Concepts Of Emulation And Analysis

HP Part No. 64000-97000
Printed in U.S.A.
November 1990

Edition 1

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.

UNIX is a registered trademark of AT&T in the U.S.A. and in other countries.

Torx is a registered trademark of Camcar Division of Textron, Inc.

Hewlett-Packard Company
Logic Systems Division
8245 North Union Boulevard
Colorado Springs, CO 80920, U.S.A.

Printing History

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

Edition 1 **64000-97000, November 1990**

Using This Manual

This manual discusses concepts associated with emulation and analysis. Separate chapters are devoted to discussions of emulation monitors and the process of mapping memory.

This manual will be useful to a designer who has been using an emulator and/or analyzer and would like a greater understanding of these tools. This manual will also be useful to the person who is new to emulation and analysis and would like to gain a quick perspective on these tools.

Notes

Contents

Illustrations

1

Introduction To Emulation

This chapter answers the following questions:

- What is an emulator?
- What is the purpose of an emulator?
- Can an emulator help me before my target system is ready to run?
- Physical description of an emulator.
- What are the steps in using an emulator?
- What tasks are done by an emulator?
- What is happening while my program is running in emulation?
- How does the emulator affect my program?
- How does the emulator affect my target system?
- Can an emulator run interactively with other emulation/analysis systems, and with external analyzers?

What Is An Emulator?

An emulator is part of an environment in which hardware and software are developed and integrated to create products that depend on embedded microprocessors. To understand an emulator, you need to understand the microprocessor development

environment (also called development system). Then you will understand how an emulator fits into a development system. A development system will have a work station (a terminal in a network, or a personal computer).

A development system has an editor (figure 1). An editor is a software module that allows you to create and modify files by typing them on the keyboard of your workstation. You use the editor to create the text files (also called source files) of program code you intend to run in your target system (the product you are developing).

Figure 1-1. Developing Software On An Editor

A development system has a compiler and/or assembler and a linker (figure 2). These are software modules that read source files and create corresponding files of machine code that can be executed by target-system hardware. You use these modules to generate the object files that perform the tasks implemented in the text files you created in your editor.

These object files contain your actual binary code which directly controls your target microprocessor. However, it isn't until the linker module creates an absolute file from the object file, that the file can be loaded into the emulation and/or target-system memory.

Figure 1-2. Creating Absolute Files

Finally, you have an emulator (figure 3). An emulator is the interface between your workstation and the product you are developing (called the target system). An emulator provides the mechanism that loads the executable (or absolute) file you created into the target system. An emulator is the linkage that allows you to run and stop program execution in your target system. An emulator lets you view (on your workstation display) the changing conditions within the target-system microprocessor as each of the instructions in your program is executed.

An emulator is connected to a target system by removing the microprocessor from the target system and plugging in the emulator probe cable in its place. Differences between the

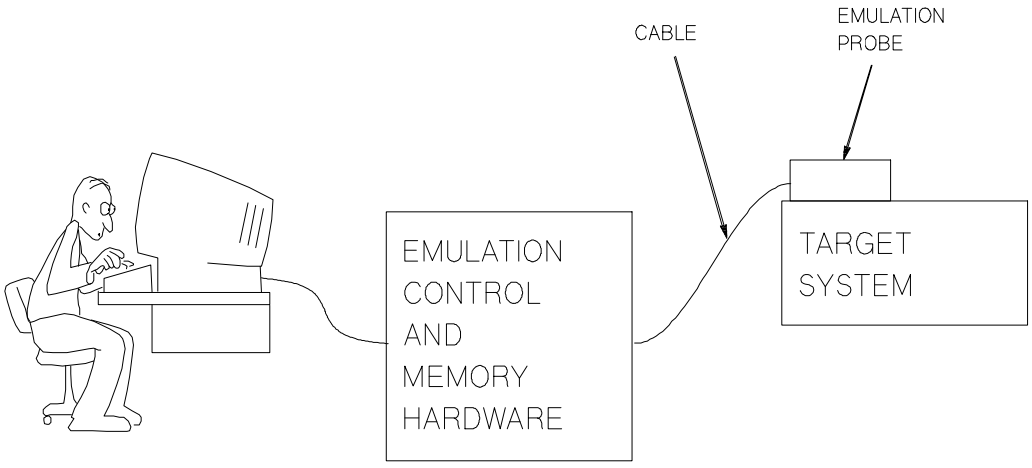

Figure 1-3. Running In Emulation

emulator and the target microprocessor are usually transparent to the target system.

