HEWLETT PACKARD

User's Guide

HP 64700 Emulators
Terminal Interface Analyzer

HP 64700 Emulators Terminal Interface: Analyzer User's Guide

Edition 1

64740-90909E1187
Printed in U.S.A. 11/87

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

Logic Systems Division
8245 North Union Boulevard
Colorado Springs, CO 80918, U.S.A.

Printing History

New editions are complete revisions of the manual. The dates on the title page change only when a new edition is published.

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual revisions.

Edition 1 11/87 64740-90909E1187

Using this Manual

This manual will show you how to use the HP 64700 series analyzer with the firmware resident Terminal Interface.

This manual will:

- Briefly introduce the analyzer and its features.
- Show you how to use the analyzer in its simplest, power-up condition. From there, it will progressively show you how and why you would use additional trace commands.
- Show you how to use the external analyzer.
- Show you how to cross-trigger between the emulation analyzer and the external analyzer.
- Show you how to specify analyzer clocks.
- Show you how to save the analyzer configuration in a command file.

This manual will not:

- Show you how to use the analyzer with the PC Interface; this is done in the *HP 64700 Emulators PC Interface: Analyzer User's Guide.*
- Show you how to use the analyzer with the Softkey Interface; this is done in the *HP 64700 Emulators Softkey Interface: Analyzer User's Guide.*
- Describe all analyzer commands options in alphabetical order; this is done in the *HP 64700 Emulators Terminal Interface: User's Reference.*
- Show you how to use the external timing analyzer. Timing analysis is only available when using host computer interfaces such as the PC Interface or the Sofkey Interface. Refer to the appropriate host computer interface *Analyzer User's Guide.*
- Show you how to cross-trigger the analyzers of multiple HP 64700 Series emulators over the Coordinated Measurement

Bus (CMB); this is done in the *HP 64700 Emulators Terminal Interface: CMB User's Guide.*

Organization

Chapter 1 **Introducing the HP 64700 Series Analyzer**. This chapter lists the basic features of the analyzer. The following chapters show you how to use these features.

Chapter 2 **Getting Started**. This chapter shows you how to use the analyzer from its simplest power-up condition to making simple sequence specifications.

Chapter 3 **Accessing Full Analyzer Capability**. This chapter shows you how to access and use the full power and capability of the HP 64700 Series analyzer (more powerful sequencing and the use complex expressions).

Chapter 4 **Using the External Analyzer**. This chapter shows you how to use the external analyzer as part of the emulation analyzer or as an independent state analyzer.

Chapter 5 **Making Coordinated Measurements**. This chapter shows you how to use the analyzer trigger condition to break the emulator

and how to cross-trigger between the emulation analyzer and the external analyzer.

Chapter 6 **Special Analyzer Topics**. This chapter shows you how to name and qualify analyzer clock sources. It shows you how to use slave clocks to demultiplex data on analyzer trace signals. It also shows you how to save and retrieve analyzer command specifications to and from command files.

Contents

Chapter 1 Introducing the HP 64700 Series Analyzer

Chapter 2 Getting Started

Chapter 3 Accessing Full Analyzer Capability

Chapter 4 Using the External Analyzer

Chapter 5 Making Coordinated Measurements

Chapter 6 Special Analyzer Topics

Illustrations

1

Introducing the HP 64700 Series Analyzer

Overview

This manual describes the HP 64700 Series analyzer. Each HP 64700 Series emulator contains an internal emulation analyzer. Your emulator may optionally contain an external analyzer.

The *emulation analyzer* captures emulator bus cycle information synchronously with the processor's clock signal. A *trace* is a collection of these captured states. The *trigger* state specifies when the trace measurement is taken. The *external analyzer* captures activity on signals external to the emulator, typically other target system signals.

The analyzer commands are the same in every emulator; consequently, this manual is shipped with every HP 64700 Series emulator. A block diagram of the analyzer is shown in figure 1-1.

Analyzer Features

This chapter lists basic features of the HP 64700 Series analyzer. The chapters which follow show you how to use these features.

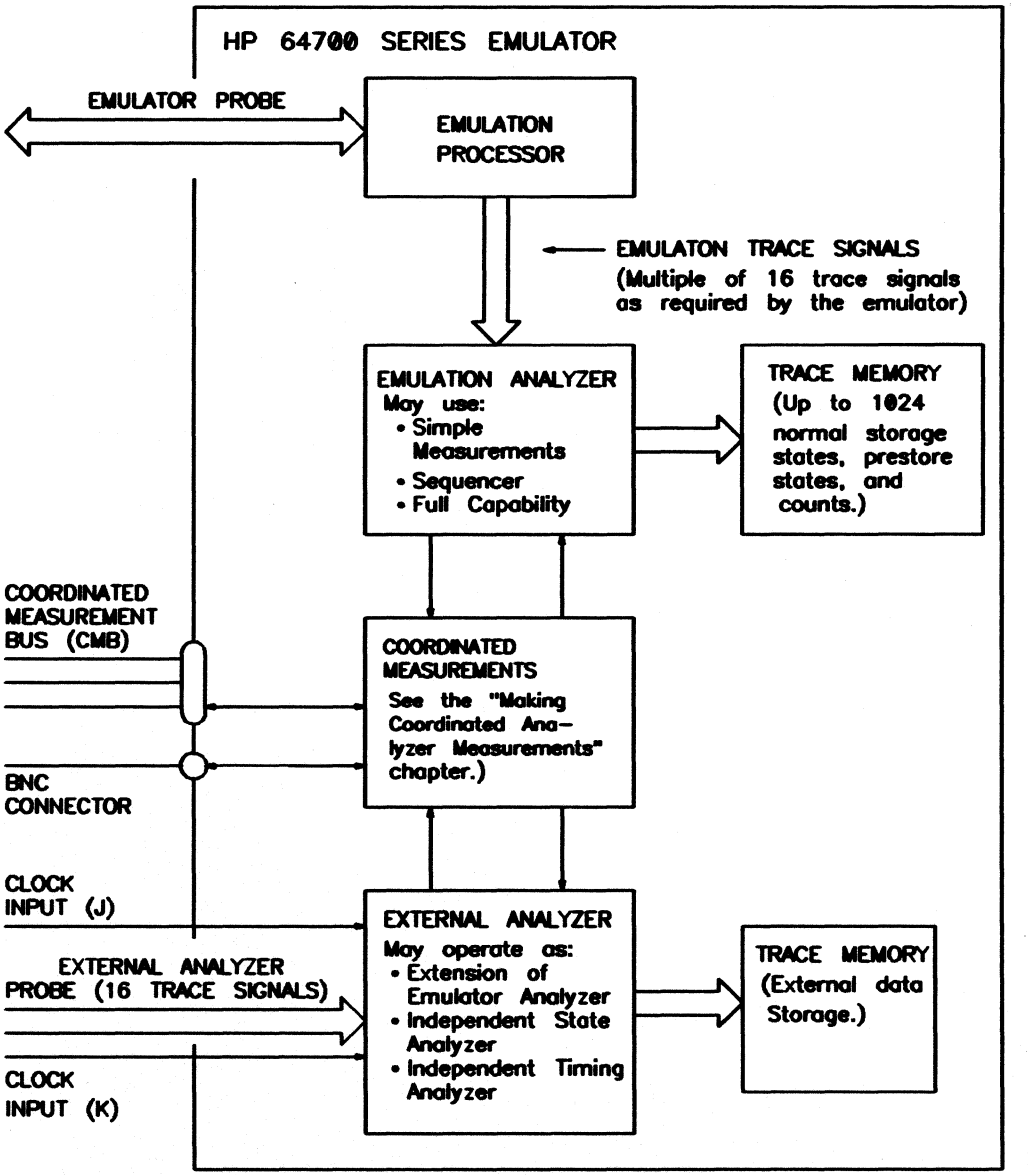


Figure 1-1. Block Diagram of HP 64700 Series Analyzer

Simple Measurements

The default condition of the analyzer allows you to perform a simple measurement by entering a single "trace" command. You can enter additional trace commands to qualify when execution should be traced and which bus cycle states should be stored.

Trace Storage, Prestore, and Count

The analyzer can store up to 1024 states in trace memory. These states can be normal storage states or prestore states (states which precede normal storage states). A count may be associated with normal storage states; you can specify that the analyzer count in either time or the occurrences of some state. When counts are specified, only 512 states can be stored.

Sequencer

You can use the analyzer to search for a particular sequence of states. The sequencer, which makes this possible, has several levels (also called *sequence terms*). Each level of the sequencer can search for two states at a time. When one of these states is found, the sequencer branches to another sequence term. The term that is branched to depends on which state is found first.

Simple Commands for Common Measurements

When the emulator is powered up or initialized, the analyzer is set up in its "easy" configuration. The "easy" configuration hides much of the complexity of the analyzer and makes it easier to use; it allows you to make simple measurements without requiring a thorough knowledge of the analyzer. You can access the full capability of the analyzer via a command to select the "complex" configuration.

External Analysis

Your HP 64700 Series emulator may optionally contain an external analyzer. The external analyzer provides 16 external trace signals and two external clock inputs. You can use the external

analyzer as an extension to the emulation analyzer, as an independent state analyzer, or as an independent timing analyzer.

Coordinated Measurements

When multiple HP 64700 Series emulators are connected via the Coordinated Measurement Bus (CMB), you can use the analyzer to trigger the analyzers of other emulators. You can also use the analyzer to trigger instruments connected to the BNC port. Conversely, the analyzer may be triggered by other emulators and instruments.

Also, if your emulator contains an external analyzer being used as an independent analyzer, coordinated measurements may take place between the emulation analyzer and the external analyzer.

Other Features

The list above is only a basic description of the HP 64700 Series analyzer features. The chapters which follow show you how to use these features.

2

Getting Started

Introduction

This chapter shows you how to use the emulation analyzer from making simple measurements to searching for a sequence of states. It does **not** describe how to access or use the full capability of the analyzer (see the chapter on "Accessing Full Analyzer Capability").

This chapter:

- Describes the sample program on which example measurements are made.
- Describes the default, power-up condition of the analyzer (including how to: initialize the analyzer, start the trace measurement, halt the trace, display the trace status, display the trace, and change the format of the trace listing).
- Describes expressions allowed in trace commands.
- Shows you how to specify a simple trigger.
- Shows you how to specify a storage qualifier.
- Shows you how trace prestore is used.
- Shows you how to change the count qualifier.
- Shows you how to use the sequencer.
- Shows you how to change the position of the trigger state in the trace.

Prerequisites

Before reading the examples in this chapter you should already know how the emulator operates. You should know what the various emulator prompts mean, and you should know how to use the emulation commands. Refer to the appropriate *Terminal Interface: Emulator User's Guide* manual to learn about the emulator; then, return to this manual.

The Sample Program

The sample program is used to illustrate analyzer examples. The sample program is written in assembly language so the disassembled trace listings will be more meaningful.

The examples in this chapter have been generated using an 80186 (HP 64764) emulator. The sample program is written in 80186 assembly language.

It is not important that you know the 80186 assembly language; however, you should understand what the various sections of the program do and associate these tasks with the labels used in the program.

You are encouraged to rewrite the sample program in the assembly language appropriate for your emulator. Then, you can use your analyzer to perform the examples shown in this chapter. Of course, the output of your commands will be different than those shown here.

Description of the Sample Program

A pseudo-code algorithm of the sample program is shown in figure 2-1.

```
                Initialize the stack pointer.
        AGAIN:  Save the two previous random numbers.
                Call the RAND random number generator subroutine.
                Test the two least significant bits of the previous random number.
                    If 00B then goto CALLER_0.
                    If 01B then goto CALLER_1.
                    If 10B then goto CALLER_2.
                    If 11B then goto CALLER_3.
     CALLER_0:  Call the WRITE_NUMBER subroutine.
                Goto AGAIN (repeat program).
     CALLER_1:  Call the WRITE_NUMBER subroutine.
                Goto AGAIN (repeat program).
     CALLER_2:  Call the WRITE_NUMBER subroutine.
                Goto AGAIN (repeat program).
     CALLER_3:  Call the WRITE_NUMBER subroutine.
                Goto AGAIN (repeat program).

WRITE_NUMBER:   Write the random number to a 256 byte data area, using the second
                previous random number as an offset into that area.
                RETURN from subroutine.

        RAND:   Pseudo-random number generator which returns a random number
                from 0-0FFH.
                RETURN from subroutine.
```

Figure 2-1. Pseudo-Code Algorithm of Sample Program

The sample program is not intended to represent a real routine. The program uses four different callers of the WRITE_NUMBER subroutine to simulate situations in real programs where routines are called from many different places. An example later in this chapter will show you how to use the analyzer to determine where a routine is called from.

An assembler listing of the sample program is shown in figure 2-2. It is provided so that you can see the addresses associated with the program labels. The program area, which contains the instructions to be executed by the microprocessor, is located at 400H. The RESULTS area, to which the random numbers are written, is located at 500H. The area which contains a variable used by the RAND subroutine and the locations for the stack is located at 600H.

```
FILE: anly.S             HEWLETT-PACKARD: 80186 Assembler

LOCATION OBJECT CODE LINE     SOURCE LINE

                       1 "80186"
                       2                 ORG     400H
                       3                 ASSUME  DS:ORG,ES:ORG
                       4
    0400 B80000        5 START           MOV     AX,SEG RAND_SEED
    0403 8ED8          6                 MOV     DS,AX
    0405 8ED0          7                 MOV     SS,AX
    0407 B80000        8                 MOV     AX,SEG RESULTS
    040A 8EC0          9                 MOV     ES,AX
    040C BCFE06       10                 MOV     SP,OFFSET STACK
                      11 * The next three instructions move the second
                      12 * previous random number into DI (offset to
                      13 * RESULTS area).
    040F 8AC7         14 AGAIN           MOV     AL,BH
    0411 25FF00       15                 AND     AX,#0FFH
    0414 8BF8         16                 MOV     DI,AX
                      17 * Previous random # moved to BH.
    0416 8AFB         18                 MOV     BH,BL
                      19 * RAND returns random number in AX.
    0418 E83300       20                 CALL    RAND
                      21 * Current random # moved to BL.
    041B 8AD8         22                 MOV     BL,AL
    041D 8AE7         23                 MOV     AH,BH
                      24 * The following instructions determine which
                      25 * caller calls WRITE_NUMBER (depends on last
                      26 * two bits of the previous random number).
    041F D0DC         27                 RCR     AH,1
    0421 7207         28                 JC      ONE_THREE
    0423 D0DC         29                 RCR     AH,1
    0425 7216         30                 JC      CALLER_2
    0427 E90700       31                 JMP     CALLER_0
    042A D0DC         32 ONE_THREE       RCR     AH,1
    042C 7215         33                 JC      CALLER_3
    042E E90600       34                 JMP     CALLER_1
                      35 * The WRITE_NUMBER routine is called from four
                      36 * different places.  The program is repeated
                      37 * after the subroutine return.
    0431 E81400       38 CALLER_0        CALL    WRITE_NUMBER
    0434 EBD9         39                 JMP     AGAIN
    0436 90           40                 NOP
    0437 E80E00       41 CALLER_1        CALL    WRITE_NUMBER
    043A EBD3         42                 JMP     AGAIN
    043C 90           43                 NOP
    043D E80800       44 CALLER_2        CALL    WRITE_NUMBER
    0440 EBCD         45                 JMP     AGAIN
    0442 90           46                 NOP
    0443 E80200       47 CALLER_3        CALL    WRITE_NUMBER
    0446 EBC7         48                 JMP     AGAIN
```

Figure 2-2. Sample Program Listing

```
                        49 * The WRITE_NUMBER routine writes the random
                        50 * number to the RESULTS area.  The second
                        51 * previous number is the offset in this area.
0448 26889D0005         52 WRITE_NUMBER    MOV     RESULTS[DI],BL
044D C3                 53                 RET
                        54 * The RAND routine generates a pseudo-random
                        55 * number from 0-0FFH, and leaves the result
                        56 * in AX.
044E B86D4E             57 RAND            MOV     AX,#4E6DH
0451 26F72E0006         58                 IMUL    RAND_SEED
0456 153903             59                 ADC     AX,#339H
0459 7301               60                 JNC     PAST_INC
045B 42                 61                 INC     DX
045C 26A30006           62 PAST_INC        MOV     RAND_SEED,AX
0460 8BC2               63                 MOV     AX,DX
0462 25FF00             64                 AND     AX,#0FFH
0465 C3                 65                 RET
                        66
                        67                 ORG     500H
                        68 * Random numbers written to this area.
0500                    69 RESULTS         DBS     0FFH
                        70
                        71                 ORG     600H
                        72 * Variable used in RAND subroutine.
0600 0100               73 RAND_SEED       DW      1
0602                    74                 DDS     3FH
06FE                    75 STACK           DWS     1       ; Stack area.
                        76                 END

Errors=    0
```

Figure 2-2. Sample Program Listing (Cont'd)

Before You Can Use the Analyzer

Before you can use the analyzer to perform measurements on the sample program, you must map memory and load the sample program.

Map Memory

The program, destination, and stack areas of the sample program were ORGed at addresses 400H, 500H, and 600H, respectively. Therefore, map the range from 400H through 7ffH to emulation memory before loading the program, as shown in the command below.

```
R>map 400..7ff eram
```

To display the resulting memory map:

```
R>map
  # remaining number of terms    : 15
  # remaining emulation memory   : 1f400h bytes
  map  00400..007ff  eram   # term 1
  map  other tram
```

Mapping memory is described in more detail in your *Terminal Interface: Emulator User's Guide.*

Load the Program

Absolute files, in a number of different file formats, can be loaded into an HP 64700 Series emulator in a number of different ways. Refer to the *Terminal Interface: Emulator User's Guide* for information on loading programs into the emulator.

Run the Program

To start the emulator executing the example you would enter the run command below.

```
R>r 400
U>
```

The address 400H is the start address of the sample program and the "U>" prompt shows that the emulator is executing the "user" sample program.

The Default Trace Specification

After the emulator is powered-up or initialized, the analyzer is in its simplest configuration. The default condition will trigger on any state, and store all captured states. You can simply issue a trace command (**t**) to trace the states currently executing.

Initializing the Analyzer (tinit)

To be sure that the analyzer is in its default or power-up state, or to reset the analyzer to its default state, you can enter the **tinit** (trace initialization) command.

```
U>tinit
```

Starting the Trace (t)

Enter the **t** (trace) command to tell the analyzer to begin monitoring the states which appear on the trace signals. You will see a message which confirms that a trace is started.

```
U>t
  Emulation trace started
```

Halting the Trace (th)

The **th** (trace halt) command allows you to halt a trace measurement. When the **th** command is entered, the message "Emulation trace halted" is displayed.

Displaying the Trace Status (ts)

Enter the **ts** (trace status) command to view what the analyzer is doing (or what the analyzer has done if the trace has completed).

```
U>ts
  --- Emulation Trace Status ---
  NEW User trace complete
  Arm ignored
  Trigger in memory
  Arm to trigger ?
  States 512 (512) 0..511
  Sequence term 2
  Occurrence left 1
```

The first line of the emulation trace status display shows that the user trace has been "completed"; other possibilities are that the trace is still "running" or that the trace has been "halted". The word "NEW" indicates that the most recent trace has not been displayed. The word "User" indicates that the trace was taken in

response to a **t** command; the other possibility is that a "CMB" execute signal started the trace.

The "Arm ignored" line shows that the arm condition, which can be used to qualify trace measurements, is ignored. Consequently, the "Arm to trigger" time is not meaningful and a question mark is displayed. (The "Making Coordinated Measurements" chapter explains arm conditions.)

The trigger state (indicated by state number 0) has been stored in trace memory, as well as the 511 states which follow the trigger. Because the default trigger condition is any state, the first state after the **t** command becomes the trigger state. Because all captured states are stored, the next 511 states are stored in the trace.

The "sequence term" and "occurrence left" items are explained later.

Displaying the Trace (tl)

Use the **tl** (trace list) command to display the trace data.

```
U>tl 0..20
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ----------------------------------  ---------  ---
      0    00434         d9ebH, opcode fetch                  ---   +
      1    00448         8826H, opcode fetch            0.960 uS   .
      2    006fc         0434H, mem write               0.560 uS   .
      3    00448   MOV ES:BYTE PTR 0500H[DI],BL         0.120 uS   .
      4    00449                                        0.280 uS   .
      5    0044a         009dH, opcode fetch            0.120 uS   .
      6    0044c         c305H, opcode fetch            0.560 uS   .
      7    0044e         6db8H, opcode fetch            0.520 uS   .
      8    00450         264eH, opcode fetch            1.080 uS   .
      9    00536         xx94H, mem write               0.560 uS   .
     10    0044d   RET                                  0.120 uS   .
     11    006fc         0434H, mem read                0.840 uS   .
     12    00434         d9ebH, opcode fetch            0.800 uS   .
     13    00436         e890H, opcode fetch            0.560 uS   .
     14    00434   JMP SHORT 040fH                      0.120 uS   .
     15    00438         000eH, opcode fetch            0.400 uS   .
     16    0040f         8axxH, opcode fetch            0.680 uS   .
     17    00410         25c7H, opcode fetch            0.560 uS   .
     18    0040f   MOV AL,BH                            0.120 uS   .
     19    00412         00ffH, opcode fetch            0.400 uS   .
     20    00411   AND AX,#00ffH                       0.160 uS   .
```

The first column on the trace list contains the line number. The trigger is always on line 0.

