U79-2  U57-1  U15-8
U45-5   U9-9  U30-20  U29-20
U2-16  U65-3  U73-14  U30-25  U29-25  U28-25  U27-25
U2-19  U66-14   U83-2  U30-3  U29-3  U28-3  U27-3
U46-6  U31-20
U46-4  U31-19
U46-2  U31-18
U16-6  U66-2 U83-14  U30-6  U29-6  U28-6  U27-6
U3-14  U52-2  U52-14  U68-11  U10-9  U11-19  U72-11  U31-31   \
  U29-15  U27-15
U3-13  U53-2  U53-14  U68-13  U10-6  U11-27  U72-10  U31-32   \
  U29-13  U27-13
U88-4  U78-28
U82-1  U13-10
U3-15  U51-2  U51-14  U68-8  U10-12  U11-26  U72-12  U31-30   \
  U29-16  U27-16
U22-4  J5-66  U14-66
```

```
U22-14  J5-13  U14-13
U22-18  J5-15  U14-15
U2-4  J5-17  U14-17
U22-13  J5-12  U14-12
U22-17  J5-14  U14-14
U14-52  C4-1
J5-52   C13-1
U16-8   J5-27  U14-27
U16-7   J5-26  U14-26
U16-13  J5-28  U14-28
U1-10  U17-8  U44-3  U44-11  U59-4  J5-31   \
  U7-1  U13-1  U14-31  U15-2
U16-14  J5-32  U14-32
U16-18  J5-34  U14-34
R1-2  U1-4  U19-1  J5-63  U4-12  U14-63  U15-1
U23-2  J5-51  U14-51
U23-3  J5-49  U14-49
U3-2  J5-50  U14-50
U3-6  J5-42  U14-42
U23-4  J5-47  U14-47
U23-5  J5-45  U14-45
U23-6  J5-43  U14-43
U23-8  J5-39  U14-39
U2-11  U16-11  U22-11  U7-2  U15-5
U56-5  U11-10  U72-1  U31-5  U15-12
U16-16  U65-2  U84-14  U11-2  U72-37  U31-9  U30-10  U29-10   \
  U28-10  U27-10
U23-9  J5-37  U14-37
U26-1  U13-4  U13-13  U14-64
U3-5  J5-44  U14-44
U22-8  J5-1  U14-1
U23-7  J5-41  U14-41
U3-9  J5-36  U14-36
J5-64  U13-12
U3-8  J5-38  U14-38
U3-7  J5-40  U14-40
U2-8  J5-19  U14-19
U2-2  U66-10  U21-3  U30-27  U29-27  U28-27  U27-27
U3-3  J5-48  U14-48
U2-18  J5-23  U14-23
U3-4   J5-46  U14-46
U84-12  U74-8  U85-8
U84-9  U74-7  U85-7
U1-16  J5-4  U14-4  U15-3
R26-1  U13-3
U16-12  U65-11  U84-5  U11-6  U72-39  U30-8  U29-8  U28-8   \
  U27-8
U6-4  U45-6  U30-22  U29-22
U21-13  U4-10  U31-6
U3-12  U54-2  U54-14  U68-15  U10-5  U11-18  U72-9  U31-33   \
```

U29-12 U27-12
R5-1 S1-1 U31-14
R6-1 S2-1 U31-15
R7-1 S3-1 U31-16
R8-1 S4-1 U31-17
U56-8 U5-5
U68-7 U74-11 U77-4
U71-3 U82-4
U16-9 U65-14 U84-2 U11-7 U30-7 U29-7 U28-7 U27-7
U61-13 U58-6 U59-2 U59-3 U60-1 U63-9
U58-12 U62-8
U56-13 U59-12 U13-2
U65-15 U66-15 U44-5 U44-12
U2-7 J5-18 U14-18
U1-12 U19-3 J5-29 U11-38 U13-11 U31-35 U14-29
U1-15 J5-5 U14-5 U15-19
U84-4 U74-5 U85-5
U84-13 U72-34
U88-13 U72-18