You control what the emulator does by entering commands on your keyboard. The emulator has all the capabilities of the target system processor, and a lot more. An emulator can load the absolute files into hardware memory in your target system. If your target system does not have the hardware memory needed to contain your program, your emulator can provide memory hardware; your target processor can address code in emulation memory and execute it as easily as if it were in the target system.

Your emulator gives you control over the running of your absolute files in the target system microprocessor. You can start execution at any instruction address you desire in your program.

Your emulator can show you how your target system processes your code. Your emulator can show you how your code affects registers and other components in your target processor. If your emulator has an internal analyzer associated with it, the analyzer can record the series of states that were executed so you can see the details of how your target system processed the program you wrote. In this way, your emulator helps you understand the results of running your target program in your target system.

What Is The Purpose Of An Emulator?

The purpose of an emulator is to help you develop and integrate target system hardware and software. You can use an emulator to ensure that your target system hardware and software work together. The emulator can be used by itself, or it can be used with other test equipment (other emulators, analyzers, debuggers) to debug and integrate your target system hardware and software. In order to meet its purpose, the emulator must have excellent transparency, that is, it must look like the target microprocessor from the point of view of your target system hardware. Refer to the glossary in this manual for details of characteristics that are considered when designing for transparency.

Can An Emulator Help Me Before My Target System Is Ready To Run?

Yes it can. You can run your emulator even though it's not connected to a target system. An emulator can make a lot of hardware available to substitute for your target system hardware. An emulator has memory (called emulation memory) that you can load your absolute files into before your target system memory is available. You can run your program from emulation memory just as if the program was loaded into target system memory.

Many emulators can supply other hardware to substitute for hardware that is not yet available in your target system. The keyboard of your workstation may be used as a substitute for a keyboard you intend to include in your target system. The display of your workstation may substitute for a display you intend to have in your target system. Other equipment (printers, etc) can often be simulated by your emulator so that you can test code you are developing to communicate with the other equipment even before the target system printers, etc., are available.

Physical Description Of An Emulator

A typical emulator will include:

- An emulation probe. The probe replaces the processor in your target system. The probe plugs into your target system microprocessor socket. Remove the microprocessor from your target system and plug in the emulation probe in its place. To your target system, the emulation probe appears to be the processor you just removed. For details, refer to the discussion of transparency in the glossary of this manual.
- An emulation probe cable. This cable carries control signals and processor information back and forth between the emulation probe and the emulator control and memory hardware.

- The emulator control and memory hardware. This circuitry controls operation of the processor, and provides emulation memory in which you can store part or all of your absolute file.

- An emulation bus analyzer (also called internal analyzer). This analyzer is usually installed in the same frame with the emulator control and memory hardware. The analyzer stores a record of the details of state executions in the emulator and target system. Your workstation can display a trace list of these state executions (or inverse-assemble them into opcodes and operands to show you the instructions that were executed) so you can see how the target system processed each instruction in your absolute file.

What Are The Steps In Using An Emulator?

There are three steps in the emulation process.

1. Prepare The Software

   Create your program. Assemble or compile the program and link the assembled or compiled modules. This obtains an absolute file that performs the tasks in the program you created, but in a form that can be executed by your target microprocessor. These are the steps shown in figures 1-1 and 1-2.

2. Prepare The Emulator

   Install your emulator hardware (emulator control board, emulation memory, analyzer, etc). Then connect your emulator probe to the target system. This connection is optional; it depends on whether or not you need to perform tests in target-system hardware. (Refer to the discussion of in-circuit/out-of-circuit emulation in a separate chapter in this manual). Finally, on your workstation, answer a series of questions that define the details of the configuration of your emulator; invoke this

series of questions using a keyboard command (such as "modify configuration").