The second column contains the address information associated with the trace states. Addresses in this column may be locations of instruction opcodes on fetch cycles, or they may be sources or destinations of operand cycles.

The third column shows mnemonic information about the emulation bus cycle. The disassembled instruction mnemonic is shown for instruction cycles. The data and mnemonic status ("d9ebH, opcode fetch", for example) are shown for bus cycles. In the 80186 emulator, the mnemonic information is already disassembled (i.e., assembly language mnemonics are shown); in other emulators, like the 68000, you must use the **-d** option to the **tl** command to view the mnemonic information in disassembled form.

The fourth column shows the count information (**time** is counted by default). The "R" indicates that each count is relative to the previous state.

The fifth column contains information about the sequencer. The "+" on line 0 indicates the state satisfied a branch condition (in this case, a trigger condition).

An important thing to notice about the trace list above involves lines 7, 13, and 15. These states show opcode fetches for instructions which are not executed because of a transfer of execution to other addresses. This can happen with microprocessors like the 80186 and the 68000 because they have pipelined architectures or instruction queues which allow them to prefetch the next instructions before the current instruction is finished executing.

You can enter the **help tl** command to see the other options available when displaying a trace.

Expressions in Trace Commands

So far, the default trace specifications have been used, and you have not entered any expressions. Expressions are used in commands which qualify the trace. This section describes the expressions which may be used in trace commands. Expressions may be specified in the following forms (the pound sign, #, appears before comments):

```
any/all                                          # special tokens
never/none
arm

label=<value>
label!=<value>
label=<value> and label=<value> ...              # this condition
label!=<value> or label!=<value> ...             # not this condition
label=<value>..<value>                           # this range
label!=<value>..<value>                          # not this range
```

Note 

If you wish to specify an expression such as "label = <value> and label != <value>", you must configure the analyzer so that you have access to its full capability.

Note 

Only one range resource is available. You can, however, use this range (or "not this range") in more than one trace command.

Tokens

The tokens **any** or **all** specify any or all conditions; you can use these tokens interchangeably. The tokens **never** or **none** specify false conditions; they are used to turn off qualifiers. The **never** and **none** tokens may also be used interchangeably. The **arm** token represents a condition external to the analyzer. Arm conditions are described in the "Making Coordinated Measurements" chapter.

Trace Labels

Labels shown in the forms above may be predefined trace labels or labels which you define with the **tlb** (trace label) command or the **xtlb** (external trace label) command if you have an external analyzer. Trace labels can be up to 31 characters long.

Predefined Trace Labels

To see the trace labels which have been predefined, enter the **tlb** (trace label) command with no options and the **xtlb** (external trace label) command with no options (if an external analyzer is present).

```
U>tlb
  #### Emulation trace labels
  tlb addr 0..19
  tlb data 20..35
  tlb stat 36..46
U>xtlb
  #### External trace labels
  xtlb xbits 0..15
```

The labels **addr**, **data**, **stat**, and **xbits** are predefined. The **addr** label represents the trace signals (0 through 19) which monitor the emulation processor's address pins. The **data** label represents the trace signals (20 through 35) which monitor the emulation processor's data pins. The **stat** label represents the trace signals (36 through 46) which monitor other emulation processor signals. The **xbits** label represents the external trace signals. The definitions of the address, data, and status bits are different for each emulator.

Values

Values are a series of 1s, 0s, or don't cares (x). Don't cares are not allowed in ranges or decimal numbers. A value of all don't cares may be represented by a question mark (?).

Constants

A value may be specified as a constant in any of the following number bases. (Constants with no base specified are assumed to be hexadecimal numbers.)

- Hexadecimal (base **H** or **h**). For example: 6eh, 9xH, 0f3, or 0cfh. (The leading digit of a hexadecimal constant must be 0-9.)
- Decimal (base **T** or **t**, for base "ten"). For example: 27t or 99T. (Don't cares are not allowed in decimal numbers.)
- Binary (base **Y** or **y**). For example: 1101y, 01011Y, or 0xx10xx11y. (The leading digit of a binary constant must be 0 or 1. Do not use the characters "B" or "b" to specify the base of binary numbers because they will be interpreted as hexadecimal numbers; for example, 1B equals 27 decimal.)
- Octal (base **Q**, **q**, **O**, or **o**). For example: 777o, 6432q, or 7xx3Q. (The leading digit of an octal constant must be 0-7.)

Operators

When specifying values, constants can be combined with the following operators (in descending order of precedence):

-, ~	Unary two's complement, unary one's complement. The unary two's complement operator is not allowed on constants containing don't care bits.
*, /, %	Integer multiply, divide, and modulo. These operators are not allowed on constants containing don't care bits.
+, -	Addition, subtraction. These operators are not allowed on constants containing don't care bits.
<<, <<<, >>, >>>	Shift left, rotate left, shift right, rotate right.
&	Bitwise AND.
^	Bitwise exclusive or, XOR.
|	Bitwise inclusive OR.
&&	Logical AND/bit-wise merge. When bits are different, the first value overrides the second; e.g., 10xxy && 11x1y = = 10x1y.

Note All operations are carried out on 32-bit numbers.

Refer to the *Terminal Interface: User's Reference* description of **expr** for operator truth tables.

Predefined Equates for Emulation Analyzer Status

The **equ** (specify equates) command allows you to equate values with names. Equates for common status values are predefined. To view the names equated with common analysis status, enter the **equ** command with no options. (These status equates are also listed in the **help proc** information.)

```
U>equ
  ### Equates ###
  equ bus=1xxxxxxxxxxy          # Bus cycle.
  equ coproc=0xxxxxx0xxxxy      # Coprocessor cycle.
  equ dma=0xxxxx1xxxxxy         # DMA cycle.
  equ grd=0xxxx1xxxxxxy         # Guarded memory access.
  equ hlt=0xxxxxxxx100y         # Halt acknowledge cycle.
  equ instr=0xxxxxxxxxxy        # Executed instruction state.
  equ inta=0xxxxxxxx111y        # Interrupt acknowledge cycle.
  equ ior=0xxxxxxxx110y         # I/O port read cycle.
  equ iow=0xxxxxxxx101y         # I/O port write cycle.
  equ mr=0xxxxxxxx010y          # Memory read cycle.
  equ mw=0xxxxxxxx001y          # Memory write cycle.
  equ of=0xxxxxxxx011y          # Opcode fetch.
  equ proc=0xxxxx0xxxxxy        # Processor (not DMA) cycle.
  equ rom=0xxx1xxxxxxxy         # Access to ROM cycle.
```

These predefined equates may be used to specify values for the **stat** trace label. For example:

```
stat=bus
```

is the same as:

```
stat=0xxxxxxxxxxy
```

Refer to the appropriate *Terminal Interface: Emulator User's Guide* for information on the status signals for your HP 64700 series emulator.

Expression Examples

Some example trace command expressions follow.

```
addr=500 and data=30 and stat=mr
addr=400+5*20t and data=0
stat=0xx10y
addr=520..532
stat!=0xx10y or stat!=0x1xxy
```

Changing the Trace Format (tf)

You can change the format of the trace information with the **tf** (trace format) command. Use the **help tf** command to review the options available.

```
U>help tf

  tf - specify trace display format

    tf                          - display current format
    tf <label>,<base>           - display the label in the specified base
    tf mne                      - disassembled mnemonic
    tf count                    - count, absolute (relative to trigger)
    tf count,a                  - count, absolute (relative to trigger)
    tf count,r                  - count, relative to preceding state
    tf seq                      - sequencer state change
    tf mne <label>,<base> count count,r seq
                                - multiple fields may be specified
    tf addr,H mne count,r seq   - default format

  --- VALID <label> NAMES ---
    any <label> defined via the tlb or xtlb command

  --- VALID <base> OPTIONS ---
    Y or y = binary             T or t = decimal
    H or h = hexadecimal        Q, q, O, or o = octal
    A or a = ascii
    <base> defaults to hex if not specified
```

The **tf** command primarily allows you to arrange the columns of trace information in a different manner. However, notice that you can include any trace label in the trace. (This is especially useful with the external analyzer.) Also, notice that the trace label information can be displayed in various number bases, and that counts can be displayed relative or absolute. To display the default trace format, enter the **tf** command with no options.

```
U>tf
  tf addr,H mne count,R seq
```

The following trace format command will move the sequencer information to the first column, add the status information in binary format, and delete the count column.

```
U>tf seq addr,h stat,y mne
U>tl
   Line   seq    addr,H   stat,Y        8018x mnemonic,H
  -----   ---    ------   -----------   ---------------------------------
     21    .     00414    11000010011         f88bH, opcode fetch
     22    .     00416    11000010011         fb8aH, opcode fetch
     23    .     00414    01000010011   MOV DI,AX
     24    .     00418    11000010011         33e8H, opcode fetch
     25    .     00416    01000010011   MOV BH,BL
     26    .     0041a    11000010011         8a00H, opcode fetch
     27    .     00418    01000010011   CALL NEAR PTR 044eH
     28    .     0041c    11000010011         8ad8H, opcode fetch
     29    .     0044e    11000010011         6db8H, opcode fetch
     30    .     006fc    11000010001         041bH, mem write
     31    .     0044e    01000010011   MOV AX,#4e6dH
     32    .     00450    11000010011         264eH, opcode fetch
     33    .     00452    11000010011         2ef7H, opcode fetch
     34    .     00451    01000010011   IMUL ES:WORD PTR 0600H
     35    .     00454    11000010011         0600H, opcode fetch
     36    .     00452    01000010011
     37    .     00456    11000010011         3915H, opcode fetch
     38    .     00600    11000010010         0119H, mem read
     39    .     00458    11000010011         7303H, opcode fetch
     40    .     0045a    11000010011         4201H, opcode fetch
     41    .     00456    01000010011   ADC AX,#0339H
```

Notice that the number of lines specified in the last **tl** (trace list) command become the default.

Enter the following command to return to the default trace format.

```
U>tf addr,h mne count,r seq
```

Specifying a Simple Trigger (tg)

The **tg** (specify simple trigger) command allows you to specify when the analyzer should begin storing states. For example, suppose you want to look at the execution of the sample program after the AGAIN label, and therefore, you would like to begin storing states after the AGAIN address occurs. To do this you could enter the **tg** command shown below and display the trace.

```
U>tg addr=40f
U>t
  Emulation trace started
U>ts
  --- Emulation Trace Status ---
  NEW User trace complete
  Arm ignored
  Trigger in memory
  Arm to trigger ?
  States 512 (512) 0..511
  Sequence term 2
  Occurrence left 1
U>tl
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ----------------------------------  ---------  ---
      0    0040f         8axxH, opcode fetch                  ---   +
      1    00410         25c7H, opcode fetch            0.520 uS   .
      2    0040f   MOV AL,BH                            0.120 uS   .
      3    00412         00ffH, opcode fetch            0.440 uS   .
      4    00411   AND AX,#00ffH                        0.120 uS   .
      5    00414         f88bH, opcode fetch            0.400 uS   .
      6    00416         fb8aH, opcode fetch            0.560 uS   .
      7    00414   MOV DI,AX                            0.120 uS   .
      8    00418         33e8H, opcode fetch            0.400 uS   .
      9    00416   MOV BH,BL                            0.160 uS   .
     10    0041a         8a00H, opcode fetch            0.400 uS   .
     11    00418   CALL NEAR PTR 044eH                   0.120 uS   .
     12    0041c         8ad8H, opcode fetch            0.400 uS   .
     13    0044e         6db8H, opcode fetch            0.960 uS   .
     14    006fc         041bH, mem write               0.560 uS   .
     15    0044e   MOV AX,#4e6dH                        0.120 uS   .
     16    00450         264eH, opcode fetch            0.400 uS   .
     17    00452         2ef7H, opcode fetch            0.560 uS   .
     18    00451   IMUL ES:WORD PTR 0600H               0.280 uS   .
     19    00454         0600H, opcode fetch            0.240 uS   .
     20    00452                                        0.160 uS   .
```

In the trace list above, line 0 shows the beginning of the program loop and line 11 shows the call of the RAND subroutine. The disassembled mnemonics on lines 15 and 18 show instructions which are executed in the RAND subroutine.

As you can see in the trace status display, 512 analyzer states are saved in the trace list. To display the "next" lines in a trace list, enter the **tl** (trace list) command with no options.

```
U>tl
   Line    addr,H  8018x mnemonic,H                        count,R  seq
  -----    ------  ---------------------------------  ---------  ---
     21    00456         3915H, opcode fetch            0.400 uS   .
     22    00600         5c9eH, mem read                0.800 uS   .
     23    00458         7303H, opcode fetch            0.560 uS   .
     24    0045a         4201H, opcode fetch            0.520 uS   .
     25    00456   ADC AX,#0339H                        3.680 uS   .
     26    00459   JAE SHORT 045cH                      0.560 uS   .
     27    0045c         a326H, opcode fetch            0.240 uS   .
     28    0045c         a326H, opcode fetch            0.960 uS   .
     29    0045e         0600H, opcode fetch            0.560 uS   .
     30    0045c   MOV ES:0600H,AX                      0.120 uS   .
     31    0045d                                        0.120 uS   .
     32    00460         c28bH, opcode fetch            0.280 uS   .
     33    00600         9680H, mem write               0.680 uS   .
     34    00460   MOV AX,DX                           0.160 uS   .
     35    00462         ff25H, opcode fetch            0.400 uS   .
     36    00464         c300H, opcode fetch            0.520 uS   .
     37    00462   AND AX,#00ffH                       0.160 uS   .
     38    00466         f006H, opcode fetch            0.400 uS   .
     39    00465   RET                                  0.280 uS   .
     40    00468         0001H, opcode fetch            0.240 uS   .
     41    006fc         041bH, mem read                0.560 uS   .
```

In the trace list above you see the last few instructions executed by the RAND subroutine (the RET is the last instruction). To see the instructions executed upon return from the RAND subroutine, enter the **tl** command again.

```
U>tl
   Line    addr,H  8018x mnemonic,H                        count,R  seq
  -----    ------  ---------------------------------  ---------  ---
     42    0041b         8axxH, opcode fetch            0.800 uS   .
     43    0041c         8ad8H, opcode fetch            0.560 uS   .
     44    0041b   MOV BL,AL                            0.120 uS   .
     45    0041e         d0e7H, opcode fetch            0.400 uS   .
     46    0041d   MOV AH,BH                            0.160 uS   .
     47    00420         72dcH, opcode fetch            0.400 uS   .
     48    0041f   RCR AH,1                             0.120 uS   .
     49    00422         d007H, opcode fetch            0.440 uS   .
     50    00421   JB SHORT 042aH                       0.120 uS   .
     51    00424         72dcH, opcode fetch            0.400 uS   .
     52    00423   RCR AH,1                             0.280 uS   .
     53    00426         e916H, opcode fetch            0.280 uS   .
     54    00425   JB SHORT 043dH                       0.120 uS   .
     55    00428         0007H, opcode fetch            0.400 uS   .
     56    0043d         e8xxH, opcode fetch            0.960 uS   .
     57    0043e         0008H, opcode fetch            0.560 uS   .
     58    0043d   CALL NEAR PTR 0448H                  0.120 uS   .
     59    00440         cdebH, opcode fetch            0.400 uS   .
     60    00448         8826H, opcode fetch            0.960 uS   .
     61    006fc         0440H, mem write               0.520 uS   .
     62    00448   MOV ES:BYTE PTR 0500H[DI],BL        0.160 uS   .
```

The instructions shown in the trace list above decide which caller will call the WRITE_NUMBER subroutine. Line 58 shows the

disassembled mnemonic of the instruction which calls the WRITE_NUMBER subroutine. The address information shows that the caller is CALLER_2. Line 62 shows the MOV instruction associated with the WRITE_NUMBER subroutine. To view the remaining instruction cycles of the WRITE_NUMBER subroutine, enter the **tl** command again.

```
U>tl
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ---------------------------------------  ---------  ---
     63    00449                                            0.280 uS   .
     64    0044a         009dH, opcode fetch                0.120 uS   .
     65    0044c         c305H, opcode fetch                0.560 uS   .
     66    0044e         6db8H, opcode fetch                0.520 uS   .
     67    00450         264eH, opcode fetch                1.080 uS   .
     68    0051d         5fxxH, mem write                   0.560 uS   .
     69    0044d   RET                                      0.120 uS   .
     70    006fc         0440H, mem read                    0.840 uS   .
     71    00440         cdebH, opcode fetch                0.800 uS   .
     72    00442         e890H, opcode fetch                0.520 uS   .
     73    00440   JMP SHORT 040fH                          0.160 uS   .
     74    00444         0002H, opcode fetch                0.400 uS   .
     75    0040f         8axxH, opcode fetch                0.680 uS   .
     76    00410         25c7H, opcode fetch                0.560 uS   .
     77    0040f   MOV AL,BH                                0.120 uS   .
     78    00412         00ffH, opcode fetch                0.400 uS   .
     79    00411   AND AX,#00ffH                            0.160 uS   .
     80    00414         f88bH, opcode fetch                0.400 uS   .
     81    00416         fb8aH, opcode fetch                0.520 uS   .
     82    00414   MOV DI,AX                                0.160 uS   .
     83    00418         33e8H, opcode fetch                0.400 uS   .
```

Line 69 in the trace list above shows the RET instruction associated with the WRITE_NUMBER subroutine. Line 68 shows the random number 5FH is written to address 51DH.

The bus cycle data contains "don't cares" when bytes are read or written. Lower byte writes are made to even addresses, and upper byte writes are made to odd addresses.

Line 77 shows the AGAIN address associated with the next loop of the program.

Specifying an Occurrence Count

When specifying a simple trigger, you can include an occurrence count. The occurrence count specifies that the analyzer trigger on the Nth occurrence of some state. For example, to trigger the analyzer when the address 40FH occurs a hundred times, enter the command below.

```
U>tg addr=40f 100
```

The default base for an occurrence count is decimal. You may specify occurrence counts from 1 to 65535.

Specifying Storage Qualifiers (tsto)

By default, all captured states are stored; however, you can qualify which states get stored with the **tsto** (trace storage qualifier) command. For example, to store only the states which write random numbers to the RESULTS area, enter the following command.

```
U>tsto addr=500..5ff
```

Issuing the trace command and then listing the trace will result in a display similar to the one shown below.

```
U>t
  Emulation trace started
U>tl
   Line   addr,H  8018x mnemonic,H                        count,R  seq
  -----   ------  ---------------------------------  ---------  ---
      0    0040f  INSTRUCTION--opcode unavailable          ---   +
      1    0055a        xx16H, mem write                31.48 uS   .
      2    0050b        11xxH, mem write                34.44 uS   .
      3    00516        xx45H, mem write                36.48 uS   .
      4    00511        dbxxH, mem write                36.48 uS   .
      5    00545        10xxH, mem write                35.40 uS   .
      6    005db        8fxxH, mem write                34.72 uS   .
      7    00510        xxb0H, mem write                35.40 uS   .
      8    0058f        39xxH, mem write                35.00 uS   .
      9    005b0        xxe2H, mem write                36.48 uS   .
     10    00539        afxxH, mem write                34.44 uS   .
     11    005e2        xx85H, mem write                35.40 uS   .
     12    005af        9cxxH, mem write                36.48 uS   .
     13    00585        35xxH, mem write                35.00 uS   .
     14    0059c        xx3bH, mem write                36.24 uS   .
     15    00535        c1xxH, mem write                35.40 uS   .
     16    0053b        45xxH, mem write                36.48 uS   .
     17    005c1        7dxxH, mem write                36.48 uS   .
     18    00545        11xxH, mem write                36.20 uS   .
     19    0057d        e0xxH, mem write                36.20 uS   .
     20    00511        3fxxH, mem write                35.00 uS   .
```

Notice that the trigger state (line 0) is included in the trace list; trigger states are always stored.

This trace shows that the last two hex digits of the address in the RESULTS area are the same as the random number which gets written two states earlier (see the data in the "mnemonic" column of the trace list). This is expected because the sample program writes the current random number using the second previous random number as an offset into the RESULTS area.

Prestoring States (tpq)

Suppose you find a bug in a subroutine, but you determine that the problem is actually due to something set up by the calling routine. Suppose also that the subroutine is called from a variety of places in your program. Prestore can be used to determine where the subroutine is called from when the bug occurs.

Prestore allows you to save up to two states which precede a normal store state. Prestore is turned off by default. However, you can use the **tpq** command to specify a prestore qualifier.

Qualifying Prestore States

You can use a prestore qualifier to find out which caller calls the WRITE_NUMBER subroutine in the sample program. Because you know the CALL assembly language instruction is used to call a subroutine, you can qualify prestore states as states whose data equals the CALL opcode.

```
U>tpq data=0e8xx
U>t
  Emulation trace started
U>tl
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ----------------------------------  ---------  ---
     -1    00434   INSTRUCTION--opcode unavailable        prestore   .
      0    0040f   INSTRUCTION--opcode unavailable             ---   +
      1    00430         e800H, opcode fetch              prestore   .
      2    00443         e8xxH, opcode fetch              prestore   .
      3    005d4         xxcdH, mem write                 31.48 uS   .
      4    00430         e800H, opcode fetch              prestore   .
      5    00437         e8xxH, opcode fetch              prestore   .
      6    0057f         a4xxH, mem write                 36.20 uS   .
      7    0043a   INSTRUCTION--opcode unavailable        prestore   .
      8    00431         e8xxH, opcode fetch              prestore   .
      9    005cd         91xxH, mem write                 35.00 uS   .
     10    00430         e800H, opcode fetch              prestore   .
     11    00437         e8xxH, opcode fetch              prestore   .
     12    005a4         xxb9H, mem write                 36.48 uS   .
     13    00430         e800H, opcode fetch              prestore   .
     14    00437         e8xxH, opcode fetch              prestore   .
     15    00591         74xxH, mem write                 36.24 uS   .
     16    0043a   INSTRUCTION--opcode unavailable        prestore   .
     17    00431         e8xxH, opcode fetch              prestore   .
     18    005b9         8exxH, mem write                 34.96 uS   .
     19    00436         e890H, opcode fetch              prestore   .
```

The prestore state immediately preceding each write state shows the address of the caller.