U88-1 U72-19
U73-13 U72-26
U73-10 U72-25
U72-7 U78-8
U83-7 U74-2 U85-2
U83-4 U74-1 U85-1
U6-8 U5-1
U13-5 U13-8
R33-1 U20-4
U76-9 U78-37
R20-1 R21-1 R22-1 U12-2
U19-2 U7-3
U43-8 U42-3
U86-9 U78-14
U17-9 U4-11 U5-2
U7-5 U8-4
U19-4 U7-15
U45-9 U56-6
U84-7 U74-6 U85-6
U20-6 U10-7
U76-19 U63-4 U63-13
U20-2 U11-24 R3-1
U76-16 U63-5 U78-2
U69-7 U86-14 U85-11
U8-13 U62-1
U45-10 U5-8
U75-9 U62-4
U63-6 U78-39
U76-2 U63-12 U78-5
U80-9 U80-13 U70-1 U70-4 U70-10 U82-2
U81-12 U82-12

U68-16 U74-16 U77-21
U75-13 U83-3 U72-27
U68-12 U74-14 U77-25
U57-6 U5-12
U69-5 U86-17 U85-10
U69-16 U85-16 U77-2
J2-2 U12-7
J2-3 U12-13 R21-2
U1-13 U42-4
U25-1 U25-9 U78-32
U80-1 U61-2 U61-5 U70-2 U82-7
U71-9 U79-8
U60-2 U60-5 U60-15
U20-9 U10-3
U20-7 U10-4
U11-13 U12-10
R34-1 U25-15
U63-11 U78-6
U76-12 U78-3
U68-18 U74-17 U77-23
U69-3 U86-18 U85-9
U75-4 U83-13 U72-30
U75-12 U83-6 U72-28
U73-6 U72-24 U78-13
U4-5 U5-6
U4-1 U5-11
U11-35 U13-6
U11-5 U12-9
U60-14 U19-5
U44-2 U64-13
U11-14 U12-11
U4-2 U5-9 U15-13
U57-4 U9-5
U57-9 U15-16
U22-12 U57-3 U8-5
U20-12 U11-15 R2-1
U3-17 U49-2 U49-14 U68-4 U10-16 U11-25 U72-14 U31-28 \
U29-18 U27-18
U68-9 U74-13 U77-3
U62-3 U72-17 U78-12
U22-6 U21-5 U8-6 U9-6
U12-1 C15-1
J6-2 U11-33 U13-9 R31-1 C7-1
U16-3 J5-24 U14-24
U16-17 J5-33 U14-33
R80-1 J5-61 U14-61
R77-1 J5-59 U14-59
U20-15 J5-57 U14-57
R78-2 J5-54 U14-54
U16-4 J5-25 U14-25

U1-5 U25-5
J5-16 U14-16 U2-3
U3-1 U23-1 U15-17
U1-2   U4-6
U26-2 U14-65
U45-8 U24-5 U4-13
U24-4 U19-6
U26-9 U56-4 U15-9
U1-11 R10-2 R9-2 C5-1 CR1-2
R22-2 J2-20
J2-5 U12-8 R20-2
J2-4 U12-14
U11-17 J6-3
U73-7 U74-19 U85-19
U62-6 U74-20 U85-20
U73-12 U74-23 U85-23
U73-9 U74-22 U85-22
U11-11 U12-12
U12-3 C15-2
U12-4 C17-1
U80-7
R10-1 S6-1
U3-19 U23-19 U57-8
U22-16 U8-2 U9-2
U22-15 U8-3 U9-3
U17-11 U5-4
U4-3 U4-9 U10-1 U10-11
U3-16 U50-2 U50-14 U68-6 U10-15 U11-20 U72-13 \
  U31-29 U29-17 U27-17
U64-9 U24-6
U6-6 U59-6
U56-11 U4-8
U61-8 U79-12
U61-11 U59-11
U57-5 U8-12
U56-2 U67-14 U44-6
U45-2   U9-7 U28-20 U27-20
U58-1 U8-15
U44-8 U63-10
U57-12 U58-10
U58-5 U59-9 U64-11
U58-9 U58-13 U64-10
U22-19 U66-13 U8-1 U9-1
U22-7 J5-67 U14-67 U15-18
U2-13 J5-20 U14-20
U2-14 J5-21 U14-21
U2-17 J5-22 U14-22
R79-2 J5-53 U14-53
U42-1 U42-7
U76-17 U87-16