3. Run Your Target Software Using The Emulator

   Use the emulator to load your absolute file into memory hardware. Then use the features of the emulator to execute your absolute code and see how your program runs in your target processor. While using the emulator, you can display the contents of the registers and/or memory to help you debug your hardware and software. This is shown in figure 1-3.

What Tasks Are Done By An Emulator?

The following paragraphs describe tasks that are normally performed by an emulator under control of the emulator user.

- Program Loading. An emulator can load emulation memory or target system memory with the absolute file you developed using an editor, compiler, assembler, and linker.
- Run/Stop Controls. An emulator can run your programs, starting from any address in memory. A program running in your target system can be stopped by the emulator at any point you select, and then resumed again at the point where it was stopped. Execution can also be stopped by resetting the microprocessor.
- Reset Support. The emulator can be reset from the emulation system under your control; or your target system can reset the emulation processor.

- Memory Display/Modification. An emulator can display the code contained at addresses in target system memory or emulation memory. You can use an emulator to modify the code present at these memory addresses, if desired.

- Global and Local Symbols Display. An emulator can show you the names of all the global and local symbols you defined in your program, along with the memory addresses where code associated with those symbols is located.

- Internal Resource Display/Modification. An emulator can display the present values in the internal resources of the emulation processor, such as registers, and modify those values, if desired. This includes modifying the program counter so you can control where the emulator starts a program run. An emulator can also display or modify the contents of a memory-management unit (MMU register), if your target processor includes memory management.

- Analysis (in some systems, analyzer boards are optional). An analyzer can record the states that appeared on the emulation processor bus. The analyzer can show you a list of those states as a trace list of instructions and data flow.

- Program Stepping. An emulator can execute your program one instruction at a time (or a selected number of instructions at a time). After each execution, the emulator will stop so you can see how the internal machine states changed.

- Memory Mapping. An emulator can place your absolute code in emulation memory or in target system memory, or it can place portions of your code in each memory.

- Memory Characterization. An emulator can cause emulation memory to behave as ROM or RAM (even though all emulation memory is RAM). This allows you to test code that will eventually reside in ROM without having to use ROM hardware.

- Breakpoint Generation. Breakpoints are temporary instruction codes that you can have an emulator place in your program code to halt execution or perform desired tasks after a particular event occurs. You can use your emulator to set breakpoints that are recognized on the occurrence of a selected machine state or on the occurrence of any state within a range of states. Breakpoints cause the emulator to halt program execution when the program reads from or writes to a memory location, or after the program executes an instruction at a specified memory location.

- Clock Source Selection. An emulator generates an internal clock that you can use for program execution before your target-system clock is available. You may need to include wait states in your program code if the clock is too fast for some of the hardware in your target system, such as memory or I/O ports.

- Modify I/O Ports. An emulator can read from and write to external devices.

What Is Happening While My Program Is Running In Emulation?

The emulator performs different tasks, depending on whether it is executing your target program, or it is executing its monitor program. (A monitor program is the set of software routines that provide the features of the emulator, such as read/write target memory, display/modify registers, and control execution of the target program. Monitor concepts are discussed in a separate chapter in this manual.)

While Execution Is In The Target Program

During execution of your target program, the emulation processor (acting as the target processor) generates an address for each bus cycle. If the generated address is mapped to target system memory, the emulator enables the data path buffers between the emulation processor and your target system. If the generated address is mapped to emulation memory, the emulator enables the data path

buffers between the emulation processor and the emulation bus. This way, the emulator makes sure that each address generated by the emulation processor arrives at the appropriate memory hardware.

As your program runs, the analyzer records the activity on the emulation bus. Activity of the emulation processor is always available to the emulation bus analyzer, regardless of whether an address is located in the target system or in emulation memory. If you set up the analyzer to store state flow, your terminal can display a list of the states in the order that they appeared on the bus.

While Execution Is In The Monitor Program

Many emulator functions are done by breaking (transferring execution to the routines in the emulation monitor). One monitor routine stores the values of the emulation processor registers. These values can be reloaded later if you want to resume execution of your target program at the point where the break occurred.