The analyzer uses the same resource to save prestore states as it does to save count tags. Consequently, the "prestore" string is shown in the "count" column of the trace list. Notice that the time counts are relative to the previous normal storage state. Turning off the count qualifier does not turn off prestore; however, the "prestore" string cannot be seen in the "count" column of the trace list.

States which satisfy the prestore qualifier and the storage qualifier at the same time are stored as normal states.

Turning Off Prestore

When you do not wish to have prestored states saved in the trace, you can turn off the prestore feature with the following **tpq** (trace prestore qualifier) command.

```
U>tpq none
```

Changing the Count Qualifier (tcq)

Suppose now that you are interested in only one address in the RESULTS area. You wish to see how many loops of the program occur between each write of a random number to this address. You can use the **tcq** (trace count qualifier) command to count a state which occurs once on each loop of the program. For example, let the address of interest be 5C2H. The following commands set up the sequencer so that only this state is stored in the trace.

```
U>tg addr=5c2
U>tsto addr=5c2
```

In the analyzer's default state, the count qualifier is **time**, which means that the time between states in the trace is saved. Entering the **tcq** command with no options shows the current count qualifier.

```
U>tcq
  tcq time
```

Specify the count qualifier as the AGAIN address (40FH) which gets executed once on each program loop. Then, start the trace and list the trace.

```
U>tcq addr=40f
U>tf addr,h mne count,r count,a
U>t
  Emulation trace started
U>tl
   Line    addr,H  8018x mnemonic,H                          count,R     count,A
  -----    ------  ------------------------------------  ---------   ---------
      0    005c2         xx75H, mem write                       ---           0
      1    005c2         xx2bH, mem write                        92          92
      2    005c2         xx90H, mem write                       166         258
      3    005c2         xxeaH, mem write                       124         382
      4    005c2         xxb7H, mem write                       140         522
      5    005c2         xxbfH, mem write                       274         796
      6    005c2         xxd3H, mem write                       124         920
      7    005c2         xx44H, mem write                       364        1284
      8    005c2         xx33H, mem write                      1256   2.540e03
      9    005c2         xx8dH, mem write                       478   3.018e03
     10    005c2         xxe5H, mem write                       148   3.166e03
     11    005c2         xx78H, mem write                       274   3.440e03
     12    005c2         xxecH, mem write                       272   3.712e03
     13    005c2         xx5dH, mem write                      1062   4.774e03
     14    005c2         xxa2H, mem write                       610   5.384e03
     15    005c2         xxd2H, mem write                       540   5.924e03
     16    005c2         xx46H, mem write                       746   6.670e03
     17    005c2         xx17H, mem write                       434   7.104e03
     18    005c2         xxe8H, mem write                       756   7.860e03
     19    005c2         xx44H, mem write                       682   8.542e03
     20    005c2         xx78H, mem write                       192   8.734e03
```

The trace listing shown above shows that the program executes a variable number of times for each time that a random number is written to 5C2H. The command which follows will change the trace format back to its previous specification.

```
U>tf addr,h mne count,r seq
```

Using the Sequencer (tsq)

The sequencer is a state machine that searches for a particular sequence of states. The sequencer has several levels, called *sequence terms*. Each sequence term can search for two states at a time, and the primary state may have an occurrence count specified. If the primary state occurs the number of times specified, the sequencer branches to the next term; if the secondary state is found before the primary state occurs the number of

times specified, the sequencer branches back to the first term. The same secondary branch condition is used for all sequence terms, and secondary branches are always back to the first term; therefore, the secondary branch is called the *global restart.*

Resetting the Sequencer (tsq -r)

To reset the sequencer to its default, power-up state use the **-r** option to the **tsq** (trace sequencer) command. To display the default sequencer specification, enter the **tsq** command with no options.

```
U>tsq -r
U>tsq
  tif 1 any       # Any state will cause a branch out of term 1.
  tsto all        # Store all states.
  telif never     # Global restart turned off.
```

The Default Sequencer Specification

After power-up, initialization, or sequencer reset, the sequencer consists of one term (see figure 2-3).

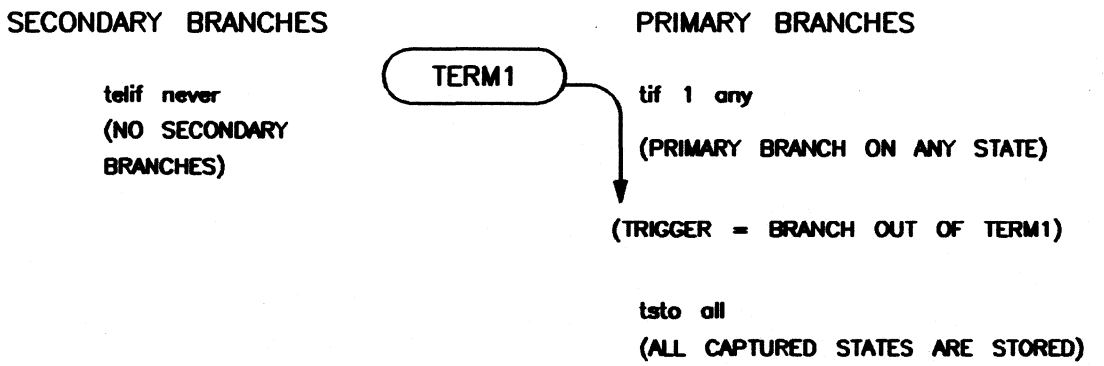


Figure 2-3. The Default Sequencer Specification

It may be helpful to think of the **tif** (primary branch expression) command as a conditional statement. For example, "If (some state occurs), then branch".

Because sequence term 1 is the last term and a branch out of the last term constitutes the trigger, the primary branch expression (**any**) of term 1 specifies the trigger condition. The expression **any** says that any captured trace state will cause a branch. Therefore, the trigger will occur immediately after the **t** (trace) command is issued (if instructions are being executed).

The **tsto** (trace storage qualifier) command specifies that **all** captured states are stored. The trace storage qualifier is a global; that is, it applies to all sequence terms. In addition to states which satisfy the trace storage qualifier, any state which causes a branch is stored in trace memory. Also, prestore states can be saved before states which satisfy the trace storage qualifier.

The **telif** command is used to specify the secondary branch expression for every sequence term; this expression is called the *global restart*. It may be helpful to think of the **telif** command as an "else if" conditional statement. For example, "Else if (some state occurs before) then branch to term 1".

The global restart in the default sequencer specification is **never**. This means no trace state can cause a secondary branch.

You have already seen how the **tsto** command is used. You will learn how to use the **tif** and **telif** commands later in this chapter.

Simple Trigger and the Sequencer

The simple trigger command used previously in this chapter has the following effect on the sequencer:

```
U>tg addr=40f      # If address of 40FH occurs once, then trigger.
U>tsq
  tif 1 addr=40f
  tsto all
  telif never
```

Notice that only the primary branch expression of the first sequence term (the trigger condition) is different than the default sequencer specification. The address 40FH is the AGAIN address of the sample program, the first address of the sample program loop.

A trace state whose address equals 40FH will trigger the analyzer, causing trace memory to be filled with states and stop.

When the **tg** command is entered with no options, the primary branch expression of the first sequence term is displayed. This is the trigger condition only when one term exists in the sequencer.

Primary and Secondary Branch Expressions (tif, telif)

You can use sequence terms to trace a specific combination of events. For example, CALLER_3 can be used to write any random number, but suppose you want to trace only the situation where CALLER_3 is used to write a random number to address 5C2H. You can set up the sequencer so that it first searches for CALLER_3 by specifying the address of CALLER_3 as the primary branch expression of the first sequence term.

```
U>tif 1 addr=443
```

After CALLER_3 is found, the sequencer should then search for the write to address 5C2H. You can do this by specifying the address 5C2H as the primary branch expression of the second sequence term.

```
U>tif 2 addr=5c2
```

However, if the program jumps to AGAIN before the write to 5C2H, you know that CALLER_3 is not used to write the random number this time, and the sequencer should start over. You can specify the global restart expression to do this.

```
U>telif addr=40f
```

If the write to address 5C2H occurs before the program executes the instruction at AGAIN, the sequencer will take a primary branch out of the last term and trigger the analyzer. The resulting sequencer specification is shown below.

```
U>tsq
  tif 1 addr=443
  tif 2 addr=5c2
  tsto all
  telif addr=40f
```

The sequencer specification above is represented in figure 2-4. The primary branch expression of the first sequence term is the

address associated with CALLER_3 (443H). The primary branch expression for the second sequence term is the specific write condition we would like to trace; it is also the trigger condition. The primary branch out of the second term constitutes the trigger.

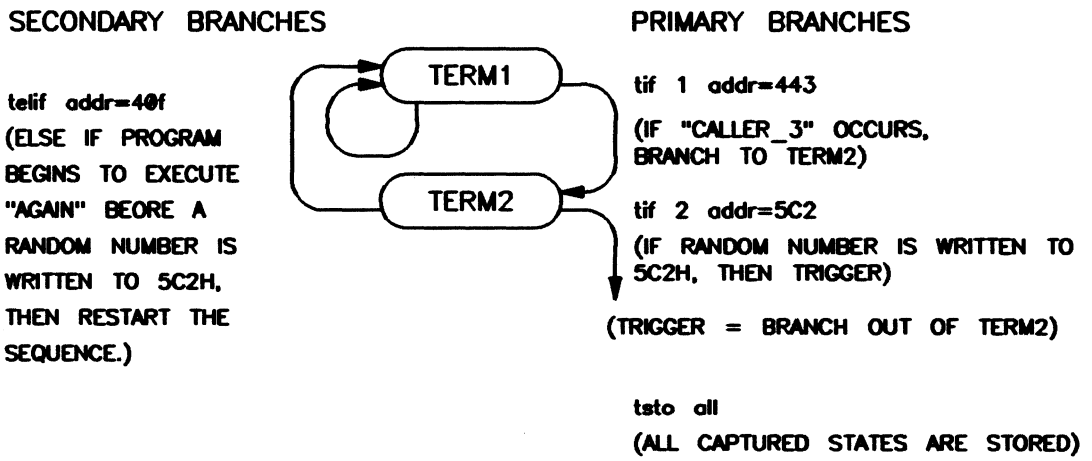


Figure 2-4. Specifying Primary and Secondary Branches

The sequencer works like this: After the trace is started, the first sequence term searches for the CALLER_3 address. When the CALLER_3 state is found, the sequencer branches to term 2. Now, the second sequence term searches for the address 5C2H. If address 5C2H is found before the state which satisfies the secondary branch expression (the AGAIN address), the analyzer is triggered, causing the analyzer memory to be filled with states before the analyzer stops. If the AGAIN address occurs before the primary branch (in either the first or second terms), the sequencer branches back to the first sequence term. The following commands start the trace and display the trace status.

```
U>t
  Emulation trace started
U>ts
  --- Emulation Trace Status ---
  NEW User trace complete
  Arm ignored
  Trigger in memory
  Arm to trigger ?
  States 512 (512) 0..511
  Sequence term 3
  Occurrence left 1
```

The "Sequence term" line of the trace status display shows the number of the term the sequencer was in when the trace completed. Because a branch **out of the last sequence term** constitutes the trigger, the number displayed is what would be the next term (3 in the preceding example) even though that term is not defined. If the trace is halted, the sequence term number just before the halt is displayed; otherwise, the current sequence term number is displayed. If the current sequence term is changing too quickly to be read, a question mark (?) is displayed.

The "Occurrence left" line of the trace status display shows the number of occurrences remaining before the primary branch can be taken out of the current sequence term. If the occurrence left is changing too quickly to be read, a question mark (?) is displayed.

Listing the trace will result in the following display.

```
U>tl
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ---------------------------------  ---------  ---
      0    005c2         xx75H, mem write                      ---   +
      1    0044d   INSTRUCTION--opcode unavailable               0   .
      2    006fc         0446H, mem read                         0   .
      3    00446         c7ebH, opcode fetch                     0   .
      4    00448         8826H, opcode fetch                     0   .
      5    00446   JMP SHORT 040fH                               0   .
      6    0044a         009dH, opcode fetch                     0   .
      7    0040f         8axxH, opcode fetch                     1   .
      8    00410         25c7H, opcode fetch                     0   .
      9    0040f   MOV AL,BH                                     1   .
     10    00412         00ffH, opcode fetch                     0   .
     11    00411   AND AX,#00ffH                                 0   .
     12    00414         f88bH, opcode fetch                     0   .
     13    00416         fb8aH, opcode fetch                     0   .
     14    00414   MOV DI,AX                                     0   .
     15    00418         33e8H, opcode fetch                     0   .
     16    00416   MOV BH,BL                                     0   .
     17    0041a         8a00H, opcode fetch                     0   .
     18    00418   CALL NEAR PTR 044eH                           0   .
     19    0041c         8ad8H, opcode fetch                     0   .
     20    0044e         6db8H, opcode fetch                     0   .
```

Remember, the primary branch out of the last term constitutes the trigger. Also, a primary branch always advances to the next higher term. A secondary branch from any term is always made back to the first sequence term (global restart).

Inserting Sequence Terms (tsq -i)

The sequencer may have a total of 4 terms. You can insert sequence terms with the **tsq** (trace sequencer) command using the **-i** (insert) option. For example, to insert a sequence term before the second term, enter the following command.

```
U>tsq -i 2
```

Enter the **tsq** command with no options to display the resulting sequencer specification.

```
U>tsq
  tif 1 addr=443
  tif 2 any          # Inserted term.
  tif 3 addr=5c2
  tsto all
  telif never
```

You can also use the **tsq -i** command to add sequence terms. For example, to add a fourth sequence term, enter the following command.

```
U>tsq -i 4
```

Enter the **tsq** command with no options to display the resulting sequencer specification.

```
U>tsq
  tif 1 addr=443
  tif 2 any
  tif 3 addr=5c2
  tif 4 any          # Added term.
  tsto all
  telif never
```

Deleting Sequence Terms (tsq -d)

You delete sequence terms using the **-d** option to the **tsq** (trace sequencer specification) command. For example, to delete the terms which were just inserted, enter the following commands.

```
U>tsq -d 2
U>tsq -d 3
```

After a term is deleted, the remaining terms are renumbered; this is why the third term is deleted above instead of the fourth (which no longer exists after the **tsq -d 2** command). Enter the **tsq** command with no options to verify that the sequencer is as it was before inserting and deleting terms.

```
U>tsq
  tif 1 addr=443
  tif 2 addr=5c2
  tsto all
  telif addr=40f
```

Changing the Trigger Position (tp)

The preceding trace specification caused the analyzer to fill trace memory with the states which followed the trigger. The reason the trigger appears at the start of the trace list is because of the current trigger position specification. To see the current trigger position specification, enter the tp (trigger position) command with no options.

```
U>tp
  tp s
```

The trigger position default is **s**, which specifies that the trigger appears at the start of the trace. You can also specify that the trigger appear in the center of the trace with the **c** option, or that the trigger appear at the end of the trace with the **e** option; additionally, you can specify a certain number of states to appear before (**-b**) or after (**-a**) the trigger in the trace. For example, changing the trigger position so that 10 states appear before the trigger in the trace and reissuing the trace will result in the trace list which follows.

```
U>tp -b 10
U>t
  Emulation trace started
U>tl
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ----------------------------------  ---------  ---
    -11    00444         0002H, opcode fetch                  ---   .
    -10    00443   INSTRUCTION--opcode unavailable              0   .
     -9    00446         c7ebH, opcode fetch                    0   .
     -8    00448         8826H, opcode fetch                    0   .
     -7    006fc         0446H, mem write                       0   .
     -6    00448   MOV ES:BYTE PTR 0500H[DI],BL                 0   .
     -5    00449                                                0   .
     -4    0044a         009dH, opcode fetch                    0   .
     -3    0044c         c305H, opcode fetch                    0   .
     -2    0044e         6db8H, opcode fetch                    0   .
     -1    00450         264eH, opcode fetch                    0   .
      0    005c2         xx75H, mem write                       0   +
      1    0044d   RET                                          0   .
      2    006fc         0446H, mem read                        0   .
      3    00446         c7ebH, opcode fetch                    0   .
      4    00448         8826H, opcode fetch                    0   .
      5    00446   JMP SHORT 040fH                              0   .
      6    0044a         009dH, opcode fetch                    0   .
      7    0040f         8axxH, opcode fetch                    1   .
      8    00410         25c7H, opcode fetch                    0   .
      9    0040f   MOV AL,BH                                    1   .
U>
```

Notice that the top of the trace is not exactly 10 lines before the trigger. The actual trigger position is within +/- 1 state of the number specified if counting states or time; otherwise, the actual trigger position is within +/- 3 states of the number specified.

Tracing a Program as it Starts Up

If a background monitor is being used, you can trace the program as it starts up by breaking to background, starting the trace, and running the program as shown by the commands below.

```
U>tinit
U>b
M>t
  Emulation trace started
M>r 400
U>tl -t 20
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ------------------------------------  ---------  ---
      0    00400         00b8H, opcode fetch                    ---   +
      1    00402         8e00H, opcode fetch              0.520 uS   .
      2    00400   MOV AX,#0000H                          0.160 uS   .
      3    00404         8ed8H, opcode fetch              0.400 uS   .
      4    00403   MOV DS,AX |                            0.280 uS   .
      5    00406         b8d0H, opcode fetch              0.240 uS   .
      6    00405   MOV SS,AX |                            0.160 uS   .
      7    00408         0000H, opcode fetch              0.400 uS   .
      8    00407   MOV AX,#0000H                          0.120 uS   .
      9    0040a         c08eH, opcode fetch              0.440 uS   .
     10    0040c         febcH, opcode fetch              0.520 uS   .
     11    0040a   MOV ES,AX |                            0.160 uS   .
     12    0040e         8a06H, opcode fetch              0.400 uS   .
     13    0040c   MOV SP,#06feH                          0.120 uS   .
     14    00410         25c7H, opcode fetch              0.400 uS   .
     15    0040f   MOV AL,BH                              0.280 uS   .
     16    00412         00ffH, opcode fetch              0.280 uS   .
     17    00411   AND AX,#00ffH                          0.120 uS   .
     18    00414         f88bH, opcode fetch              0.400 uS   .
     19    00416         fb8aH, opcode fetch              0.560 uS   .
U>
```

3

Accessing Full Analyzer Capability

Introduction

This chapter:

- Introduces the terms "complex configuration" and "easy configuration" to represent the analyzer configurations which respectively allow access to the full capability (as described in this chapter) and the capability provided with the easy-to-use configuration (as described in the "Getting Started" chapter).
- Describes the trace commands which are different in the "complex" configuration. Also describes how expressions are different in the "complex" configuration.
- Describes the sample program used for the examples in this chapter.
- Shows you how to configure the analyzer so that you have access to its full capability.
- Describes the sequencer upon entry into the "complex" configuration and how to reset the sequencer to this state.
- Describes the sequencer after a "simple trigger" specification.
- Shows you how to use the sequencer in the "complex" configuration.

Prerequisites

Before reading the examples in this chapter you should already know how the emulator operates. You should know what the various emulator prompts mean, and you should know how to use the emulation commands. Refer to the appropriate *Terminal Interface: Emulator User's Guide* to learn about the emulator; then, return to this manual.

You should also know how the analyzer operates in its limited capability configuration (refer to the "Getting Started" chapter).

"Easy" and "Complex" Configuration Differences

The analyzer configuration which allows you to access its full capability is called the "complex" configuration. The easy-to-use configuration (as described in the previous chapter) is called the "easy" configuration. The differences between the two configurations are as follows.

Sequence Terms and the Trigger

In the "easy" configuration, you can insert or delete terms from the sequencer, and the branch out of the last sequence term constitutes the trigger. The simple trigger command (**tg**) sets up a one term sequencer, and the expression specified with the **tg** command becomes the primary branch expression of the first sequence term.

In the "complex" configuration, there are always eight terms in the sequencer. Any of the sequence terms except the first may be specified as the *trigger term.* In the "complex" configuration, entry into the trigger term constitutes the trigger. The simple trigger command (**tg**) sets the primary branch expression of sequence term 1, and specifies the second sequence term as the trigger term.

Primary Branch Expressions

In the "easy" configuration, primary branches are always made to the next higher sequence term.

In the "complex" configuration, primary branches may be made to any sequence term.

Secondary Branch Expressions

In the "easy" trace configuration, the secondary branch expression is a global restart. In other words, the secondary branch expression applies to all sequence terms, and the branch is always back to the first sequence term.

In the "complex" configuration, secondary branch expressions may be specified for each sequence term. Also, secondary branches can be made to any sequence term.

Storage Qualifiers

In the "easy" configuration, the trace storage qualifier is "global" and applies to all sequence terms.

In the "complex" trace configuration, a storage qualifier is associated with each sequence term; however, the **tsto** command still allows you to specify storage qualifiers globally.