```
U76-13  U87-12
U76-8  U87-9
U4-4  U5-3
U12-6 C16-2
U59-10  U59-14
U76-4  U87-5
U88-11  J3-9
U88-3  J3-8
R73-2  R71-2  J3-7
U12-5  C17-2
U18-8  U82-9  U25-13
U71-10  U71-11
U58-3  U58-4
R32-1  J6-1  C6-1
U76-14  U87-15
U76-3  U87-2
U71-6  U82-5
U71-8  U82-13
R67-2  Q2-1  Q1-2
U76-7  U87-6
R68-1  R70-1  U88-6
R28-2  U33-2
R30-2  U33-6
R16-2  U47-2
R18-2  U47-6
U71-1  U81-6
R70-2  R72-2  R66-2  Q2-2
R71-1  Q1-1
R61-1  R62-1  R63-1  R64-1  R65-1   U78-1
U76-18  U87-19
J2-7  R4-1
R35-1  DS1-2

! GROUND NODES

R73-1  U1-3  U1-9  U2-1  U2-10  U3-10  U6-7  U16-1   \
  U16-10  U22-1  U22-10 U23-10  U26-7  U26-10  U34-16   \
  U35-16  U36-16  U37-16  U38-16  U39-16  U40-16  U41-16  U43-7    \
  U45-7  U48-16  U49-16  U50-16  U51-16  U52-16  U53-16   \
  U54-16  U55-16  U56-7  U61-7  U65-8  U66-8  U67-8  U67-12   \
  U68-10  U69-10  U70-7  U71-7  U75-8  U76-1  U76-10   \
  U79-7  U81-7  U86-1  U86-3  U86-4  U86-10  U87-1  U87-10   \
  U88-2  U88-5  U88-7 U88-10  U17-7 \
  U42-2 U42-8  U42-12  U42-14  U42-15  U57-7  U58-7  U44-7  \
  U59-8  U82-8  U60-8  U73-8  U73-15  U83-8  U83-15  U84-8   \
  U84-15  U64-7  U24-7  U19-7  U20-5  U20-8  U21-8   \
  J5-9  J5-35  J5-60 U4-7  U5-7  \
  R4-2  U7-8  U8-8  U9-4  U9-8  U10-8   \
  U10-10  U10-13  U10-17  U10-18  U11-22  J3-1  J3-6   \
  U12-15  C16-1 U13-7 J4-6  J4-7  J4-8  J4-9  S4-2  S3-2   \
```

```
  S2-2   S1-2   S6-2 U25-8   C5-2   U32-1   U32-10   U32-19   U46-1    \
  U46-10   U46-19 Q2-3 U62-7 U63-7 U74-12 U74-18 J6-4 C4-2 C13-2 \
  U72-20   U85-12   U85-18   U77-14   U77-20   U77-22   U78-9    \
  U78-10   U78-15   U78-20   U78-21   U78-22   U78-23   U78-24   U78-31    \
  U31-7   U30-14   U29-14   U28-14   U27-14   U14-9   U14-35   U14-60   \
  U15-6   U15-7   U15-10   C1-2   C6-2   C7-2   \
  U18-7   R35-2    R77-2    R80-2 C8-2    C9-2    Z1-1