In the monitor, the emulation processor executes a tight loop while it waits for a command from you. You can display the register values that were saved at the time when the "break" occurred, display the present content of code in emulation memory or target-system memory, load new program code, single-step through the target program instructions, or perform a variety of other tasks, depending on the capabilities of the routines that make up the monitor. When you have finished making the tests and measurements that are provided by monitor routines, you can exit the monitor and resume execution of your target program, if desired.

How Does An Emulator Affect My Program?

An emulator does not alter your program, but the operating mode of the emulator may affect the way your program runs. There are two operating modes of an emulator: real-time mode, and non-real-time mode.

Real-Time Mode

In real-time mode, the emulator runs your program without interfering with its execution. Real-time mode is the mode to use if you have circuitry that can be damaged by an inadvertent break during execution of your target code. The only way to break execution when in real-time mode is for you to enter a BREAK command on the command line and press the RETURN key.

Whenever the emulator is executing any routine in the monitor program, it is not executing in real-time mode. The following is a list of features that cannot be performed when the emulator is in real-time mode because these require execution of one or more routines in the monitor program. Depending on which emulator you are using, there may be additional features that cannot be performed when the emulator is in real-time mode:

- Target memory accesses -- display, copy, load, modify, and store.
- Register accesses--display, copy, and modify.
- Software breakpoints--set and reset.

The features above can be accessed even if the emulator is configured for real-time mode if you first issue a BREAK command. In the case of a software breakpoint, you can set a breakpoint and then run your program in real-time mode. When the software breakpoint instruction is executed, the monitor will be entered. Of course, your mode will no longer be real-time when this happens.

Non-Real-Time Mode

In non-real-time mode, the emulator can break program execution and begin executing a monitor routine in response to a variety of internal program conditions (such as an attempted write to ROM), as well as a BREAK command from you. The non-real-time mode

offers use of the full feature set of the emulator, but the timing of a break during execution of your target code may cause erratic behavior in your target system.

How Does An Emulator Affect My Target System?

When an emulator is designed, every effort is made to make the emulator transparent to the target system. Even so, an emulator is not the processor it replaces. It will usually have some parameters that are different from those of the target processor. The following is a series of items you should consider when using an emulator in place of your target-system microprocessor.

Consider mechanical accessibility of the target processor. It may be necessary to place the circuit board containing the microprocessor on an extender board to gain access to the processor connection. Some circuits may not work properly if the circuit board is extended from its motherboard connection.

Consider interference with nearby components. Other components close to the processor may interfere with installation of the emulator probe cable. With older processors, you could install extender sockets between the processor socket and the emulator to overcome this problem, but with todays clock speeds, extender sockets may cause unreliable operation do to their effects on impedence or crosstalk.

Consider the processor timing specifications. It's important that your target system avoids violating any of the timing specifications of the microprocessor that will be emulated. Even if the microprocessor will work in circuits that violate some of its timing specifications, the emulator may become erratic or not be able to work at all because of these timing violations.

Consider delays. Some processor cycles, such as memory accesses may not have the full memory access timing cycle available when the emulator is connected. This is because of signal delays caused by the emulation probe cable and the buffers for the cable drivers/receivers. In these cases, you might replace your standard memory hardware with faster hardware in your prototype design.

Consider power and ground circuits. If your target-system prototype is a wire-wrapped design with little or no power and ground facilities, it may not work well with an emulator. This is because emulators use high-current drivers to maintain the integrity of the signals in the emulation probe cable. The high currents can cause voltage pulses to develop in unpredictable points in such a wire-wrapped circuit. These voltage pulses may exceed the thresholds of some of your target-system components.

Consider circuitry that needs to be refreshed. If your target system includes a watch-dog timer that must be updated periodically, you may need to disable it in order to prevent it from shutting down your target system when the emulator is executing routines in a background monitor.

Can An Emulator Run Interactively With Other Emulation/Analysis Systems, And With External Analyzers?

By using intermodule connections to connect control board assemblies together, emulation and analysis hardware can interact with other emulators and/or analyzers. Interaction allows you to develop systems that use two or more microprocessors together or that need more analyzer channels then are available in a single analyzer. Things you can do by using interaction include:

- Starting two or more measurements at the same time.
- Using conditions found by one analyzer to control measurements taken by another analyzer.
- Starting execution of a program when a measurement starts.