Complex Expressions

In the "complex" configuration, up to eight patterns and one range are used in trace commands wherever expressions were used in the "easy" configuration. Patterns and ranges are equal to "easy" configuration expressions. The additional capability allowed in the "complex" configuration is that these patterns may be used in combinations to specify more complex expressions.

Specifying Trace Patterns

Use the **help tpat** command to see how trace patterns may be specified.

```
U>help tpat

  tpat - set and display pattern resources

    tpat                                  - display all patterns
    tpat <pattern>                        - display named patterns
    tpat <pattern> <label>=<value>        - equals pattern
    tpat <pattern> <label>!=<value>       - not equals pattern
    tpat <pattern> <label>=<value> and <label>=<value>
    tpat <pattern> <label>!=<value> or <label>!=<value>

  --- VALID <pattern> NAMES ---
    p1 through p8  - defining patterns 1 through 8
  --- VALID <label> NAMES ---
    label - labels defined via tlb command
  --- NOTE ---
    the analyzer mode must be complex to use this command
```

Up to eight trace patterns can be specified with the **tpat** (trace pattern) command. The trace pattern names are **p1**, **p2**, ..., **p8**.

The expression associated with a trace pattern can be the keywords **all**, **any**, **none**, or **never**, or the expression may be trace labels equated to values (which can be ANDed together) or trace labels not equal to values (which can be ORed together). Examples of valid pattern specifications follow.

```
U>tpat p1 addr=520 and data=0xxaa and stat=mw
U>tpat p5 addr!=5c2 or data!=0xx3x or stat!=mr
```

The values which are associated with trace labels are the same as described in the "Getting Started" chapter.

Specifying a Trace Range

Use the **help trng** command to find out how the trace range resource may be specified. The range name is **r**, and **!r** specifies "not in range".

```
U>help trng

  trng - set or display range pattern

    trng                              - display range
    trng <label>=<value>..<value>     - define range

  --- VALID <label> NAMES ---
    label - labels defined via tlb command
  --- NOTE ---
    the analyzer mode must be complex to use this command
```

Again, values may be specified as described in the "Getting Started" chapter. Examples of valid range specifications follow.

```
U>trng addr=500..5ff
U>trng data=0080..008f
```

Combining Resources

The eight patterns (**p1**..**p8**), the range (**r** for "in range" or **!r** for "not in range"), and the **arm** qualifier (described in the "Making Coordinated Measurements" chapter) are grouped into the two sets shown below.

Set 1: **p1**, **p2**, **p3**, **p4**, **r**, and **!r**.

Set 2: **p5**, **p6**, **p7**, **p8**, and **arm**.

Resources within a set may be combined using one of the intraset operators, | (OR) or ~ (NOR). Examples of some valid and invalid intraset combinations follow.

```
U>tsto p1 | p2 | p3 | r
U>tsto p5 ~ p6 ~ arm
U>tsto p1 | p2 ~ p3
!ERROR 1249! Invalid qualifier expression: ~ p3
```

This expression is invalid because you cannot use both | (OR) and ~ (NOR) operators within the same set.

```
U>tsto p1 ~ p2 ~ p5
!ERROR 1249! Invalid qualifier expression: p5
```

This expression is invalid because you cannot combine resources from different sets with the | (OR) or ~ (NOR) operators.

The two sets can be combined with the **and** and **or** interset (between set) operators. Interset operators are also called global set operators.

The intraset (within a set) operators (~, |) are evaluated first; then, the interset operators are evaluated. You cannot use interset operators on patterns in the same set. Examples of some valid and invalid combinations of the two sets follow.

```
U>tsto p1 ~ p2 and p5 | p6
U>tsto p3 | p4 | !r or p7
U>tsto p8 ~ arm and p1 ~ p2
U>tsto p1 and p2
!ERROR 1249! Invalid qualifier expression: p2
```

This set combination is invalid because **p1** and **p2** are in the same set.

Note that "p1 ~ p1" is allowed; this type of expression may occasionally be useful if you are running out of pattern resources and wish to specify a logical NOT of some existing pattern. For example, consider the following commands:

```
tpat p1 addr=0
tif 1 p1
tif 2 p1 ~ p1
```

The primary branch of term 2 will be taken when "addr!=0".

Limitations of Combining Patterns

Only the OR (|) and NOR (~) logical operators are available as intraset operators. However, you can create the AND and NAND operators by applying DeMorgan's law (the "/" character is used to represent a logical NOT):

```
AND        A and B = /(/A and /B)        NOR
NAND    /(A and B) = /A or /B            OR
```

For example, suppose you want to specify the following storage qualifier:

```
U>tsto p1 & p2 or p5 & p6
!ERROR 1241! Invalid qualifier resource or operator: &
```

The error occurs because the & operator is not a valid intraset operator. If the specifications for the trace patterns are:

```
tpat p1 addr=5ff
tpat p2 data=39xx and stat=mw
tpat p5 addr=500
tpat p6 data=0xx39 and stat=mw
```

You can enter an equivalent expression to the one which caused the error by making the following changes to the trace patterns and using the NOR (~) operator in the **tsto** command.

```
U>tpat p1 addr!=5ff
U>tpat p2 data!=39xx or stat!=mw
U>tpat p5 addr!=500
U>tpat p6 data!=0xx39 or stat!=mw
U>tsto p1 ~ p2 or p5 ~ p6
```

Commands that Change in the "Complex" Configuration

Changing the trace configuration will affect the following trace commands. In a few cases, the options of the affected trace command are different. However, in most cases, the only difference is that complex expressions are used where easy configuration expressions were used before.

- **tcq** (Trace Count Qualifier) -- Options are the same. Complex expressions are used.
- **telif** (Secondary Branch Expressions) -- Different options. In the "easy" configuration, the secondary branch expression is a "global restart". It applies to all sequence terms and causes branches back to the first sequence term. In the "complex" configuration, you can specify secondary branch expressions for each sequence term and the branch may be to any sequence term. Complex expressions are used.
- **tg** (Simple Trigger) -- Options are the same. Complex expressions are used.
- **tif** (Primary Branch Expressions) -- Different options. In the "easy" configuration, primary branches are always to the next sequence term. In the "complex" configuration, primary branches may be to any sequence term. (The number of the destination term must be specified before the occurrence count.) Complex expressions are used.

- **tpq** (Trace Prestore Qualifier) -- Options are the same. Complex expressions are used.
- **tsq** (Trace Sequencer Specification) -- Different options. In the "easy" configuration, you can insert or delete terms. A branch out of the last sequencer term constitutes the trigger. In the "complex" configuration, you cannot insert or delete sequence terms. Eight terms are always in the sequencer. Any term but the first can be designated as the trigger term. (No expressions are involved.)
- **tsto** (Trace Storage Qualifier) -- Different options. In the "easy" configuration, the trace storage qualifier is global, that is, it applies to all sequence terms. In the "complex" configuration, storage qualifiers are associated with each sequence term (though you can specify that one storage qualifier applies to all terms). Complex expressions are used.

```
FILE: srnd.S              HEWLETT-PACKARD: 80186 Assembler

LOCATION OBJECT CODE LINE     SOURCE LINE

                       1 "80186"
                       2                    ORG     400H
                       3                    ASSUME  DS:ORG,ES:ORG
                       4
    0400 B80000        5 START              MOV     AX,SEG RAND_SEED
    0403 8ED8          6                    MOV     DS,AX
    0405 8ED0          7                    MOV     SS,AX
    0407 B80000        8                    MOV     AX,SEG RESULTS
    040A 8EC0          9                    MOV     ES,AX
    040C BC3A08       10                    MOV     SP,OFFSET STACK
                      11 * CX used as a counter for the random numbers written.
    040F B9FF04       12                    MOV     CX,#4FFH
    0412 8AC7         13 AGAIN              MOV     AL,BH
    0414 25FF00       14                    AND     AX,#0FFH
                      15 * DI contains the offset to the RESULTS area (3rd
                      16 * previous random number).
    0417 8BF8         17                    MOV     DI,AX
                      18 * BH contains the previous random number.
    0419 8AFB         19                    MOV     BH,BL
    041B E83E00       20                    CALL    RAND
                      21 * RAND returns the random number in AX.
                      22 * BL contains the current random number.
    041E 8AD8         23                    MOV     BL,AL
    0420 8AE7         24                    MOV     AH,BH
```

Figure 3-1. "Complex" Configuration Sample Program

```
                        25 * The instructions which follow determine which
                        26 * caller calls WRITE_NUMBER (depends on the last
                        27 * two bits of the previous random number).
0422 D0DC               28                  RCR     AH,1
0424 7207               29                  JC      ONE_THREE
0426 D0DC               30                  RCR     AH,1
0428 7216               31                  JC      CALLER_2
042A E90700             32                  JMP     CALLER_0
042D D0DC               33 ONE_THREE        RCR     AH,1
042F 7215               34                  JC      CALLER_3
0431 E90600             35                  JMP     CALLER_1
                        36 * The WRITE_NUMBER routine is called from four
                        37 * different places.  After the subroutine return,
                        38 * the program checks how many random numbers have
                        39 * been written.
0434 E83D00             40 CALLER_0         CALL    WRITE_NUMBER
0437 E90F00             41                  JMP     NEAR PTR TEST
043A E83700             42 CALLER_1         CALL    WRITE_NUMBER
043D E90900             43                  JMP     NEAR PTR TEST
0440 E83100             44 CALLER_2         CALL    WRITE_NUMBER
0443 E90300             45                  JMP     NEAR PTR TEST
0446 E82B00             46 CALLER_3         CALL    WRITE_NUMBER
0449 49                 47 TEST             DEC     CX
                        48 * If the counter is not zero, continue to write
                        49 * random numbers.
044A 75C6               50                  JNZ     AGAIN
                        51 * The counter is zero.  Sort the random numbers
                        52 * in the RESULTS area.
044C B9FF04             53                  MOV     CX,#4FFH    ; Reset counter.
                        54 * Push the "high address" and "low address"
                        55 * parameters expected by the QSORT routine.
044F B8FF05             56                  MOV     AX,OFFSET RESULTS+0FFH
0452 50                 57                  PUSH    AX
0453 B80005             58                  MOV     AX,OFFSET RESULTS
0456 50                 59                  PUSH    AX
                        60 * Call the QSORT routine.
0457 E82000             61                  CALL    NEAR PTR QSORT
045A EBB6               62                  JMP     AGAIN     ; Repeat program.
                        63

                        64 *-------------------------------------------------
                        65 * The RAND subroutine generates a pseudo-random
                        66 * number from 0-0FFH.  The result is left in
                        67 * register AX.
                        68 *-------------------------------------------------
                        69
045C B86D4E             70 RAND             MOV     AX,#4E6DH
045F 26F72E0006         71                  IMUL    RAND_SEED
0464 153903             72                  ADC     AX,#339H
0467 7301               73                  JNC     PAST_INC
0469 42                 74                  INC     DX
046A 26A30006           75 PAST_INC         MOV     RAND_SEED,AX
046E 8BC2               76                  MOV     AX,DX
0470 25FF00             77                  AND     AX,#0FFH
0473 C3                 78                  RET
                        79
```

Figure 3-1. "Complex" Config. Sample Program (Cont'd)

```
                    80 *-------------------------------------------------
                    81 * The WRITE_NUMBER subroutine writes the random
                    82 * number to the RESULTS area.  The second previous
                    83 * random number is the offset in this area.
                    84 *-------------------------------------------------
                    85
0474 26889D0005     86 WRITE_NUMBER    MOV     RESULTS[DI],BL
0479 C3             87                 RET
                    88


                    89 *-------------------------------------------------
                    90 * The QSORT subroutine is passed the high and low
                    91 * addresses of some area of bytes to be sorted on
                    92 * the stack.
                    93 *-------------------------------------------------
                    94
047A 8BEC           95 QSORT           MOV     BP,SP
047C 8B7E04         96                 MOV     DI,[BP+4]  ; DI = high index.
047F 8B7602         97                 MOV     SI,[BP+2]  ; SI = low index.
                    98
                    99 * The following section splits the area to be sorted
                   100 * into two areas.  QSORT will be called to sort each
                   101 * of these smaller areas.
                   102
                   103 * If high index is less than low index, then sort
                   104 * is done.
0482 3BFE          105 OVER            CMP     DI,SI
0484 7C3A          106                 JL      DONE
                   107 * AL = dividing value (from low index).
0486 8A04          108                 MOV     AL,[SI]
                   109 * (Increment allows DEC_HIGH loop to work first
                   110 * time through.)
0488 47            111                 INC     DI
                   112 * Move low index up until it points to a value
                   113 * greater than the dividing value.
0489 46            114 INC_LOW         INC     SI
048A 3A04          115                 CMP     AL,[SI]
048C 7E06          116                 JLE     DEC_HIGH
048E 3BFE          117                 CMP     DI,SI
0490 7E15          118                 JLE     OUT
0492 EBF5          119                 JMP     INC_LOW
                   120 * Move high index down until it points to a value
                   121 * less than or equal to the dividing value.
0494 4F            122 DEC_HIGH        DEC     DI
0495 3A05          123                 CMP     AL,[DI]
0497 7CFB          124                 JL      DEC_HIGH
                   125 * If high index is less than or equal to low index,
                   126 * the area is split; do not swap values.
0499 3BFE          127                 CMP     DI,SI
049B 7E0A          128                 JLE     OUT
                   129 * If high index is greater than low index, swap
                   130 * values and move indexes again.
049D 8A24          131                 MOV     AH,[SI]
049F 8A15          132                 MOV     DL,[DI]
04A1 8814          133                 MOV     [SI],DL
```

Figure 3-1. "Complex" Config. Sample Program (Cont'd)

```
04A3 8825          134                  MOV     [DI],AH
04A5 EBE2          135                  JMP     INC_LOW
                   136 * SI = low address (needed to swap dividing value).
04A7 8B7602        137 OUT              MOV     SI,[BP+2]
                   138 * Swap dividing value and high index value.
04AA 8A15          139                  MOV     DL,[DI]
04AC 8814          140                  MOV     [SI],DL
04AE 8805          141                  MOV     [DI],AL
                   142
                   143 * The area is now split into two smaller areas.
                   144 * The last high index value is the middle of the
                   145 * two areas.  The high and low addresses for the
                   146 * second QSORT call are pushed first.
                   147
04B0 8B5604        148                  MOV     DX,[BP+4]
04B3 52            149                  PUSH    DX      ; Push high.
04B4 47            150                  INC     DI
04B5 57            151                  PUSH    DI      ; Push middle + 1.
04B6 4F            152                  DEC     DI
04B7 4F            153                  DEC     DI
04B8 57            154                  PUSH    DI      ; Push middle - 1.
04B9 56            155                  PUSH    SI      ; Push low.
04BA E8BDFF        156                  CALL    QSORT
04BD E8BAFF        157                  CALL    QSORT
04C0 C20400        158 DONE             RET     4    ; Pop values on return.
                   159


                   160 *-------------------------------------------------
                   161 * The 256 byte long RESULTS area is where the random
                   162 * numbers are written and are the locations which
                   163 * get sorted.  The area at 600H contains the stack.
                   164 *-------------------------------------------------
                   165
                   166                  ORG     500H
                   167 * Random numbers written to this area.
0500               168 RESULTS          DBS     0FFH
                   169
                   170                  ORG     600H
                   171 * Variable used in RAND subroutine.
0600 0100          172 RAND_SEED        DW      1
0602               173                  DDS     8EH
083A               174 STACK            DWS     1          ; Stack area.
                   175                  END

Errors=    0
```

Figure 3-1. "Complex" Config. Sample Program (Cont'd)

The Sample Program

The sample program used to illustrate the use of the analyzer in the "complex" configuration is the same as the example used in the "Getting Started" chapter, except that after a certain number of random numbers are written, a quicksort routine sorts the random numbers. After the random numbers are sorted, the program runs again. The sample program listing is shown below.

Before You Can Use the Analyzer

You must map memory, load the program, and run the program as was done in the previous chapter. The only difference is that another block of emulation memory must be mapped since the stack takes up more space.

```
R>map 800..0bff eram
R>map
  # remaining number of terms    : 14
  # remaining emulation memory   : 1f000h bytes
  map  00400..007ff  eram   # term 1
  map  00800..00bff  eram   # term 2
  map  other tram
```

Switching into the "Complex" Configuration (tcf -c)

To enter the "complex" analyzer configuration, use the **-c** option to the **tcf** (trace configuration) command. This will cause the analyzer to be initialized to its default "complex" configuration state.

```
U>tcf -c
```

The **tcf -e** command will place the analyzer back into the "easy" configuration. Changing the analyzer configuration to "easy" will reset the trace pattern specifications, the trigger position, and the count and prestore qualifiers.

The Default Sequencer Specification (tsq -r)

After entering the "complex" analyzer configuration, the sequencer is in its default state. The **tsq** (trace sequencer specification) command with no options will display the sequencer.

```
U>tsq
  tif 1 any 2
  tif 2 any 3
  tif 3 any 4
  tif 4 any 5
  tif 5 any 6
  tif 6 any 7
  tif 7 any 8
  tif 8 never
  tsq -t 2
  tsto 1 all
  tsto 2 all
  tsto 3 all
  tsto 4 all
  tsto 5 all
  tsto 6 all
  tsto 7 all
  tsto 8 all
  telif 1 never
  telif 2 never
  telif 3 never
  telif 4 never
  telif 5 never
  telif 6 never
  telif 7 never
  telif 8 never
```

If the **tsq** information scrolls off your screen, you may wish to display the sequencer specifications with a combination of other display commands; for example, you could enter the **tif**, **telif**, **tsto**, and **tsq -t** commands to display the same information.

There are eight terms in the "complex" configuration sequencer. By default, the primary branch expression for each term (except term 8) is **any**, the secondary branch expression for each term is **never**, and the storage qualifier for each term is **all**. The trigger term is the second sequence term. This sequencer specification will result in the same trace data as the default sequencer specification in the "easy" configuration (except that there will be more sequencer branches after the trigger). A diagram of the default sequencer specification is shown in figure 3-2.

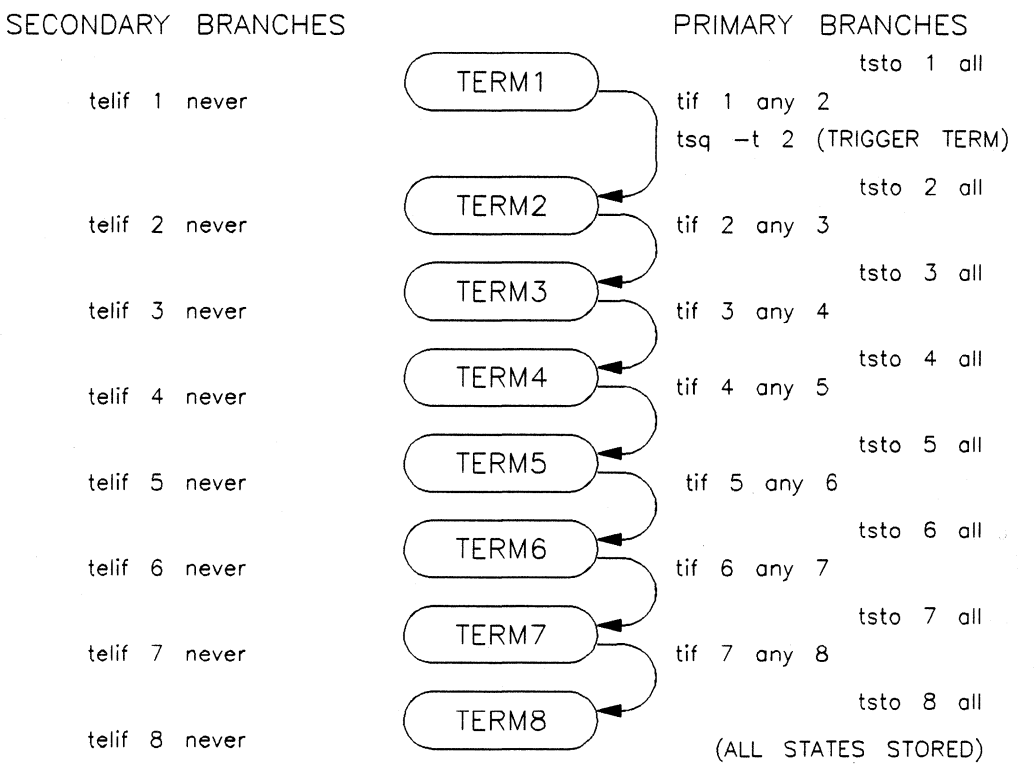


Figure 3-2. "Complex" Configuration Default Sequencer

Specifying a Simple Trigger (tg)

Using the **tg** (simple trigger) command in the "complex" configuration will cause the first two sequence terms to be modified. The pattern specified in the **tg** command becomes the primary branch expression of the first sequence term. The primary and secondary branch expressions of the second sequence term are set to **never**, and this term is specified as the trigger term. The secondary branch expression of the first sequencer term is also set to **never**.

The result of the **tg** command in the "complex" configuration is the same as in the "easy" configuration, and equivalent **tg** commands (where the pattern is the same as the "easy" configuration expression, and the storage qualifiers are the same) will yield identical traces in each of the trace configurations.

As in the "easy" configuration, the **tg** command with no options will display the primary branch expression of the first sequence term. This will only be the trigger condition when the second sequence term is the trigger term.