! POWER NODES

U18-1   DS1-1   R1-1   R34-2   R33-2   R3-2   U33-3   U33-14 R5-2   R6-2    \
  R7-2    R8-2    U80-14   R32-2   R31-2   R68-2   R69-2   R67-1    \
  R61-2   R62-2   R63-2   R64-2   R65-2   U1-1    U1-6    U1-17    \
  U1-18   U2-20   U3-20   U6-1   U6-12   U6-14   U16-20    \
  U22-3   U22-20   U23-20   U26-14   U34-8   U35-8   U36-8   U37-8    \
  U38-8   U39-8   U40-8   U41-8   U43-1   U43-2   U43-14    \
  U45-14   U47-3   U47-14   U48-8   U49-8   U50-8   U51-8   U52-8    \
  U53-8   U54-8   U55-8   U56-14   U61-14   U65-16   U66-16    \
  U67-10   U67-16   U68-20   U69-20   U70-14   U71-14   U75-1   U75-16    \
  U76-20   U79-1   U79-14   U81-14   U86-20   U87-20   U88-12    \
  U88-14   U17-1   U17-2   U17-14   R66-1   R79-1   R78-1    \
  R26-2   R9-1   J5-62   U14-62   U42-16   U57-14    \
  U58-14   U44-4   U44-10   U44-14   U59-1   U59-5   U59-15    \
  U59-16   U82-16   U60-6   U60-16   U73-16   U83-16   U84-16   U64-14    \
  U24-14   U19-14   U20-1   U20-3   U20-10   U20-11   U20-13    \
  U20-16   U21-16   J5-30   U4-14    U5-14 U7-4    \
  U7-16   U8-16   U9-16   U10-14   U10-20   U11-44   R2-2   U12-16    \
  U13-14   J4-10   J4-11   C1-1 J4-12   J4-13   U25-2    \
  U25-3   U25-4   U25-10   U25-11   U25-12   U25-14   U25-16   CR1-1    \
  U32-20   U46-20   Q1-3   U62-14   U63-14   U74-24    \
  J6-5   U72-36   U72-40   U85-24   U77-1   U77-27   U77-28   U78-7    \
  U78-30   U78-34   U78-35   U78-40   U31-26   U30-1   U30-28    \
  U29-1   U29-28   U28-1   U28-28   U27-1   U27-28   U14-30   U15-14    \
  U15-15   U15-20

! UNUSED OUTPUTS

U26-3
U73-4
U75-3
U75-6
U75-11
U75-14
U86-2
U86-5
U15-4
U59-7
U42-5
U42-6
U42-13
```

U42-11
U42-10
U42-9
U43-3
U43-4
U43-5
U43-6
U43-10
U43-12
U43-13
U67-9
U31-13
U31-12
U31-11
U31-10
U11-8
U11-40
U11-3
U11-43
U11-42
U11-41
U11-32
U11-31
U11-30
U11-16
U11-29
U8-9
U8-7
U9-15
U9-14
U9-13
U9-12
U9-11
U9-10
U21-12
U21-11
U21-10
U21-9
U21-7
U25-7
U25-6
U20-14
U17-3
U17-4
U17-5
U17-6
U17-10
U17-12
U17-13