Notes

2

Comparing In-Circuit And Out-Of-Circuit Emulation

Emulators can be used for both out-of-circuit emulation and in-circuit emulation. Simply stated, in-circuit emulation is when you have your emulator connected into your target system, and you are using the emulator to run your target code in your target hardware. Out-of-circuit emulation is when you have your emulation probe disconnected from any target hardware. Using out-of-circuit emulation, you run your target code on the emulation processor, storing it in emulation memory, and using the facilities of your emulation system to substitute for the hardware of your target system.

Out-Of-Circuit Emulation

Figure 2-1 shows an emulator used to perform out-of-circuit emulation. Normally, out-of-circuit emulation is used to develop software before the target-system hardware is available. The out-of-circuit emulator allows you to run your code on your target processor; the emulator contains the target processor (also called the emulation processor). When you have written a program, you can store it in emulation memory, and the emulation processor can fetch its instructions and write its data to locations in emulation memory.

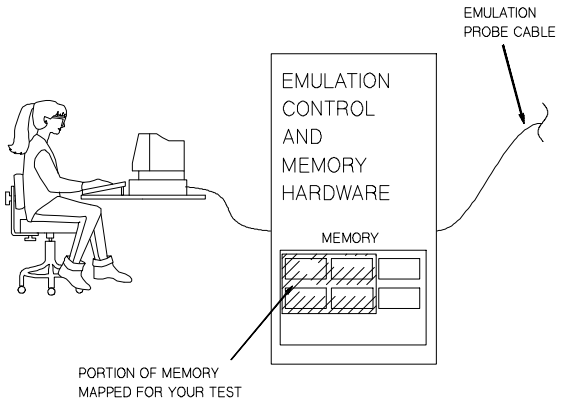

Figure 2-1. Out-Of-Circuit Emulation

Things You Can Do Using Out-Of-Circuit Emulation

The most obvious thing you can do is start developing and testing your code before the target system hardware exists. The following paragraphs list some emulator features that allow you to do this code development.

The emulator has memory. You can use it to store your code before there is any target memory available.

Many emulators have simulated I/O. It can simulate the behavior of keyboards, displays, printers, etc. With simulated I/O, you can test the code that the processor will execute when it services these I/O devices, even before you have the I/O devices available.

In-Circuit Emulation

Figure 2-2 shows an emulator being used to perform in-circuit emulation. Using in-circuit emulation, you can develop and test your target system hardware. You can also test your code in the actual environment where you intend for your code to run.

Your target system operates as though its normal microprocessor is installed, but the emulator gives you control over that processor so you can control program execution.

With in-circuit emulation, you can load the program you have created into address space that exists on your target system. If your target system is not yet complete, you can load part of your program code in target system memory, and the rest of your code in emulation memory (where emulation memory simply provides space that will ultimately reside in your target system).

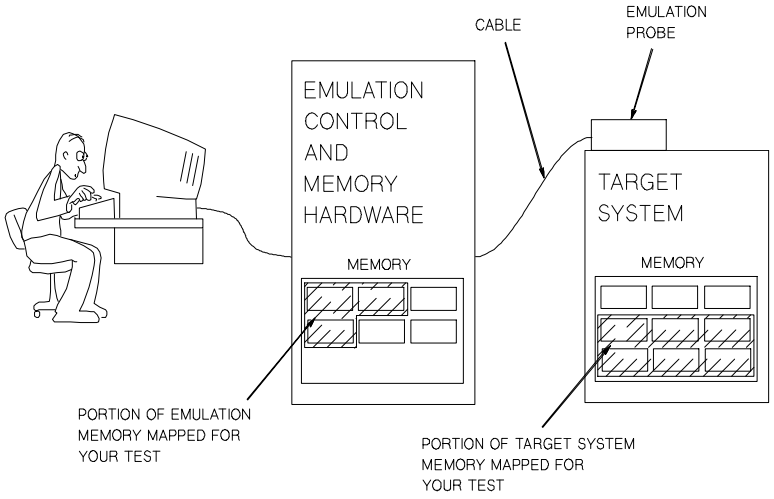

Figure 2-2. In-Circuit Emulation

Things You Can Do In-Circuit That You Can't Do Out-Of-Circuit

When you have target system hardware available and connect your emulation probe into it, you can load your target program into your target-system memory and test the behavior of your target hardware when it tries to execute the instructions.