The commands below specify a simple trigger and display the resulting sequencer. A diagram of this sequencer specification is shown in figure 3-3.

```
U>tpat p1 addr=412
U>tg p1
U>tsq
  tif 1 p1 2
  tif 2 never
  tif 3 any 4
  tif 4 any 5
  tif 5 any 6
  tif 6 any 7
  tif 7 any 8
  tif 8 never
  tsq -t 2
  tsto 1 all
  tsto 2 all
  tsto 3 all
  tsto 4 all
  tsto 5 all
  tsto 6 all
  tsto 7 all
  tsto 8 all
  telif 1 never
  telif 2 never
  telif 3 never
  telif 4 never
  telif 5 never
  telif 6 never
  telif 7 never
  telif 8 never
```

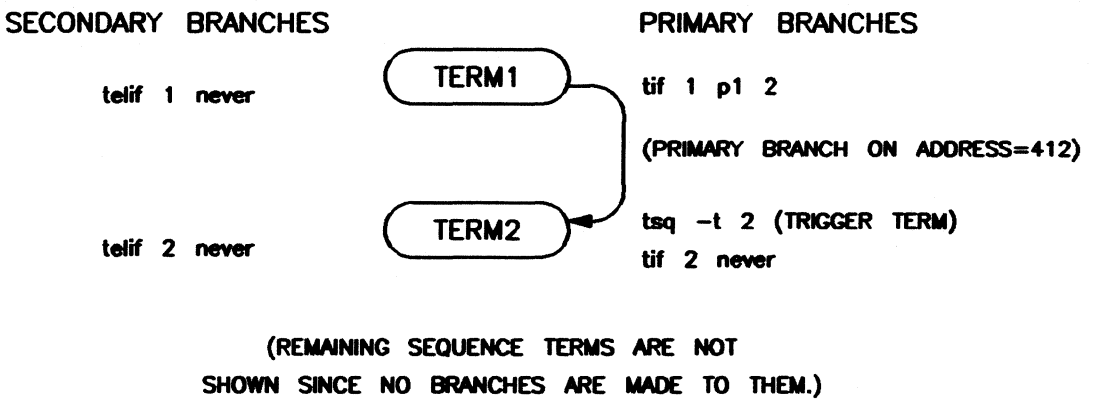


Figure 3-3. Simple Trigger in "Complex" Configuration

Using the Sequencer in the "Complex" Configuration

This section contains three examples of setting up the sequencer:

- The first example shows the general steps to follow when setting up the sequencer in the complex configuration. Labels from a hypothetical program are used to illustrate the steps involved.
- The second example shows how to set up the sequencer to trace "windows" of program activity. The sequencer is set up to trace activity in the RAND subroutine of this chapter's sample program.
- The third example shows how to use the sequencer to isolate and trace specific conditions. The analyzer is used to find the cause of a "bug" in this chapter's sample program.

Hints to Make Setting Up the Sequencer Easy

When you become experienced at using the "complex" configuration, you will be able to simply enter the trace commands for the measurement you want. Until then, following the steps listed below may make it easier for you to set up the sequencer.

1. Write down the sequencer algorithm.
2. Draw the sequencer diagram.
3. Define the trace patterns (**tpat**) and range (**trng**).
4. Specify the primary and secondary branch expressions (**tif**, **telif**).
5. Specify the trigger term (**tsq -t X**)
6. Specify the storage qualifiers (**tsto**).

Generally, you will always follow steps 3 through 6 when setting up the sequencer in the "complex" configuration. In reality, you will probably perform steps 1 and 2 at the same time, but here the algorithm is explained before the sequencer diagram is presented. Once you become experienced with how the sequencer works, you may be able to visualize steps 1 and 2 without having to write anything down.

Write Down Sequencer Algorithm

It is a good idea to write down what you want the sequencer to do. A sequence term can be used to "search" for some trace state; this is a sequence term with a primary branch expression but no secondary branch expression.

A sequence term can also be used for conditional branching; this is a sequence term with both primary and secondary branch expressions. If some trace state occurs, then go to sequence term X (primary branch). Else, if another trace state occurs before the first, go to term Y (secondary branch).

Either branch may be to any sequence term. If a state satisfies both the primary and secondary branch expressions, the primary

branch will be taken. Also, occurrence counts may only be specified with primary branch expressions.

The following examples are based on a hypothetical program whose flowchart is shown in figure 3-4.

Suppose there is a problem in the hypothetical program. You can identify two situations which cause this problem, but you are not quite sure as to why the problem occurs, and you would like to trace the program execution around either of these situations.

The first situation which causes the problem is when TRIG_STATE_1 occurs in PROCESS_1. The second situation is when TRIG_STATE_2 occurs in PROCESS_2 (which may or may not be called after PROCESS_1). Either state can occur in both processes and in other processes in the program loop; the

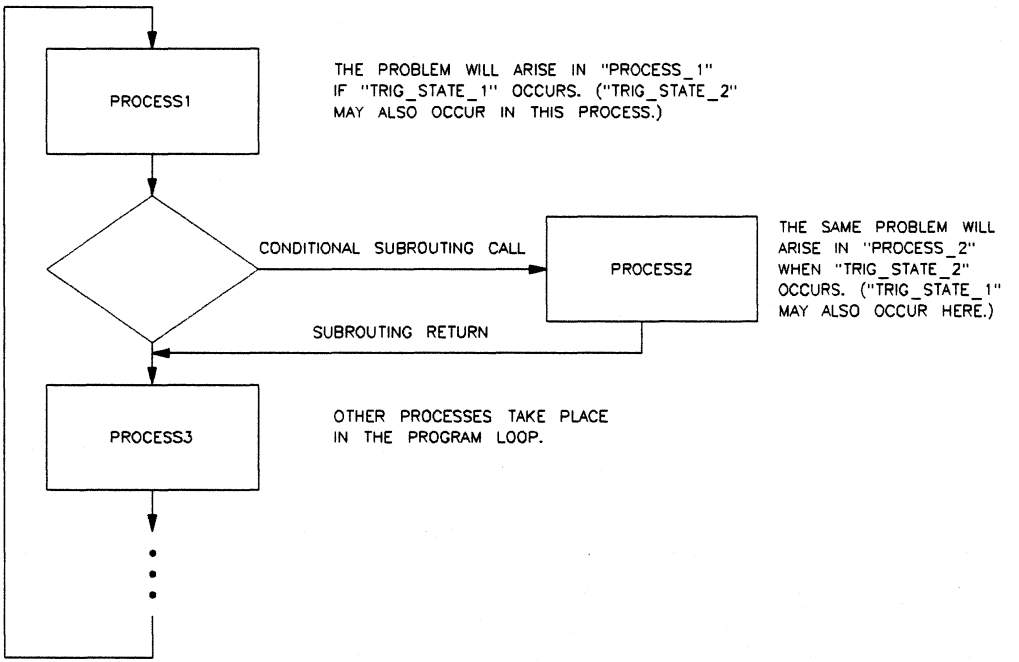


Figure 3-4. Flowchart of Hypothetical Program

problem will only arise when the specific state occurs in the specific process. The sequencer should take the following steps.

Step 1: First of all, you want the sequencer to search for PROCESS_1.

Step 2: After PROCESS_1 is found, you want the sequencer to search for TRIG_STATE_1 until PROCESS_1 exits. If TRIG_STATE_1 is found before PROCESS_1_EXIT, the sequencer should trigger the analyzer. If PROCESS_1 exits before TRIG_STATE_1 is found, the sequencer should go on and search for the next problem situation.

Step 3: After PROCESS_1 exits, you want to search for PROCESS_2. If PROCESS_3 occurs first, then you know PROCESS_2 was not called, and the problem situation did not occur in this loop of the program. The sequencer should go back and search for the next occurrence of PROCESS_1. If PROCESS_2 is found before PROCESS_3, the sequencer should go on and look for the state which identifies the problem in PROCESS_2.

Step 4: If PROCESS_2 is called, you want to search for TRIG_STATE_2. If PROCESS_3 occurs before TRIG_STATE_2, you know PROCESS_2 has exited and that the problem situation did not occur in this loop of the program. The sequencer should go back and search for the next occurrence of PROCESS_1. If TRIG_STATE_2 is found before PROCESS_3, the sequencer should trigger the analyzer.

Step 5: If the trigger condition is found in steps 2 or 4, the sequencer should trigger the analyzer by branching to the trigger term. There should be no branches out of the trigger term.

A pseudo-code algorithm of the sequencer follows.

```
Term_1:  If (PROCESS_1 occurs)
              Then go to Term_2.
Term_2:  If (TRIG_STATE_1 occurs before PROCESS_1_EXIT)
              Then trigger the analyzer, i.e., go to Term_5.
         Else if (PROCESS_1_EXIT occurs before TRIG_STATE_1)
              Then go to Term_3.
Term_3:  If (PROCESS_2 occurs before PROCESS_3)
              Then go to Term_4.
         Else if (PROCESS_3 occurs before PROCESS_2)
              Then go to Term_1.
Term_4:  If (TRIG_STATE_2 occurs before PROCESS_3)
              Then trigger the analyzer, i.e., go to Term_5.
         Else if (PROCESS_3 occurs before TRIG_STATE_2)
              Then go Term_1.
Term_5:  Analyzer is triggered on entry.
         No branches are made from this term.
```

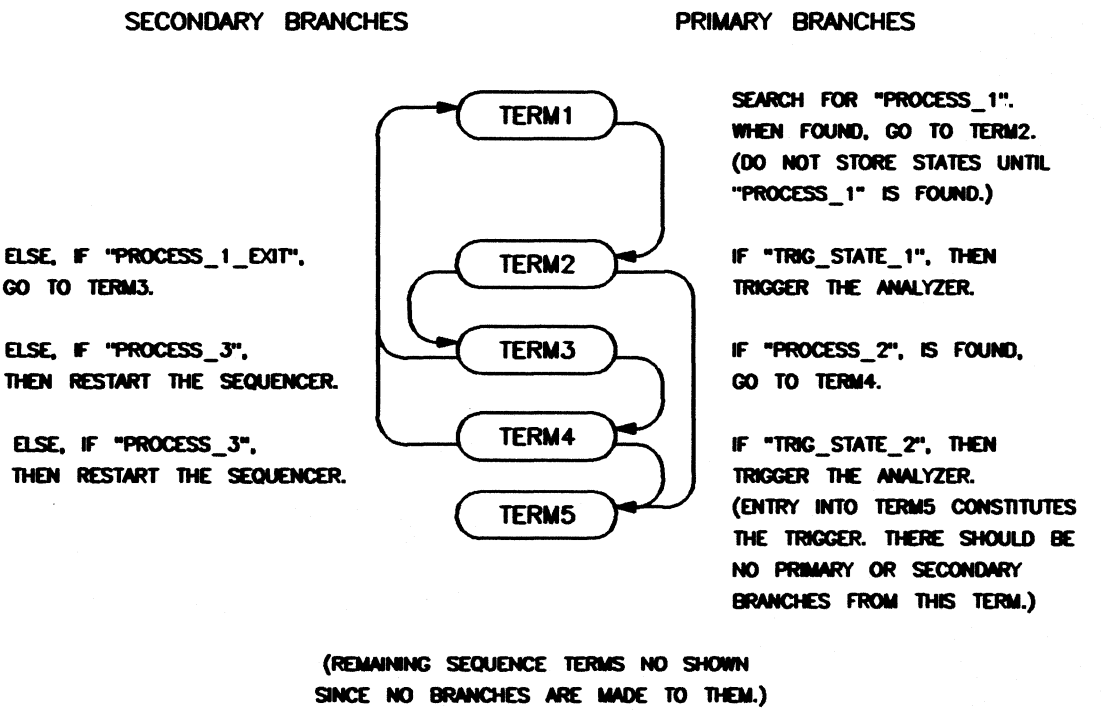


Figure 3-5. Drawing the Sequencer Diagram

Draw Sequencer Diagram

After you have listed (or while you are listing) the steps you want the sequencer to take, draw a state diagram of the sequencer as it would follow those steps. For example, the sequencer diagram for the steps listed above is shown in figure 3-5.

Define the Trace Patterns

When you know which states the sequencer is to look for, specify those states in trace patterns. Consider whether or not you will be using global set operators (**and** or **or**) with any of the patterns; if so, make sure those patterns are in different sets. Below are the **tpat** specifications to be used in the sequencer above.

```
U>tpat p1 addr=448                                  # PROCESS_1.
U>tpat p2 addr=5ff and data=0f7xx and stat=mw       # TRIG_STATE_1.
U>tpat p3 addr=490                                  # PROCESS_1_EXIT.
U>tpat p4 addr=4c2                                  # PROCESS_2.
U>tpat p5 addr=4f0                                  # PROCESS_3.
U>tpat p6 addr=5fe and data=0xxf7 and stat=mw       # TRIG_STATE_2.
```

Specify Primary and Secondary Branch Expressions

After the trace patterns are defined, you are ready to specify the primary and secondary branch expressions of the sequence terms using the **tif** and **telif** commands.

```
U>tif 1 p1
U>tif 2 p2 5
U>telif 2 p3 3
U>tif 3 p4
U>telif 3 p5 1
U>tif 4 p6
U>telif 4 p5 1
U>tif 5 never
U>telif 5 never
```

Specify the Trigger Term

From the sequencer diagram in figure 3-4, you can see that entry into the fifth term constitutes the trigger. The trigger term is specified with the **-t** option to the **tsq** command as shown below.

```
U>tsq -t 5
```

Specify Storage Qualifiers

Since each sequence term may have a storage qualifier, storage qualifier specification is part of the sequencer setup. Suppose, in the example above, that you do not wish to store states while searching for PROCESS_1 but that you wish to store all states after PROCESS_1 is found. The commands below will do this. Remember, states which cause sequencer branches are stored regardless of the trace storage qualifier.

```
U>tsto all
U>tsto 1 none
```

The command which follows will cause the trigger state to appear in the center of the trace.

```
U>tp c
```

To view the resulting sequencer setup, enter the **tsq** command with no options.

```
U>tsq
  tif 1 p1 2
  tif 2 p2 5
  tif 3 p4 4
  tif 4 p6 5
  tif 5 never
  tif 6 any 7
  tif 7 any 8
  tif 8 never
  tsq -t 5
  tsto 1 none
  tsto 2 all
  tsto 3 all
  tsto 4 all
  tsto 5 all
  tsto 6 all
  tsto 7 all
  tsto 8 all
  telif 1 never
  telif 2 p3 3
  telif 3 p5 1
  telif 4 p5 1
  telif 5 never
  telif 6 never
  telif 7 never
  telif 8 never
```

Tracing "Windows" of Activity

One common use for the "complex" configuration sequencer is to trace "windows" of execution or, perhaps, to eliminate "windows" of execution from traces. For example, suppose you wish to trace only the execution within a certain range of addresses. These addresses could be a subroutine or perhaps they are just the addresses of instructions in which you are interested.

A simple windowing sequencer specification would consist of a window enable term, a window disable term, and perhaps a trigger term (if you wish to trigger on a condition other than the enable or disable terms). Only the states which occur between the window enable condition and the window disable condition are stored.

To trace only the execution of the sample program's RAND subroutine, you would set up the sequencer specification so that the first address of the RAND subroutine is the window enable term and the address of the RAND subroutine's "return" instruction is the window disable term. Suppose also that you wish to trigger when the QSORT routine is called. The diagram of the sequencer to do this is shown in figure 3-6.

Enter the following commands to set up the sequencer. First of all, reset the sequencer.

```
U>tsq -r
```

Next, equate the addresses to be used in the sequencer branch expressions to easily recognizable names. The address of the window enable condition, the first address of the RAND subroutine, is 45CH. The address of the window disable condition, the RAND subroutine's "return" instruction, is 473H. The address of the trigger condition, the address of the call to QSORT, is 457H. Use the **equ** command, as shown below, to specify the equates.

```
U>equ Rand=45c
U>equ RandRet=473
U>equ QsortCall=457
```

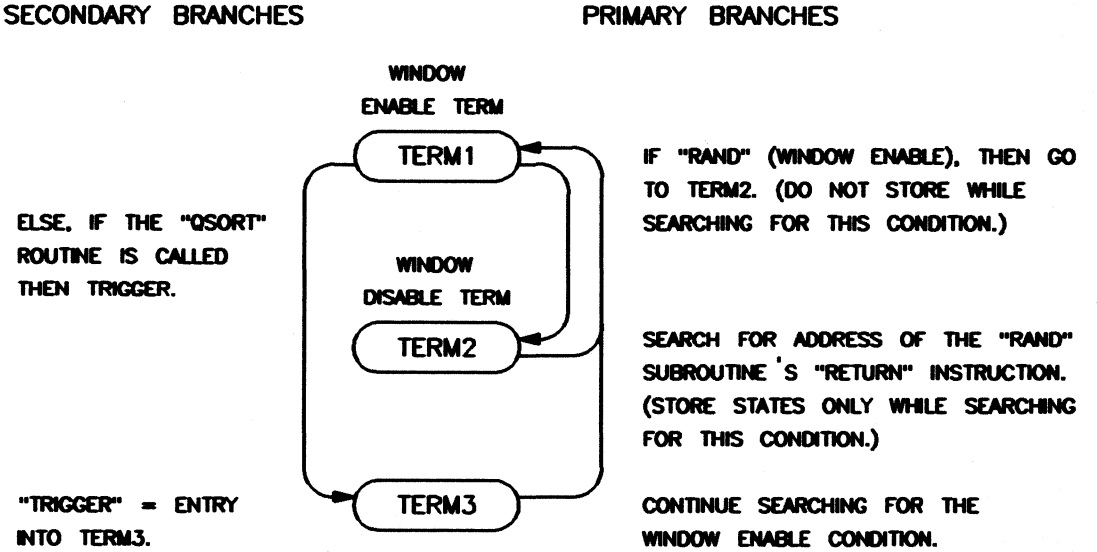


Figure 3-6. Tracing a "Window" of Activity

Specify trace patterns that equal these addresses.

```
U>tpat p1 addr=Rand          # WINDOW ENABLE.
U>tpat p2 addr=RandRet       # WINDOW DISABLE.
U>tpat p3 addr=QsortCall     # TRIGGER CONDITION.
```

Specify the primary and secondary branch expressions, and specify the trigger term.

```
U>tif 1 p1
U>telif 1 p3 3
U>tif 2 p2 1
U>tif 3 p1 1 2
U>tsq -t 3
```

Notice that the primary branch expression of the trigger term (3) is two occurrences of the Rand address. Ordinarily, you might expect to use any state as the condition on which to continue searching for the window enable. However, since the RAND subroutine

is located after the QSORT call, prefetches from the Rand address would be interpreted as window enable conditions. Two prefetches from the Rand address occur: one before the QSORT call, and one after. The primary branch condition of the trigger term causes the sequencer to continue searching for the window enable condition after the two prefetches from the Rand address.

Specify the storage qualifiers so that states are stored only while searching for the window disable condition. The first command below specifies all storage qualifiers to be **none**. The second command specifies that all states be stored while searching for the window disable condition.

```
U>tsto none
U>tsto 2 all
```

Enter the following commands to specify that time be counted (so that the count column in the trace contains useful information) and to place the trigger position 10 states below the top of the trace.

```
U>tcq time
U>tp -b 10
```

Enter the **tsq** command with no options to display the sequencer specification.

```
U>tsq
  tif 1 p1 2
  tif 2 p2 1
  tif 3 p1 1 2
  tif 4 any 5
  tif 5 any 6
  tif 6 any 7
  tif 7 any 8
  tif 8 never
  tsq -t 3
  tsto 1 none
  tsto 2 all
  tsto 3 none
  tsto 4 none
  tsto 5 none
  tsto 6 none
  tsto 7 none
  tsto 8 none
  telif 1 p3 3
  telif 2 never
  telif 3 never
  telif 4 never
  telif 5 never
  telif 6 never
  telif 7 never
  telif 8 never
```

Starting the trace, waiting for the measurement to complete, and displaying the trace will result in the following information.

```
U>t
  Emulation trace started
U>tl -t 50
   Line    addr,H   8018x mnemonic,H                          count,R   seq
  -----    ------   ------------------------------------  ---------   ---
    -11    0046c          0600H, opcode fetch                    ---    .
    -10    0046a    INSTRUCTION--opcode unavailable         0.120 uS    .
     -9    0046b    INSTRUCTION--opcode unavailable         0.160 uS    .
     -8    0046e          c28bH, opcode fetch               0.280 uS    .
     -7    00600          0f39H, mem write                  0.640 uS    .
     -6    0046e    MOV AX,DX                               0.160 uS    .
     -5    00470          ff25H, opcode fetch               0.400 uS    .
     -4    00472          c300H, opcode fetch               0.560 uS    .
     -3    00470    AND AX,#00ffH                           0.120 uS    .
     -2    00474          8826H, opcode fetch               0.400 uS    .
     -1    00473    RET                                     0.280 uS    +
      0    00457    INSTRUCTION--opcode unavailable        22.80 uS     +
      1    0045c          6db8H, opcode fetch              23.70 mS     +
      2    0045c          6db8H, opcode fetch               5.440 uS    +
      3    00838          041eH, mem write                  0.520 uS    .
      4    0045c    MOV AX,#4e6dH                           0.160 uS    .
      5    0045e          264eH, opcode fetch               0.400 uS    .
      6    00460          2ef7H, opcode fetch               0.560 uS    .
      7    0045f    IMUL ES:WORD PTR 0600H                  0.240 uS    .
      8    00462          0600H, opcode fetch               0.280 uS    .
      9    00460                                            0.120 uS    .
     10    00464          3915H, opcode fetch               0.440 uS    .
     11    00600          0f7fH, mem read                   0.800 uS    .
     12    00466          7303H, opcode fetch               0.560 uS    .
     13    00468          4201H, opcode fetch               0.520 uS    .
     14    00464    ADC AX,#0339H                           3.920 uS    .
     15    00467    JAE SHORT 046aH                         0.560 uS    .
     16    0046a          a326H, opcode fetch               0.280 uS    .
     17    0046a          a326H, opcode fetch               0.960 uS    .
     18    0046c          0600H, opcode fetch               0.520 uS    .
     19    0046a    MOV ES:0600H,AX                         0.120 uS    .
     20    0046b                                            0.160 uS    .
     21    0046e          c28bH, opcode fetch               0.280 uS    .
     22    00600          4e4dH, mem write                  0.680 uS    .
     23    0046e    MOV AX,DX                               0.120 uS    .
     24    00470          ff25H, opcode fetch               0.400 uS    .
     25    00472          c300H, opcode fetch               0.560 uS    .
     26    00470    AND AX,#00ffH                           0.120 uS    .
     27    00474          8826H, opcode fetch               0.400 uS    .
     28    00473    RET                                     0.280 uS    +
     29    0045c          6db8H, opcode fetch              21.72 uS     +
     30    00838          041eH, mem write                  0.520 uS    .
     31    0045c    MOV AX,#4e6dH                           0.160 uS    .
     32    0045e          264eH, opcode fetch               0.400 uS    .
     33    00460          2ef7H, opcode fetch               0.520 uS    .
     34    0045f    IMUL ES:WORD PTR 0600H                  0.280 uS    .
     35    00462          0600H, opcode fetch               0.280 uS    .
     36    00460                                            0.120 uS    .
     37    00464          3915H, opcode fetch               0.400 uS    .
     38    00600          4e4dH, mem read                   0.840 uS    .
```

Isolating and Tracing Specific Conditions

There is a "bug" in this chapter's sample program. Occasionally, after the 256 bytes of the RESULTS area have been sorted by the QSORT subroutine, you will see a byte out of order in the last eight or so bytes of the area. You can see what happens by setting software breakpoints before and after the QSORT routine is executed, running the program, and displaying memory.