```
*masters

! PROCESSOR ADDRESS LINES
U14-34
U14-33
U14-32
U14-28
U14-27
U14-26
U14-25
U14-24

U14-23
U14-22
U14-21
U14-20
U14-19
U14-18
U14-17
U14-16

U14-15
U14-14
U14-13
U14-12

! BUFFERED ADDRESS LINES
U16-19
U16-16
U16-15
U16-12
U16-9
U16-6
U16-5
U16-2

U2-19
U2-16
U2-15
U2-12
U2-9
U2-6
U2-5
U2-2

U22-19
U22-16
U22-15
U22-12
```

```
U22-9
U22-6
U22-5

! PROCESSOR DATA LINES

U14-51
U14-49
U14-47
U14-45
U14-43
U14-41
U14-39
U14-37

U14-50
U14-48
U14-46
U14-44
U14-42
U14-40
U14-38
U14-36

! BUFFERED DATA LINES
U23-18
U23-17
U23-16
U23-15
U23-14
U23-13
U23-12
U23-11

U3-18
U3-17
U3-16
U3-15
U3-14
U3-13
U3-12
U3-11
```

(This page is intentionally blank.)

Appendix C
Subprograms for Functional Test and Stimulus Programs

The following programs are included in this appendix:

abort_test
check_loop
check_meas
recover
tst_conten

```
program abort_test(ref)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FUNCTIONAL TEST of the Microprocessor Bus.                            !
!                                                                      !
! This program is called by many of the test programs after the test   !
! program has found a failing circuit.  This program highlights the part!
! with the FAILED test attribute, changes all parts with a TESTING test !
! atribute to UNTESTED, and then checks to see if gfi has enough test   !
! results to make an accusation.  If an accusation exists then the      !
! accusation is displayed.  Otherwise a gfi hint is generated for the   !
! part and the test programs are terminated so that GFI can begin       !
! troubleshooting.                                                      !
!                                                                      !
! TEST PROGRAMS CALLED:                                                 !
!    none                                                              !
!                                                                      !
! GRAPHICS PROGRAMS CALLED:                                             !
!    fail        (part_number)           Highlight part to be failed   !
!                                                                      !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare
   string ref                               ! The ref-pin of the failed part
   global numeric t2o                       ! Buffered I/O on /term2.
   global string array [1:107] part         ! Part shape and positions
   global numeric array [1:107] partatrb    ! Attribute number of part

   ! Next three items relate to Test window displayed by disply_pcb().

   global string testwind1 = "\1B[12;65f\1B[0m\1B[1m" ! Place text in line 2
   global string testwind2 = "\1B[13;65f\1B[0m\1B[1m" ! Place text in line 3
   global string undrtest = "\1B[15;66f\1B[0m"        ! Place text in line 5
end declare
```

```
! Highlight Failed Part.

   n = instr(ref, "-")
   if n = 0 then n = len(ref) + 1
   ic_num = (val(mid(ref, 2, n-2),16))

   ! convert decimal ic_num to hex

   dec100 = ic_num / $100
   dec10 = (ic_num - dec100 * $100) / $10
   dec1 = (ic_num - dec100 * $100 - dec10 * $10)
   hex_ic_num = dec100 * 100 + dec10 * 10 + dec1
   fail(hex_ic_num)

! Change all parts with a TESTING attribute to an UNTESTED attribute and
! display GFI TROUBLESHOOTING in the test window.

   for i = 1 to 107
      if partatrb[i] = 2 then untested(i)
   next
   print on t2o ,testwind1,"      GFI      ",testwind2,"TROUBLESHOOTING"
   print on t2o ,undrtest,"                 "

! If GFI has an accusation then display the accusation otherwise generate
! GFI Hints.

   accusation = gfi accuse
   if accusation = "" then
      gfi hint ref
      fault 'gfi hints generated' '  please  run  gfi'
   else
      fault '' '' accusation
   end if

end program
```

```
program check_loop

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This program checks the DEMO/TRAINER UUT Loopback switches.  If the    !
! loopback switches are not closed then a prompt is generated to close  !
! the loopback switches.  Otherwise no prompt is generated.              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

function pmpt_lpbk
   declare
      string q
   end declare

   print  "Close SW4-4, SW4-5 and SW6-4 for loopback"
   print  "Press \1B[7m ENTER \1B[0m key to continue "
   input  q  \  print
end function

  execute rs232_init()

  write addr $2006, data $AA
  wait time $200

  if ((read addr $2002) and $F) <> $D then
     execute pmpt_lpbk()
     return
  end if

  write addr $201E, data $FF
  write addr $2016, data $BB
  wait time $200

  if (read addr $2016) <> $BB then
     execute pmpt_lpbk()
     return
  end if

  write addr $201C, data $FF

  if ((read addr $201A) and 2) <> 0 then
     execute pmpt_lpbk()
     return
  end if

end program
```

```
program check_meas(dev, start, stop, clock, enable)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Check status of External START, STOP, CLOCK, ENABLE lines,            !
! Return 1 if measurement is complete, display prompt to fix           !
! the external lines, wait for ENTER key, and return 0 if the          !
! measurement times out.                                               !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                    !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   declare string dev
   declare string start
   declare string stop
   declare string clock
   declare string enable

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of program                                               !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   times = 0
   loop while checkstatus(dev) <> $F and times < 100
      times = times + 1
   end loop

! If START fails then STOP, ENABLE and CLOCK will also fail.
! If ENABLE fails then CLOCK will also fail.
! Diagnose cause of failure and only display START if START fails.
! Do not display CLOCK when ENABLE line fails.

   if times < 100 then
      return (1)
   else
      t1 = open device "/term1", mode "unbuffered"

      ! turn autolinefeed off and clear screen

      print "\1B[2J\1B[201"
      n = checkstatus(dev) \ str = "" \ line = ""
      if (n and 4) = 0 then
         line = line + "START "
         str = str + " START to " + start + ","
      else
         if (n and 8) = 0 and stop <> "*" then
            line = line + "STOP, "
            str = str + " STOP to " + stop + ","
         end if
         if (n and 2) = 0 and enable <> "*" then
            line = line + "ENABLE "
            str = str + " ENABLE to " + enable + ","
         else if (n and 1) = 0 then
            line = line + "CLOCK "
            str = str + " CLOCK to " + clock + ","
         end if
      end if
      print "\1B[1;1f", "External line(s) ", line, "failed."
      print "\1B[2;1f", "Connect", str, "\1B[3;1f"
      print "Press \1B[7mENTER \1B[0m to REPEAT, \1B[7mNO \1B[0m to CONTINUE"

! Wait for ENTER key to be pressed.
```

```
      input on t1 ,str
      print "\1B[20h\1B[2J"
      close channel t1
      if str = "\7F" then
         return(1)
      else
         return (0)
      end if
   end if
end program
```

```
program recover

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This program recovers sync between the 82288 Bus Controller and the   !
! 80286 pod.                                                            !
!                                                                       !
! Some of the stimulus programs disable ready before performing stimulus!
! which can cause the 80286 bus controller to get out of sync with the  !
! pod.  The recover() program is executed to resynchronize the bus      !
! controller and the pod.                                               !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    (none)                                                             !
!                                                                       !
! GRAPHICS PROGRAMS CALLED:                                             !
!    (none)                                                             !
!                                                                       !
! Global Variables Modified:                                            !
!    recover_times                        Reset to Zero                 !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   declare global numeric recover_times   ! Count of executing recover().

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   Main part of STIMULUS PROGRAM                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   recover_times = recover_times + 1
   if recover_times <= 1 then
      podsetup 'enable ~ready' "off"      ! POD is out of sync with
      setspace(getspace("memory","word")) ! the 82288 bus controller
      read addr 0                         ! Read in memory space then
      write addr 0, data 0                ! Write in memory space to
      podsetup 'enable ~ready' "on"       ! synchronize 82288 and POD.
   else
      podsetup 'enable ~ready' "off"
      print "Please press the \1B[7mUUT RESET KEY \1B[0m"

      loop until (readstatus() and $10) <> 0  ! wait for RESET active.
      end loop
      podsetup 'enable ~ready' "on"

      loop until (readstatus() and $10) = 0   ! wait for RESET inactive.
      end loop
      print "\1B[2J"
   end if
end program
```