First of all, break to the monitor.

```
U>b
M>
```

Now, define a macro called **sort** which will:

- Set a breakpoint at an address inside the QSORT subroutine, say 489H (instead of the first couple addresses of the routine so that prefetches at the end of the WRITE_NUMBER routine are not interpreted as entries into QSORT).
- Run the program until that breakpoint is hit (so you know the contents in the RESULTS area are about to be sorted).
- Set another breakpoint at the AGAIN address.
- Run the program until the AGAIN address is hit (the contents of the RESULTS area should be sorted at this point).
- Display the contents of the results area.

The following **mac** command accomplishes the items listed above.

```
M>mac sort={bp -e 489;r;w 1;bp -e 412;r;w 1;m -db 500..5ff}
```

Enable software breakpoints with the **bc** (emulator break conditions) command, and execute the sort macro.

```
M>bc -e bp
M>bp 489
M>bp 412
M>sort
  # bp -e 489;r;w 1;bp -e 412;r;w 1;m -db 500..5ff
  # waiting for 1 second....
  # waiting for 1 second....
  00500..0050f  80 80 81 83 83 85 88 89 8e 8f 8f 92 92 92 93 94
  00510..0051f  95 97 97 99 9a 9a 9b 9b 9b 9d 9d 9d 9d a0 a0 a0
  00520..0052f  a1 a2 a2 a2 a4 a5 a6 a8 a8 aa aa ab ac ad af af
  00530..0053f  af b3 b4 b4 b4 b6 b7 b7 b7 b9 ba bb bb bc be c0
  00540..0054f  c0 c1 c2 c2 c3 c4 c7 c7 c8 c8 c9 ca cc cd cd d0
  00550..0055f  d1 d1 d2 d2 d3 d4 d4 d6 d8 d9 d9 db dc df e0 e1
  00560..0056f  e2 e4 e4 e5 e6 e6 e6 e8 ea ea ec ec f0 f0 f0 f2
  00570..0057f  f4 f4 f4 f6 f6 f6 f8 fb fc fd fd fe 01 02 03 04
  00580..0058f  06 07 07 08 09 0b 0e 0e 11 13 13 14 15 18 18 19
  00590..0059f  1e 1e 20 21 22 23 24 25 26 26 27 28 28 2c 2c 2c
  005a0..005af  2c 2f 2f 31 31 32 32 32 33 33 34 35 35 36 37 37
  005b0..005bf  3e 3e 3e 3f 41 43 44 45 46 46 47 47 48 48 48 49
  005c0..005cf  4a 4a 4b 4b 4c 4d 4d 4d 4d 4d 4e 4e 4f 50 52 55
  005d0..005df  56 56 56 57 58 58 59 5a 5a 5c 5e 5f 61 61 63 66
  005e0..005ef  67 68 68 69 6a 6a 6a 6b 6b 6c 6d 6e 6e 6f 6f 6f
  005f0..005ff  70 70 71 73 74 78 7a 7a 7b 7c 7c 7d 7e 7f 7f 39
!ASYNC_STAT  615! Software breakpoint: 0000:0489
!ASYNC_STAT  615! Software breakpoint: 0000:0412
```

Look carefully at the last several bytes of the RESULTS area. You may notice that a byte is out of order. If not, execute the **sort** macro, and look at the display again. Sometimes, the program works correctly; other times, you will see a byte out of order.

The memory display shows that the QSORT routine works for the most part, which makes you suspect that the problem occurs on the final write to the RESULTS area. To verify this, you might set up the sequencer to trigger on any event, store only the address following the return from QSORT (to the main program), and prestore writes to the last eight bytes of the RESULTS area.

```
M>r
U>tg any
U>tpat p1 addr=45a
U>tsto p1
U>trng addr=5f8..5ff
U>tpq r
U>t
  Emulation trace started
U>w -m
  # waiting for analysis measurements to complete...
U>tl
   Line    addr,H  8018x mnemonic,H                        count,R  seq
  -----    ------  ------------------------------------  ---------  ---
      0    00428         1672H, opcode fetch           ---   +
      1    005fd         3dxxH, mem write                 prestore   .
      2    005fc         xx23H, mem write                 prestore   .
      3    0045a         b6ebH, opcode fetch              4.977 mS   .
      4    005ff         0bxxH, mem write                 prestore   .
      5    005ff         0bxxH, mem write                 prestore   .
      6    0045a         b6ebH, opcode fetch              23.10 mS   .
      7    0045a   JMP SHORT 0412H                        0.680 uS   .
      8    005ff         e4xxH, mem write                 prestore   .
      9    005ff         40xxH, mem write                 prestore   .
     10    0045a         b6ebH, opcode fetch              46.88 mS   .
     11    005ff         6fxxH, mem write                 prestore   .
     12    005ff         6fxxH, mem write                 prestore   .
     13    0045a         b6ebH, opcode fetch              23.33 mS   .
     14    0045a   JMP SHORT 0412H                        0.680 uS   .
     15    005ff         f8xxH, mem write                 prestore   .
     16    005f8         xx60H, mem write                 prestore   .
     17    0045a         b6ebH, opcode fetch              46.88 mS   .
     18    005ff         39xxH, mem write                 prestore   .
     19    005ff         39xxH, mem write                 prestore   .
```

From the previous trace, you see that the final writes made in the QSORT subroutine are indeed improper values for that part of the RESULTS area. Displaying additional lines of the trace shows you there are common bad values written to 5FFH. You can set up a trace to trigger on one of the common bad writes to 5FFH, and store all the states which lead up to this event. The resulting trace may show you what is wrong with the program.

The sequencer specification which follows will trigger on a write of 39xxH to 5FFH. There is nothing special about the value 39xxH; it was just a common bad value when this example was generated. You may see other bad values being written to 5FFH, and you should trace on them instead. The sequencer algorithm to capture the events which lead to a final QSORT write of 39xxH to 5FFH is listed below.

1. Search for the beginning of the QSORT routine. (The first occurrence of the INC_LOW address assures that the

QSORT routine is actually entered; this eliminates prefetches of the QSORT address from being interpreted as entry into the routine.)

2. If a write of 39H to address 5FFH occurs, this may or may not be the trigger event -- another condition must be tested (see 3). Else, if the QSORT routine exits before a write of 39H to 5FFH occurs, the trigger event has not occurred in this loop of the program; in this case, the sequencer should restart.

3. A write of 39H to 5FFH has occurred. If the QSORT routine exits without any other value being written to 5FFH, this is the trigger event. Else, if a write of some value other than 39H is made to 5FFH, the previous write

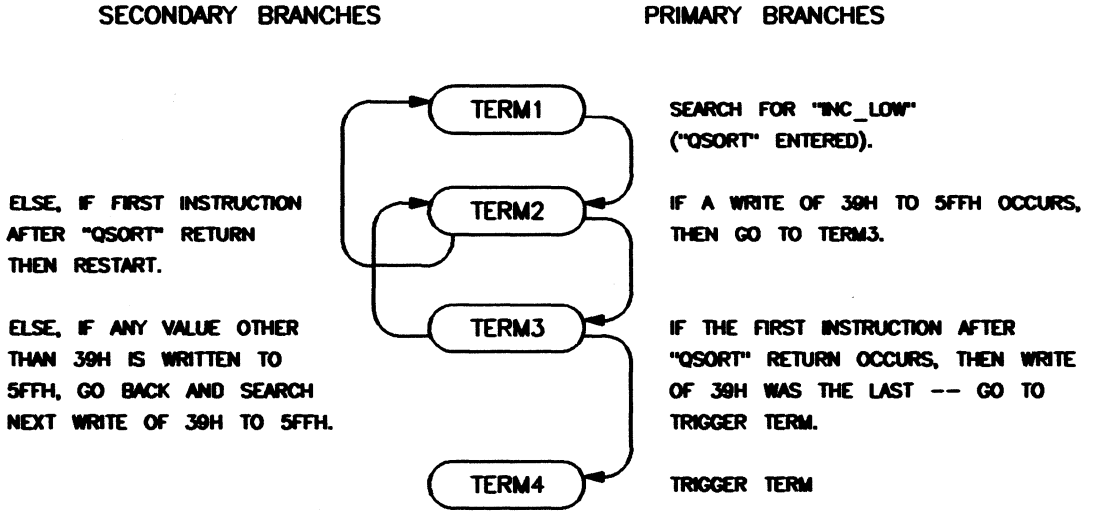


Figure 3-7. Sequencer to Isolate Sample Program Bug

is not the event to trigger on, and the sequencer should go back to searching for writes of 39H to 5FFH.

The corresponding sequencer diagram is shown in figure 3-7.

The commands to set up the sequencer, display the sequencer, issue the trace, and display the trace are shown below. Since we are interested in the instructions which occur before the trigger, the trigger position is specified such that only 10 states are stored after the trigger state.

```
U>tsq -r
U>tpq none

U>tpat p1 addr=489
U>tpat p2 addr=5ff and data=39xx and stat=mw
U>tpat p3 addr=45a
U>tpat p4 addr=5ff and stat=mw
U>tpat p5 data!=39xx

U>tif 1 p1
U>tif 2 p2
U>telif 2 p3 1
U>tif 3 p3
U>telif 3 p4 and p5 2
U>tif 4 never
U>telif 4 never
U>tsq -t 4

U>tsto none
U>tsto 2 all
U>tsto 3 all
U>tsto 4 all
```

```
U>tsq
  tif 1 p1 2
  tif 2 p2 3
  tif 3 p3 4
  tif 4 never
  tif 5 any 6
  tif 6 any 7
  tif 7 any 8
  tif 8 never
  tsq -t 4
  tsto 1 none
  tsto 2 all
  tsto 3 all
  tsto 4 all
  tsto 5 none
  tsto 6 none
  tsto 7 none
  tsto 8 none
  telif 1 never
  telif 2 p3 1
  telif 3 p4 and p5 2
  telif 4 never
  telif 5 never
  telif 6 never
  telif 7 never
  telif 8 never

U>tp -a 10
U>t
  Emulation trace started
U>w -m
  # waiting for analysis measurements to complete...
U>tl -19
   Line   addr,H  8018x mnemonic,H                          count,R  seq
  -----   ------  ---------------------------------  ---------  ---
    -19   004c2          0000H, opcode fetch           0.560 uS   .
    -18   004c0   RET #0004H                    0.120 uS    .
    -17   004c4          ffffH, opcode fetch           0.400 uS   .
    -16   00822          04c0H, mem read               0.560 uS   .
    -15   004c0          04c2H, opcode fetch           1.080 uS   .
    -14   004c2          0000H, opcode fetch           0.560 uS   .
    -13   004c0   RET #0004H                           0.120 uS   .
    -12   004c4          ffffH, opcode fetch           0.400 uS   .
    -11   00828          04c0H, mem read        0.560 uS    .
    -10   004c0          04c2H, opcode fetch           1.080 uS   .
     -9   004c2          0000H, opcode fetch           0.520 uS   .
     -8   004c0   RET #0004H                           0.160 uS   .
     -7   004c4          ffffH, opcode fetch           0.400 uS   .
     -6   0082e          04c0H, mem read               0.560 uS   .
     -5   004c0          04c2H, opcode fetch           1.080 uS   .
     -4   004c2          0000H, opcode fetch    0.520 uS    .
     -3   004c0   RET #0004H     0.160 uS    .
     -2   004c4          ffffH, opcode fetch           0.400 uS   .
     -1   00834          045aH, mem read               0.520 uS   .
      0   0045a          b6ebH, opcode fetch           1.080 uS   +
```

By continuing to list the trace lines before the trigger (tl -<line_number>), you will eventually come across the sequencer branch prior to the trigger.

```
U>tl -210
   Line    addr,H  8018x mnemonic,H                          count,R  seq
  -----    ------  ---------------------------------------  ---------  ---
   -210    0048e   CMP DI,SI                                 0.160 uS   .
   -209    00492         f5ebH, opcode fetch                 0.400 uS   .
   -208    00490   JLE SHORT 04a7H                           0.120 uS   .
   -207    00494         3a4fH, opcode fetch                 0.400 uS   .
   -206    004a7         8bxxH, opcode fetch                 0.680 uS   .
   -205    004a8         0276H, opcode fetch                 0.560 uS   .
   -204    004a7   MOV SI,WORD PTR 02H[BP]                   0.120 uS   .
   -203    004aa         158aH, opcode fetch                 0.400 uS   .
   -202    00824         05ffH, mem read                     0.840 uS   .
   -201    004aa   MOV DL,BYTE PTR [DI]                      0.400 uS   .
   -200    004ac         1488H, opcode fetch                 0.120 uS   .
   -199    00600         xx39H, mem read                     0.840 uS   .
   -198    004ac   MOV BYTE PTR [SI],DL                      0.400 uS   .
   -197    004ae         0588H, opcode fetch                 0.120 uS   .
   -196    004b0         568bH, opcode fetch                 1.080 uS   .
   -195    005ff         39xxH, mem write                    0.560 uS   +
   -194    004ae   MOV BYTE PTR [DI],AL                      0.120 uS   .
   -193    004b2         5204H, opcode fetch                 1.240 uS   .
   -192    00600         xx7eH, mem write                    0.520 uS   .
   -191    004b0   MOV DX,WORD PTR 04H[BP]                   0.160 uS   .
```

From these lines of the trace list, you can see that the instructions at addresses 4AAH and 4ACH are the ones that cause the problems. These are the instructions associated with the OUT section of the QSORT subroutine. They are used to swap the dividing value and the value at the high index after a segment of the list to be sorted is split. You can see that the high index is address 600H, which it should never be. However, looking back at the program you see that the increment of the high index so that DEC_HIGH works the first time through will cause problems when the JLE OUT instruction gets executed in the INC_LOW loop. Changing the program in the following manner will fix the problem (notice the instructions surrounded by the "#" character).

```
*-------------------------------------------------
* The QSORT subroutine is passed the high and low
* addresses of some area of bytes to be sorted on
* the stack.
*-------------------------------------------------

QSORT           MOV     BP,SP
                MOV     DI,[BP+4]  ; DI = high index.
                MOV     SI,[BP+2]  ; SI = low index.

* The following section splits the area to be sorted
* into two areas.  QSORT will be called to sort each
* of these smaller areas.

* If high index is less than low index, then sort
* is done.
OVER            CMP     DI,SI
                JL      DONE
* AL = dividing value (from low index).
                MOV     AL,[SI]
* (Increment allows DEC_HIGH loop to work first
* time through.)

*#### The following instruction deleted. ##########
*                INC     DI
*################################################

* Move low index up until it points to a value
* greater than the dividing value.
INC_LOW         INC     SI
                CMP     AL,[SI]

*#### The following instruction is changed. #######
                JLE     NEXT
*################################################

                CMP     DI,SI
                JLE     OUT
                JMP     INC_LOW

*#### The following instruction is new. ###########
NEXT            INC     DI
*################################################

* Move high index down until it points to a value
* less than or equal to the dividing value.
DEC_HIGH        DEC     DI
                CMP     AL,[DI]
                JL      DEC_HIGH
* If high index is less than or equal to low index,
* the area is split; do not swap values.
                CMP     DI,SI
                JLE     OUT
* If high index is greater than low index, swap
* values and move indexes again.
                MOV     AH,[SI]
                MOV     DL,[DI]
                MOV     [SI],DL
                MOV     [DI],AH
```

```
                JMP     INC_LOW
* SI = low address (needed to swap dividing value).
OUT             MOV     SI,[BP+2]
* Swap dividing value and high index value.
                MOV     DL,[DI]
                MOV     [SI],DL
                MOV     [DI],AL

* The area is now split into two smaller areas.
* The last high index value is the middle of the
* two areas.  The high and low addresses for the
* second QSORT call are pushed first.

                MOV     DX,[BP+4]
                PUSH    DX      ; Push high.
                INC     DI
                PUSH    DI      ; Push middle + 1.
                DEC     DI
                DEC     DI
                PUSH    DI      ; Push middle - 1.
                PUSH    SI      ; Push low.
                CALL    QSORT
                CALL    QSORT
DONE            RET     4   ; Pop values on return.
```

4

Using the External Analyzer

Introduction

Your HP 64700 Series analyzer may optionally contain 16 external trace signals. These trace lines allow you to analyze additional target system signals. The external analyzer may be configured as an extension to the emulation analyzer, as an independent state analyzer, or as an independent timing analyzer.

Note

The external analyzer's independent timing mode cannot be used from the Terminal Interface. A host computer interface is necessary to provide timing analysis. Consequently, independent timing analysis is not described in this manual. Refer to the appropriate host computer interface analyzer manual (either the *PC Interface: Analyzer User's Guide* or the *Softkey Interface: Analyzer User's Guide*).

Before You Can Use the External Analyzer

There are several things to do before you can use the external analyzer:

- Connect the analyzer probe to signals of interest in your target system.
- Specify threshold voltages of external trace signals.
- Label the external trace signals.
- Select the external analyzer mode.

Connecting the Analyzer Probe Lines to the Target System

The following steps must be taken to connect the analyzer probe to the target system:

1. Assemble the analyzer probe.
2. Connect the probe to the emulator.
3. Connect the probe wires to the target system.

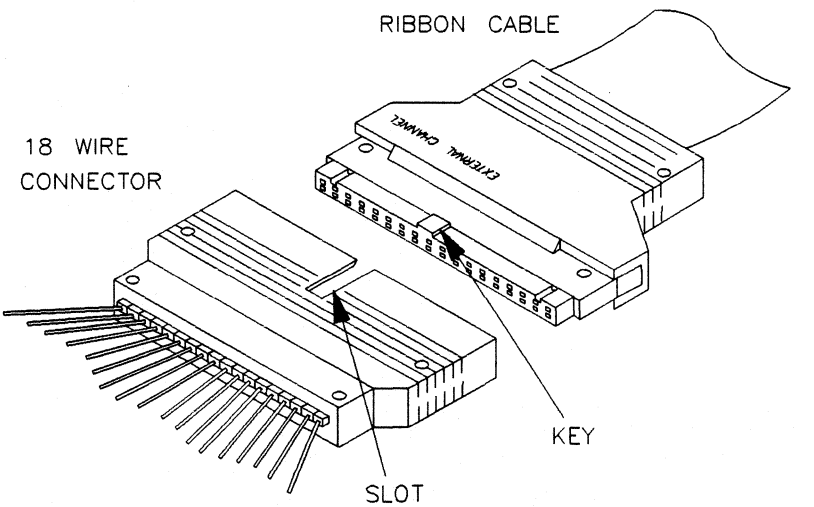


Figure 4-1. Assembling the Analyzer Probe

Assembling the Analyzer Probe

The analyzer probe is a two-piece assembly, consisting of ribbon cable and 18 probe wires (16 data channels and the J and K clock inputs) attached to a connector. Either end of the ribbon cable may be connected to the 18 wire connector, and the connectors are keyed so they may only be attached one way. Align the key of the ribbon cable connector with the slot in the 18 wire connector, and firmly press the connectors together. (See figure 4-1.)

Each of the 18 probe wires has a signal and a ground connection. Each probe wire is labeled for easy identification. Thirty-six grabbers are provided for the signal and ground connections of each of the 18 probe wires. The signal and ground connections are attached to the pin in the grabber handle. (See figure 4-2.)