```
program tst_conten (addr, data_bits)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! TEST to isolate DATA BUS CONTENTION to the failing part.              !
!                                                                       !
! This program attempts to determine the cause of Data Bus contention by!
! testing the enable lines of all the devices on the Data Bus.  This    !
! program performs several steps.  First each device on the data bus is !
! accessed and determined to be accessible or inaccessible.  The        !
! variable bad_dev is a mask that records which devices failed.         !
!                                                                       !
! Many times when Data Bus contention exists, the device that has the   !
! bad enable lines can be accessed and the rest of the devices cannot be!
! accessed.  This program checks the mask to see if all except one      !
! device is bad and then tests the enable lines on the device that      !
! appeared good.                                                        !
!                                                                       !
! If all devices are bad or more than one device is good then this test !
! checks the enable lines of all the devices on the Data Bus by brute   !
! force.                                                                !
!                                                                       !
! TEST PROGRAMS CALLED:                                                 !
!    abort_test (ref-pin)                   If gfi has an accusation    !
!                                           display the accusation else !
!                                           create a gfi hint for the   !
!                                           ref-pin and terminate the test!
!                                           program (GFI begins trouble- !
!                                           shooting).                  !
!                                                                       !
! FUNCTIONS CALLED:                                                     !
!    testic     (refname, pin1, pin2)    This function performs a gfi   !
!                                        test on refname.  Then the pins!
!                                        pin1 and pin2 (which are the   !
!                                        enable lines) are checked to   !
!                                        see if they are bad.  If so    !
!                                        abort_test is called and GFI is!
!                                        started on the failing enable  !
!                                        line.  Otherwise all test info !
!                                        about the part is discarded    !
!                                        using the gfi clear command.   !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Main Declarations                                                     !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

declare
   numeric addr                         ! Address where failure occured.
   numeric data_bits                    ! Mask of failing data bits.
   numeric bad_dev = 0                  ! Mask to record failing devices
   numeric array [0:$15] ram_ic         ! Convert RAM bit to part number
   global string contention_checked     ! Record that this test ran.
end declare

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Functions                                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   function testic (ref, pin_a, pin_b)
      declare numeric ref
      declare numeric pin_a
      declare numeric pin_b
```

```
    ! convert decimal ref to hex

      dec100 = ref / 100
    dec10 = (ref - dec100 * 100) / 10
    dec1 = (ref - dec100 * 100 - dec10 * 10)
    href = dec100 * $100 + dec10 * $10 + dec1

    ref_a = "U" + str(href,16) + "-" + str(pin_a,16)
    ref_b = "U" + str(href,16) + "-" + str(pin_b,16)

    if gfi test ref_a fails then
       if (gfi status ref_a) = "bad" then
          abort_test(ref_a)
       else
          if (gfi status ref_b) = "untested" then gfi test ref_b
          if (gfi status ref_b) = "bad" then
             abort_test(ref_b)
          end if
       end if
       gfi clear       ! Only looking at Enable Lines, Clear Other Info.
    end if
 end function


 ram_ic[0] = 55    \  ram_ic[1] = 54          ! RAMs U55, U54
 ram_ic[2] = 53    \  ram_ic[3] = 52          ! RAMs U53, U52
 ram_ic[4] = 51    \  ram_ic[5] = 50          ! RAMs U51, U50
 ram_ic[6] = 49    \  ram_ic[7] = 48          ! RAMs U49, U48
 ram_ic[8] = 41    \  ram_ic[9] = 40          ! RAMs U41, U40
 ram_ic[10] = 39  \  ram_ic[11] = 38          ! RAMs U39, U38
 ram_ic[12] = 37  \  ram_ic[13] = 36          ! RAMs U37, U36
 ram_ic[14] = 35  \  ram_ic[15] = 34          ! RAMs U35, U34

 if contention_checked <> "yes" then
    contention_checked = "yes"
    podsetup 'report intr' "off"
    podsetup 'enable ~ready' "on"
    print "\nl\nlTESTING BUS CONTENTION"


! Read from each device on the bus and record if each device reads correctly.
!
! Then check and see if all components are bad except one.  If so then check
! that component's enable lines.
!
! Otherwise brute force check all enable lines on all components connected to
! the bus.

   ! ROM0 and ROM1

      setspace( getspace("memory", "word"))
      if (read addr $E002A) <> 0 then bad_dev = bad_dev or 1
      if (read addr $F0022) <> 0 then bad_dev = bad_dev or 2

   ! Dynamic RAM

      write addr $1000, data $FFFF
      if (read addr $1000) <> $FFFF then bad_dev = bad_dev or 4
      write addr $1000, data 0
      if (read addr $1000) <> 0 then bad_dev = bad_dev or 4

   ! PIA registers
```

```
      execute pia_init()
      if (read addr $4002) <> $FF then bad_dev = bad_dev or 8
      write addr $4002, data 0
      if (read addr $4002) <> 0 then bad_dev = bad_dev or 8

   ! DUART registers

      execute rs232_init()
      if (read addr $200A) <> $11 then bad_dev = bad_dev or $10
      if (read addr $201A) <> $FF then bad_dev = bad_dev or $10
      if (read addr $2012) <>  $C then bad_dev = bad_dev or $10

   ! Video Controller registers

      execute rs232_init()
      if (read addr 8) <> $FF then bad_dev = bad_dev or $20
      if (read addr $A) <>  0 then bad_dev = bad_dev or $20

! If only one device is good, CLIP and check enable lines on that device.

      if bad_dev <> 0 and bad_dev <> $3F then
      ! CLIP and Check Enable lines on ROMs
         if bad_dev = $7E then
            if (data_bits and $FF) <> 0 then        ! Low data bits are bad
               testic(29, $20, $22)                 ! Check low byte ROM0.
            end if
            if (data_bits and $FF00) <> 0 then      ! High data bits are bad
               testic(30, $20, $22)                 ! Check high byte ROM0.
            end if

         else if bad_dev = $7D then
            if (data_bits and $FF) <> 0 then        ! Low data bits are bad
               testic(27, $20, $22)                 ! Check low byte ROM0.
            end if
            if (data_bits and $FF00) <> 0 then      ! High data bits are bad
               testic(28, $20, $22)                 ! Check high byte ROM0.
            end if

         else if bad_dev = $7B then
            testic (ram_ic[msb(data_bits)], $15, 4) ! Check RAM.
         else if bad_dev = $77 then
            testic (31, 6, 6)                       ! Check PIA.
         else if bad_dev = $2F then
            testic (11, $39, 9)                     ! Check DUART.
         else if bad_dev = $1F then
            testic (72, 2, 3)                       ! Check Video Controller
         end if
      end if

! BRUTE FORCE check enable lines of all devices on bus.

      if (data_bits and $FF) <> 0 then              ! Low data bits are bad
         testic(27, $20, $22)                       ! Check low byte ROM0.
         testic(29, $20, $22)
      end if
      if (data_bits and $FF00) <> 0 then            ! High data bits are bad
         testic(28, $20, $22)                       ! Check high byte ROM0.
         testic(30, $20, $22)
      end if
      testic (ram_ic[msb(data_bits)], $15, 4)       ! Check RAM.
      testic (31, 6, 6)                             ! Check PIA.
      testic (11, $39, 9)                           ! Check DUART.
      testic (72, 2, 3)                             ! Check Video Controller
```

```
      testic (10, $11, 1)                          ! Check Interrupt Buffer
      if bad_dev = $3F then
         if (data_bits and $FF) <> 0 then
            if gfi test "U3-1" fails then abort_test("U3-1")
         end if
         if (data_bits and $FF00) <> 0 then
            if gfi test "U23-1" fails then abort_test("U23-1")
         end if
      end if

      print "BUS CONTENTION TEST PASSES"
   end if
end program
```

(This page is intentionally blank.)

Index