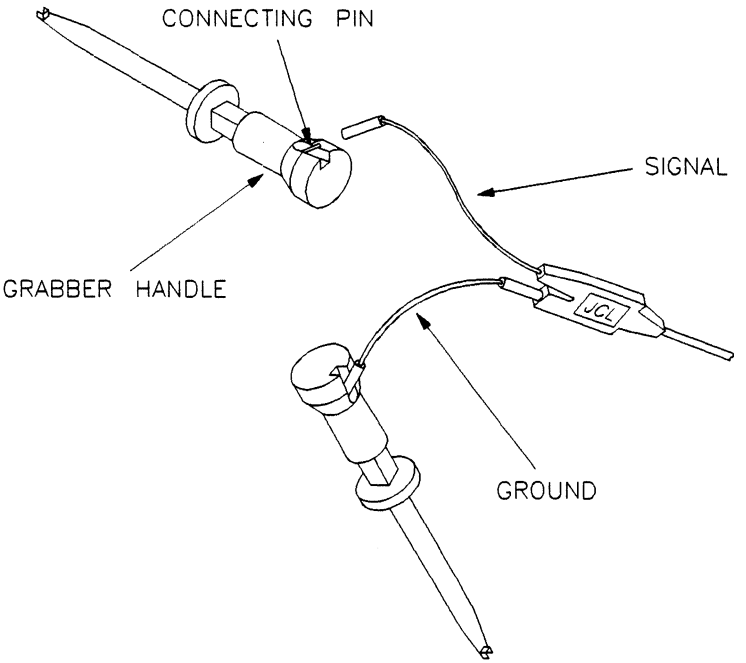


Figure 4-2. Attaching Grabbers to Probe Wires

Connecting the Probe to the Emulator

The external analyzer probe is attached to a connector under the snap-on cover in the front upper right corner of the emulator. Remove the snap-on cover by pressing the side tabs toward the center of the cover; then, pull the cover out. (See figure 4-3.)

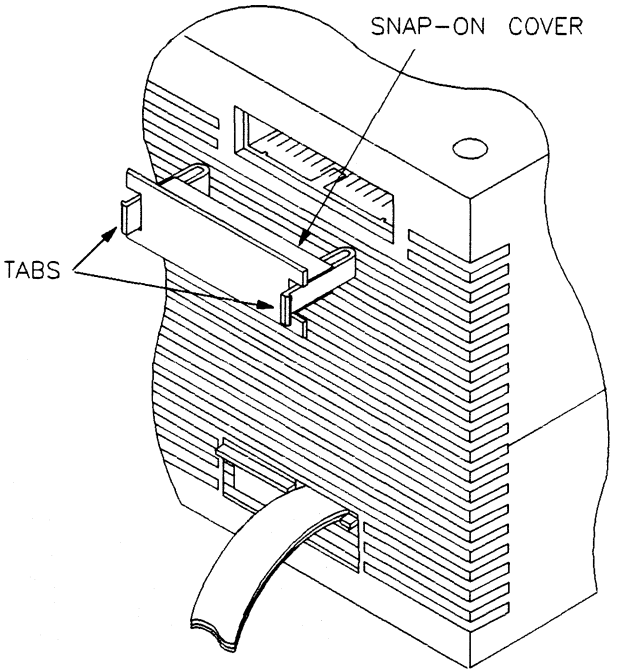


Figure 4-3. Removing Cover to Emulator Connector

Each end of the ribbon cable connector is keyed so that it can be connected to the emulator in only one way. Align the key of the ribbon cable connector with the slot in the emulator connector, and gently press the ribbon cable connector into the emulator connector. (See figure 4-4.)

Note

Check for bent connector pins before connecting the analyzer probe to the emulator.

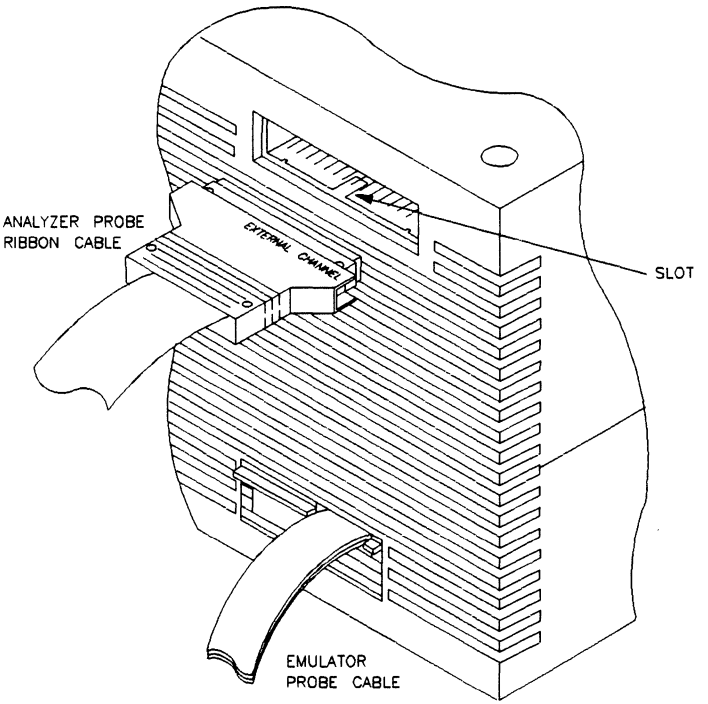


Figure 4-4. Connecting the Probe to the Emulator

Connecting Probe Wires to the Target System

Caution

Turn OFF target system power before connecting analyzer probe wires to the target system. The probe grabbers are difficult to handle with precision, and it is extremely easy to short the pins of a chip (or other connectors which are close together) with the probe wire while trying to connect it.

You can connect the grabbers to pins, connectors, wires, etc., in the target system. Pull the hilt of the grabber towards the back of the grabber handle to uncover the wire hook. When the wire hook is around the desired pin or connector, release the hilt to allow the tension of the grabber spring to hold the connection. (See figure 4-5.)

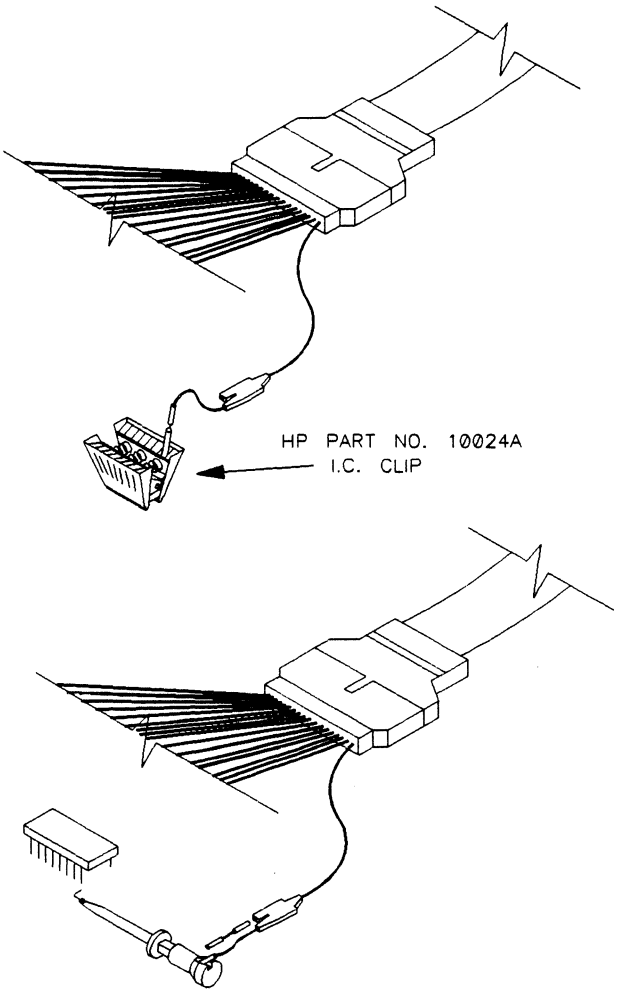


Figure 4-5. Connecting Probe to the Target System

Specifying External Trace Signal Threshold Voltages

The external analyzer probe signals are divided into two groups: the lower byte (channels 0 through 7 and the J clock), and the upper byte (channels 8 through 15 and the K clock). You can specify a threshold voltage for each of these groups. The default threshold voltages are specified with the keyword **TTL** which translates to 1.4 volts.

Use the **xtv** (threshold voltage for external trace signals) command to specify different threshold voltages. The **-l** option to **xtv** allows you to specify threshold voltages for the lower group. The **-u** option allows you to specify threshold voltages for the upper group. Voltages may be in the range from -6.4 volts to 6.35 volts (with a 50mV resolution); you may also use the keywords **TTL**, **CMOS** (which translates to 2.5 volts), or **ECL** (which translates to -1.3 volts). The command below specifies ECL threshold voltages for all external trace signals.

```
R>xtv -l ECL -h ECL
```

Defining External Trace Labels

Defining external trace labels is not something you must do before you can use the external analyzer; however, it is something you may wish to do to make specifying qualifiers easier. External trace labels may be used in any of the external analyzer modes.

One external trace label has been predefined, **xbits**. This label is associated with all 16 external trace signals. This label appears in the default trace format and listing.

If you wish to define external trace labels to further break down the external signals, use the **xtlb** (external trace label) command as shown below.

```
R>xtlb iodata 0..7
R>xtlb ioaddr 8..11
R>xtlb iostat 12..14
R>xtlb intr 15
```

You may change the trace listing format (**xtf** or **tf**) to include external trace labels after they have been defined.

Selecting the External Analyzer Mode

By default, on power-up or after trace initialization (**tinit**), the external analyzer is aligned with the emulator. In this mode, you have 16 external trace signals which are clocked with the same signal(s) as the emulation analyzer. The external trace signals may be used to capture target system signals synchronized with the emulation clock.

The external analyzer may also operate as an independent state analyzer, or it may operate as an independent timing analyzer if a host computer interface program is used. In the Terminal Interface, use the **xtmo** (external trace mode) command to select the independent state mode or to re-select the emulation mode. The **-s** option to **xtmo** is used to select the independent state analyzer mode.

```
R>xtmo -s
```

To re-select the emulation analyzer extension mode, use the **-e** option to the **xtmo** command.

```
R>xtmo -e
```

Aligned with Emulation Analyzer

When **xtmo -e** is specified (which is the default), the external analyzer becomes an extension of the emulation analyzer. In other words, they operate as one analyzer. The only external trace commands allowed in this mode are **xtv**, **xtlb**, and **xtmo**. You can, however, display the help text for the other external trace commands. The external labels may be referenced in emulation trace commands in this mode.

External trace signal data is captured on the trace clock specified in the **tck** (trace clock source) command. You should not use the external J and K signals to clock the emulation trace; however,

you may wish to use these signals to qualify the emulation trace clock (refer to the "Qualifying Clocks" section of the "Special Analyzer Topics" chapter.)

Independent State Analyzer

When **xtmo -s** is specified, the external analyzer operates as an independent state analyzer. The independent state analyzer is identical to the emulation analyzer, except that only 16 bits of analysis are available. Your HP 64700 Series emulator now contains two state analyzers; two sets of analyzer resources (trace memory, patterns, qualifiers, etc.) are available, one for the emulation analyzer and one for the independent state analyzer.

When the independent state analyzer mode is selected, you can use one analyzer to arm the other. You can specify the arm condition as a qualifier, perhaps as the trigger condition (cross-triggering). (Refer to the "Making Coordinated Measurements" chapter for more information on cross-triggering.)

Independent State Analyzer Commands (xt, xtarm, ...)

When you use the external analyzer as an independent state analyzer, a whole new set of external trace commands become available. Every trace command (except for the trace activity, **ta**, and trace initialization, **tinit**, commands) is duplicated for the independent state analyzer and prefixed with an **x**. For example, the following commands become available in the independent state mode: **xt**, **xtarm**, **xtcf**, **xtck**, **xtcq**, **xtelif**, **xtg**, **xth**, **xtif**, **xtl**, **xtlb**, **xtp**, **xtpat**, **xtpq**, **xtrng**, **xts**, **xtsck**, **xtsq**, and **xtsto**. These commands operate identically to their counterpart emulation analyzer commands.

Specifying the Independent Analyzer Clock Source

The clock source for the independent state analyzer is specified with the **xtck** (external trace clock) command. The independent state analyzer may be clocked with target system clock signals connected to the JCL and KCL external clock inputs. (Refer to the "Selecting Clock Signals" section of the "Special Analyzer Topics" chapter).

Independent Analyzer Slave Clocks

You can specify slave clocks for the external analyzer with the **xtsck** (external trace slave clock) command. Specifying slave clocks is the same for the external analyzer as it is for the emulation analyzer; refer to the "Using Slave Clocks for Demultiplexing" section of the "Special Analyzer Topics" chapter.

Independent Timing Analyzer

When **xtmo -t** is specified, the external analyzer operates as an independent timing analyzer.

Note

The external analyzer's independent timing mode cannot be used from the Terminal Interface. A host computer interface is necessary to provide timing analysis. Consequently, independent timing analysis is not described in this manual. Refer to the appropriate host computer interface analyzer manual (either the *PC Interface: Analyzer User's Guide* or the *Softkey Interface: Analyzer User's Guide*).

External Analyzer Specifications

- Threshold Accuracy = +/- 50 mV.
- Dynamic Range = +/- 10 V about threshold setting.
- Minimum Input Swing = 600 mV pp.
- Minimum Input Overdrive = 250 mV or 30% of threshold setting, whichever is greater.
- Absolute Maximum Input Voltage = +/- 40 V.
- Probe Input Resistance = 100K ohms +/- 2%.
- Probe Input Capacitance = approximately 8 pF.
- Maximum +5 Probe Current = 0.650 A.
- +5 Probe Voltage Accuracy = +5.0 +/- 5%.

External State Analyzer Specifications

- Data Setup Time = 10 nS min.
- Data Hold Time = 0 nS min.
- Qualifier Setup Time = 20 nS min.
- Qualifier Hold Time = 5 nS min.
- Minimum Clock Width = 10 nS
- Minimum Clock Period:
  - No Tagging Mode = 40 nS (25 Mhz clock).
  - Event Tagging Mode = 50 nS (20 MHz clock).
  - Time Tagging Mode = 60 nS (16 MHz clock).
- Minimum Time from Slave Clock to Master Clock = 10 nS.
- Minimum Time from Master Clock to Slave Clock = 50 nS.

5

Making Coordinated Measurements

Introduction

Coordinated measurements are measurements synchronously made in multiple emulators or analyzers. Coordinated measurements can be made between HP 64700 Series emulators which communicate over the Coordinated Measurement Bus (CMB). Coordinated measurements can also be made between an emulator and some other instrument connected to the BNC connector. These types of coordinated measurements, that is, measurements which involve signals external to an HP 64700 Series emulator, are described in the *Coordinated Measurement Bus Operating Manual.*

This chapter will describe coordinated measurements which are made internal to an HP 64700 Series emulator and which involve the HP 64700 Series analyzer. The types of coordinated measurements involving the analyzer which can be made internal to an HP 64700 series emulator are:

- Breaking into the monitor on an analyzer trigger.
- Using the emulation analyzer to arm the external analyzer (in an independent mode).
- Using the external analyzer (in an independent mode) to arm the emulation analyzer.

The last two instances above are referred to as cross-arming. When arm conditions are used to trigger an analyzer, cross-triggering takes place. Cross-triggering is a subset of cross-arming.

Arm conditions may also be used to qualify primary and secondary branches, as well as storage or prestore qualifiers.

An arm condition may not be used as a count qualifier.

Specifying an Arm Condition

By default, the analyzer is **always** armed. This means that the analyzer arm condition is always true. The **tarm** (trace arm condition) command is used to specify or display the arm condition. The **tarm** command with no options will display the current arm condition.

```
R>tarm
 tarm always
```

There are two internal signals, **trig1** and **trig2**, which may be specified as the arm condition. You can specify that the arm condition be true when one of these two signals is true (**= trig1** or **= trig2**) or when one of these two signals is false (**!= trig1** or **!= trig2**). The command below will arm the emulation analyzer when **trig1** is true.

```
R>tarm =trig1
```

The **xtarm** (external trace arm condition) command is used to specify the external analyzer arm condition when in the independent state or independent timing modes. The command below will cause the external analyzer to be armed when the **trig2** signal is false.

```
R>xtarm !=trig2
```

The keyword **arm** may be used to specify primary and secondary branch qualifiers, as well as storage or prestore qualifiers. The keyword **arm** may not be used to specify a count qualifier. For example, to trigger the emulation analyzer when it becomes armed, enter the command below.

```
R>tg arm
```

Arm Condition Status

The **ts** (trace status) command displays information on the arm condition. If the **tarm** condition is specified as **always**, the message "Arm ignored" is displayed. If the **tarm** condition is specified as one of the internal signals, either the message "Arm not received" or "Arm received" is displayed. The display indicates if the arm condition happened any time since the most recent trace started, even if it happened after the trace was halted or became complete.

The "Arm to trigger" line displays the amount of time between the arm condition and the trigger. The time displayed will be from 0.04 uS to 41.943 mS, less than 0.04 uS, or greater than 41.943 mS. If the arm signal is ignored or the trigger is not in memory, a question mark (?) is displayed.

Driving Signals When the Trigger is Found

The default condition of the analyzer specifies that neither the emulation analyzer nor the external analyzer will drive the internal **trig1** or **trig2** signals when the trigger is found. The **tgout** command is used to specify that these signals be driven when the emulation analyzer trigger is found. The **tgout** command with no options will display the signal which is currently being driven when the trigger is found (or **none** if no signal is driven when the trigger is found).

```
R>tgout
  tgout none
```

The signals which may be driven when the trigger is found are the internal signals **trig1** and **trig2**. These signals may be received by the CMB or BNC TRIGGER lines, the emulator break, or the arm condition of the external analyzer. The following command will cause the **trig1** signal to be driven when the emulation analyzer trigger is found.

```
R>tgout trig1
```

The **xtgout** command is used to specify which signal (**trig1** or **trig2**) is to be driven when the external analyzer trigger is found.

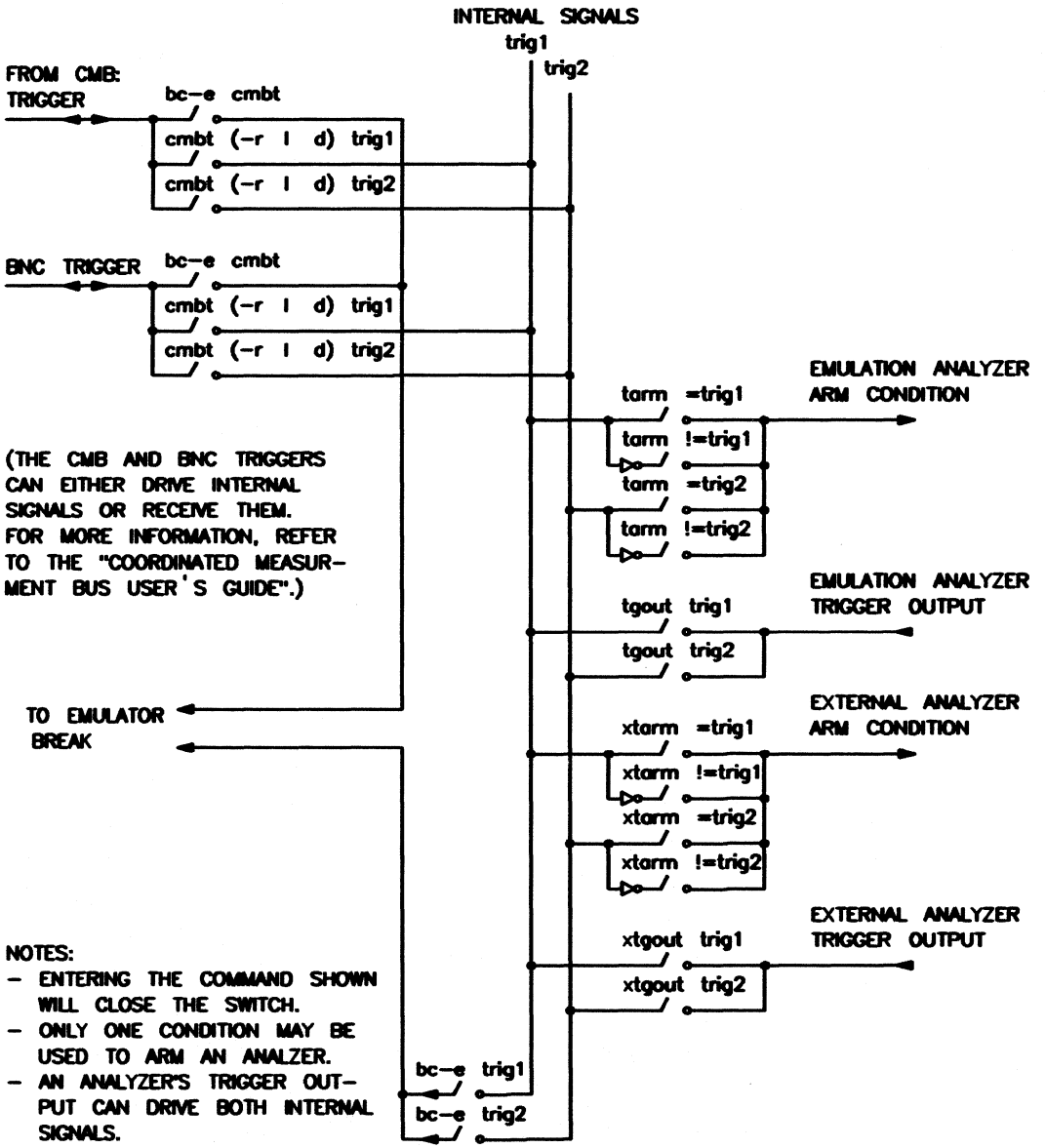


Figure 5-1. Coordinated Measurements

The keyword **none** is again used to specify that no signal should be driven. The command below specifies that **trig2** be driven when the external analyzer trigger is found.

```
R>xtgout trig2
```

A diagram of the internal signals and the commands which may be used to drive them or to arm an analyzer with them are shown in figure 5-1. This diagram is only intended to show logical connections, and does not represent actual circuitry inside the emulator.

Breaking on an Analyzer Trigger

The **bc** (break conditions) command is used to enable or disable the conditions which may break the emulator into the monitor. The internal signals **trig1** and **trig2** may be used to cause breaks to background. Therefore, to cause an analyzer trigger to break the emulator, you must specify that the analyzer drive one of the internal signals when the trigger is found, and enable a break on that internal signal. For example, the commands below will cause the emulation analyzer trigger to break the emulator.

```
R>tg any
R>tgout trig1
R>bc -e trig1
R>r 400
U>t
  Emulation trace started
U>es
  80186--Running in monitor
  --in normal mode
!ASYNC_STAT  618! trig1 break
M>
```

After the break occurs, the analyzer will stop driving the **trig** line that caused the break. Therefore, if **trig1** is used both to break and to drive the CMB TRIGGER (for example), TRIGGER will go true when the trigger is found and then will go false after the emulator breaks. However, if **trig1** is used to cause the break and

trig2 is used to drive the CMB TRIGGER, TRIGGER will stay true after the trigger until the trace is halted or until the next trace starts.

Cross-Arming Between Emulation and External Analyzers

Cross-arming between the emulation analyzer and the external analyzer is a matter of specifying that one analyzer drive one of the internal signals (**trig1** or **trig2**) and then specifying that the other analyzer be armed on that signal. For example, to cause the external analyzer to arm the emulation analyzer, the commands below are entered.

```
R>xtmo -s
R>xtgout trig1
R>tarm =trig1

R>tif 1 arm
R>tif 2 addr=40f
R>r 400

U>t
  Emulation trace started
U>xt
  External trace started
```

It is often important to start the analyzer which receives a signal before the analyzer which drives the signal. For example, if you start the analyzer which drives a signal first, the signal may already be driven before you start the analyzer which receives the signal. The receiving analyzer will most likely capture states which execute long after the condition which caused the signal to be driven.

To cause the emulation analyzer to arm the external analyzer, enter the commands below.

```
R>xtmo -s
R>tgout trig1
R>xtarm =trig1

R>xtif 1 arm
R>xtif 2 xbits=87
R>r 400

U>xt
  External trace started
U>t
  Emulation trace started
```

Cross-Triggering

Cross-triggering is a special case of cross arming in which the arm condition triggers the analyzer. The commands below will cause the emulation analyzer to trigger after it is armed by the external analyzer trigger condition.

```
R>xtmo -s
R>xtgout trig1
R>tarm =trig1
R>tg arm

U>t
  Emulation trace started
U>xt
  External trace started
```

Notes

6

Special Analyzer Topics

Introduction

This chapter describes analyzer topics which are not specifically related to the "easy" or "complex" configurations, the external analyzer, or coordinated measurements. The analyzer topics which fall into this category are listed below and described in this chapter.

- Displaying trace activity.
- Specifying the analyzer clock source.
- Slave clocks and demultiplexing.
- Saving trace specifications in command files.

Displaying Trace Activity (ta)

The **ta** (trace activity) command allows you to display the current status of the analyzer trace signals. The trace activity display allows you to view the status of trace signals at any time, regardless of whether a pending trace is completed or not. An example of the **ta** command and its output is shown below.

```
U>ta
  Pod 3          = 01100100 100?000?
  Pod 2          = 11011101 ????????
  Pod 1          = 01??1??? 00000000
  External Pod   = 0010?1?? 010??001
```

The trace signals are displayed in sets of sixteen. Pod 1 represents emulation analyzer trace signals 0 through 15 (the least significant bit is on the right). Pod 2 and Pod 3 represent emulation trace signals 16 through 31 and 32 through 48, respectively. External Pod represents the external analyzer trace signals.

A trace signal is displayed as a low (0) when it is below the threshold voltage (as specified by the **xtv** command), high (1) when it is above the threshold voltage, or moving (?).

Specifying the Analyzer Clock Source (tck)

The emulation and external analyzers have default clock source values. Use the **tck** (trace clock) command to specify or display the clock used for the emulation analyzer. The **xtck** (external trace clock) command is used to specify or display the clock used for the external analyzer. Entering the **tck** command with no options will display the current emulation trace clock specification.

```
R>tck
  tck -r L -u -s S
```

Tracing Background Execution

By default, the analyzer traces user (that is, foreground) code; this is specified by the **-u** option to the **tck** command. However, it is possible to trace background code; this is specified by the **-b** option to the **tck** command.

```
R>tck -b
R>tck
  tck -r L -b -s S
```

Notice that the user/background option is a switch in the clock specification. Changing the option as shown above does not affect the rest of the trace clock specification. It is also possible to trace both user and background code; this is accomplished by specifying both options in a single **tck** command.

```
R>tck -ub
R>tck
  tck -r L -ub -s S
```

Selecting Clock Signals

Three **tck** options may be used to select analyzer clock sources:

- **-r** Specifies that the clock should take place on the rising edge of the signal(s) which follow.
- **-f** Specifies that the clock should take place on the falling edge of the signal(s) which follow.
- **-x** Specifies that the clock should take place on both edges of the signal(s) which follow.

Five clock signals may be selected: J, K, L, M, and N. Clocks J and K are the external clock inputs available when your emulator contains an external analyzer. The external clock inputs should not be used to clock the emulation analyzer; however, it may occasionally be useful to use the external clock signals to qualify the emulation trace (see the "Qualifying Clocks" discussion below).

The L, M, and N clock signals are generated by the emulator. Typically, the L clock is the emulation clock derived by the emulator, the N clock is used as a qualifier to provide the user/background tracing options (**-u** and **-b**) to **tck**, and the M clock is not used.

When several clocks are specified, they are ORed; that is, each signal specified will clock the analyzer.

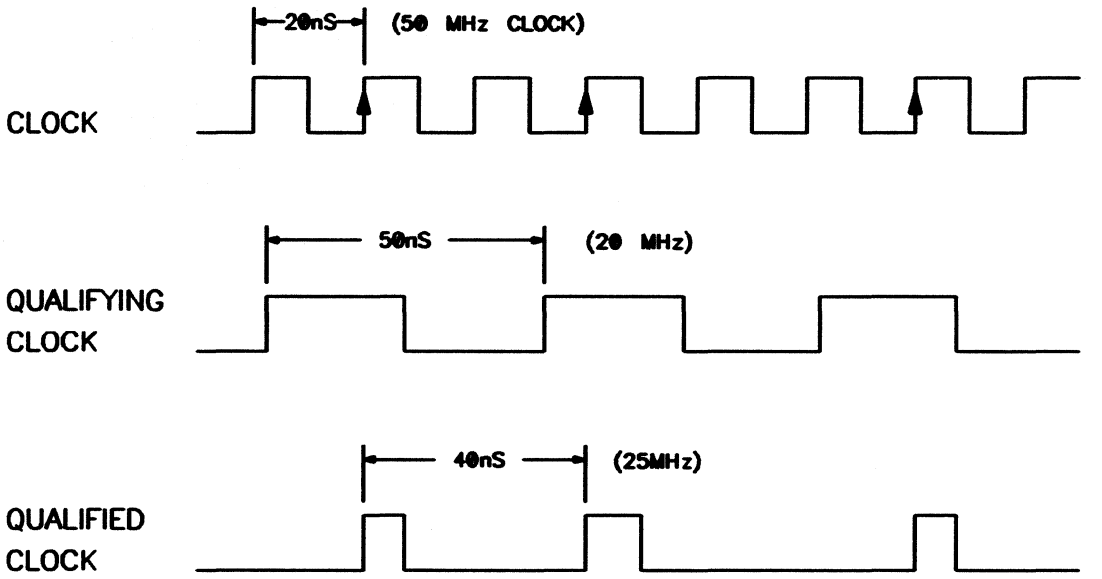


Figure 6-1. Qualified Clocks

Specifying the Maximum Qualified Clock Speed

The maximum qualified clock rate is the repetition rate of all specified clock signals (see figure 6-1). You are allowed to select the maximum qualified clock speed of the analyzer; however, there are tradeoffs involving the trace count qualifier to be considered. You select the maximum qualified clock speed with the **-s** option to the **tck** command. There are three maximum speeds that can be specified:

- Slow (tck -s S). Slow specifies a maximum qualified clock rate of 16 MHz. When S is selected, there are no restrictions on the trace count qualifier.
- Fast (tck -s F). Fast specifies a maximum qualified clock rate of 20 MHz. When "F" is selected, the trace count qualifier may be used to count states but not time.

- Very Fast (tck -s VF). Very fast specifies a maximum qualified clock rate of 25 MHz. When "VF" is selected, the trace count qualifier may not be used at all (in other words, **tcq none**).

Qualifying Clocks (tck -l, -h)

The selected clock signals may be qualified with other clock signals; that is, the selected signals may only clock the analyzer when the qualifying clock signal is true. Clock signals are qualified by using the **-l** and **-h** options to the **tck** command. The **-l** option is used to specify a qualifying signal which only allows the trace to clock when this signal is lower than the threshold voltage. The **-h** option is used to specify a qualifying signal which only allows the trace to clock when this signal is higher than the threshold voltage. Any signal, J, K, L, M, or N, may be used to qualify other signals.

Note

If several clock qualifiers are specified, the analyzer will be clocked if any one is true. This applies to the user/background qualifier as well. If you wish to use one of the external clocks as the only qualifier, you must turn off the user/background qualifier; in other words, **tck -ub**.

Qualifier Setup and Hold Times of the External Analyzer

Qualifier setup time is approximately 25 nanoseconds when the external analyzer is aligned with emulation analyzer (**xtmo -e**). Qualifier setup time is approximately 20 nanoseconds when the external analyzer operates as an independent state analyzer (**xtmo -s**). Qualifier hold time is approximately 5 nanoseconds.

Using Slave Clocks for Demultiplexing (tsck)

There are two modes of demultiplexing that can be set for each 16-bit pod: mixed clocks and true demultiplexing.

Emulation trace slave clocks are specified with the **tsck** (trace slave clock) command. External analyzer slave clocks are specified with the **xtsck** (external trace slave clock) command. (Master clocks are specified by the **tck** and **xtck** commands.) By default, the slave clocks are turned OFF, as may be specified by the **-o** option to the **tsck** command.

Rising edges (**-r**), falling edges (**-f**), or both edges (**-x**) of clocks J, K, L, M, or N may be specified as the slave clock.

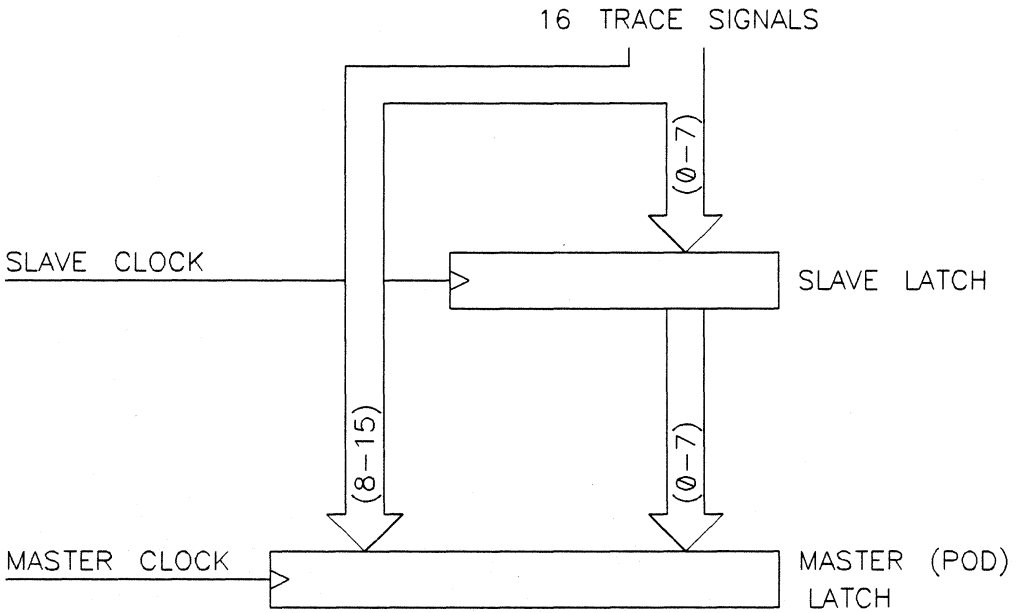


Figure 6-2. Mixed Clock Demultiplexing

Mixed Clocks

The mixed clock mode is specified with the **-m** option to the **tsck** command. In this mode, the lower 8 channels of the pod (bits 0-7) are latched with the slave clock, and the master clock gates the entire pod (see figure 6-2).

If no slave clock has appeared since the last master clock, the data on the lower 8 bits of the pod will be latched at the same time as the upper 8 bits. If more than one slave clock has appeared since the last master clock, only the first slave data will be available to the analyzer (see figure 6-3).

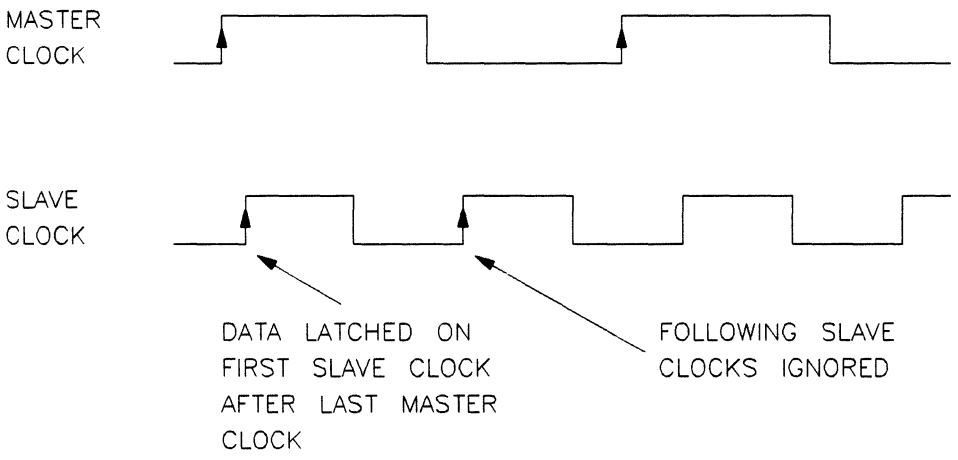
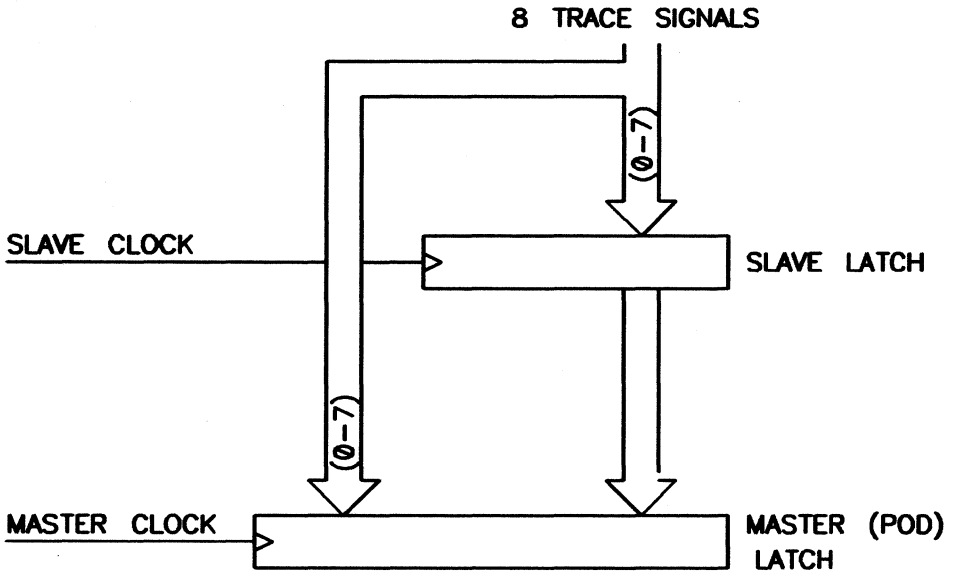


Figure 6-3. Slave Clocks



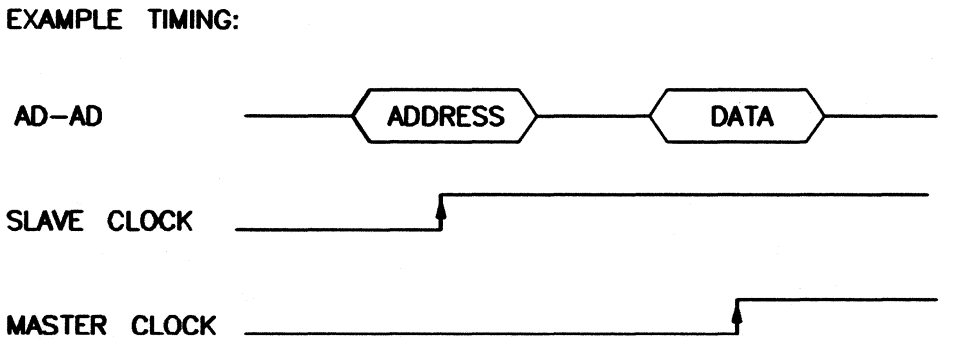


Figure 6-4. True Demultiplexing

True Demultiplexing

The true demultiplexing mode is specified with the **-d** option to the **tsck** command. In this mode, the lower 8 channels of the pod (bits 0-7) are latched with the slave clock; the upper 8 channels also get data from signals 0-7, but they are clocked with the master clock. Thus, the analyzer gets two copies of bits 0-7. The slave clock latches the data for bits 0-7, and the master clock then gates the entire pod into the analyzer (see figure 6-4).

If no slave clock has appeared since the last master clock, the data on the lower 8 bits of the pod will be the same as the upper 8 bits. If more than one slave clock has appeared since the last master clock, only the first slave data will be available to the analyzer.

Saving Trace Specifications in Command Files

If you are using your emulator in the transparent configuration (in other words, the emulator is connected between a terminal and a host computer), you can save trace specifications to command files on the host computer.

Example

The following example makes several assumptions:

- A host computer running HP-UX (which you are currently logged in to).
- The terminal is connected to port B, and the host computer is connected to port A.
- The analyzer is in the "complex" configuration, and you have a trace configuration which you wish to save.

Because you may wish to save trace specifications at any time, it is a good idea to create a macro containing the commands used to save the trace specification.

```
U>mac tsave={po -o a; echo "cat > tspec";w 1; tcf; tpat; trng; tsq; tpq; tcq; tp;
echo "#+#"; echo \04; po -o b}
```

The commands which make up the **tsave** macro do the following things:

po -o a; This command specifies that standard output be sent to port A, in this case, the host computer.

echo "cat > tspec"; This command will open a file on the host computer. This file will receive the output of the trace display commands which follow.

w 1; This command causes the emulation system to wait for one second to ensure that the "cat > tspec" command has time to set up on the host.

tcf; tpat; trng; tsq; tpq; tcq; tp; These commands send the current trace specification to the standard output.

echo "# + #"; This command sends the "# + #" command file terminator string to the standard output. The terminator string is used when you use the command file to respecify the trace.

echo \04; This command sends a <CTRL>d end of file character to the host computer to close the "tspec" file.

po -o b; This command specifies that standard output be sent to port B, in this case, the terminal.

After the macro has been defined, you can save the current trace specification by entering the name of the macro as you would any other command.

```
U>tsave
  # po -o a;echo "cat > tspec";w 1;tcf;tpat;trng;tsq;tpq;tcq;tp;echo "#+#";echo
\04;po -o b
U>
```

After **tsave** has executed, there exists a file called "tspec" on the host computer which contains the trace specification. To use the command file to load the trace specification enter the **po** (port control) command with the **-s** option.

```
U>po -s "cat tspec"

  # waiting for 1 second....
  tcf -c
  tpat p1 addr=489
  tpat p2 addr=5ff and data=39xx and stat=mw
  tpat p3 addr=45a
  tpat p4 addr=5ff and stat=mw
  tpat p5 data!=39xx
  tpat p6 addr=5fe and data=0xxf7 and stat=mw
  tpat p7 any
  tpat p8 any
  trng addr=5f8..5ff
  tif 1 p1 2
  tif 2 p2 3
  tif 3 p3 4
  tif 4 never
  tif 5 any 6
  tif 6 any 7
  tif 7 any 8
  tif 8 never
  tsq -t 4
  tsto 1 none
  tsto 2 all
  tsto 3 all
  tsto 4 all
  tsto 5 none
  tsto 6 none
  tsto 7 none
  tsto 8 none
  telif 1 never
  telif 2 p3 1
  telif 3 p4 and p5 2
  telif 4 never
  telif 5 never
  telif 6 never
  telif 7 never
  telif 8 never
  tpq none
  tcq time
  tp -a 10
#+#
U>
```

Notes

Index

Notes

Hewlett-Packard
Printed in the USA