You can use the "display/modify memory" features of the emulator to send patterns to addresses in your target hardware and then read the contents of those addresses. In this way, the emulator can help you determine that your target system components are wired correctly.

You can also eliminate the differences that may exist between the behavior of your target devices and the emulation system that simulated those devices.

- There may be differences in the response of emulation memory compared with the response of your target memory hardware.
- There may be differences between the way your I/O devices operate and the way the emulator simulated those I/O devices.
- There may be differences in the way your interrupting devices operate and the way you might have simulated an interrupt while operating out-of-circuit.

As soon as your target hardware is available, you will probably want to develop your product using in-circuit emulation.

3

The Emulation Monitor

What Is A Monitor?

A monitor is a group of software routines that provide the features of the emulator. The following is a list of features that are normally provided by routines in the monitor:

- Read/write target memory.
- Display/modify registers or flags in the emulation processor.
- Display/modify contents of memory.
- Control execution of the target program.
- Execute a target program to a predetermined point, and then halt. (This is useful when you want to begin a test at a point deep within the target program.)
- Break from target program (see The Break Function Of The Emulation Monitor later in this chapter).
- Reset into monitor.
- Single step (execute one program instruction each time a "step" key is pressed).

What Does A Monitor Do?

Many emulator functions are achieved by transferring control from your target program to one of the routines in the emulation monitor. Whenever the monitor program is entered, it saves the values of the emulation processor registers. You can immediately display and examine the register information (the values of the emulation processor registers immediately before entry into the monitor.

The monitor routines perform the emulator functions you request, such as displaying target system memory. When you display target system memory, the monitor program executes instructions in the emulation processor to read target memory locations and return their contents to the emulator.

Caution 

DAMAGE TO TARGET SYSTEM HARDWARE. When the emulator detects an illegal condition, such as when the emulation processor tries to write to guarded memory, or when you request a memory access that uses a monitor routine (such as "display memory"), the emulator stops executing your target code and begins executing code in the monitor. Be careful if you have circuitry that may be damaged if the emulator stops executing your target code at a critical point. You may need to restrict the emulator to real-time mode to prevent it from transferring control (breaking) to the emulation monitor. In real-time mode, you can only break to the monitor by entering the "break" command and pressing the Return key.

The Break Function Of The Emulation Monitor

The most common method used to transfer control (break) to the monitor program is by assertion of the microprocessor's non-maskable interrupt pin. However, other techniques may be used, depending on the architecture of the target microprocessor. When a break condition is detected, the emulator asserts the non-maskable interrupt of the emulation processor to stop execution of the target program and begin execution in the emulation monitor. The following conditions can cause a break to be generated:

- An illegal memory reference (such as an attempt to read guarded memory space).
- A bus condition that the internal analyzer detects (such as a bus error).
- A request by the target software (such as a software breakpoint).
- Typing "break" from the keyboard, and pressing RETURN.

Where Is The Monitor?

The monitor may be located in background or foreground memory space, or both background and foreground. Background memory is memory that is separate (and isolated) from the memory addressed when running the target program; you may have an instruction beginning at address 1000H in foreground memory and an unrelated instruction at address 1000H in background memory. Foreground memory is the memory space directly addressable by the target processor; it is the address space that is shared by the target program, I/O ports, etc., of the target system. Most emulators place the monitor in either background or foreground memory space. Some emulators place part of the monitor in background and the rest in foreground. Background and foreground monitors are discussed in the following paragraphs.

Background Monitors

If the monitor is in background, it operates in a memory space that is not directly accessible to your target system. This memory space is called "background" memory, from which the name "background monitor" is derived. The emulation system controls processor access to the background memory.

Advantages Of Background Monitors

Background monitors have two main advantages:

1. A background monitor is easy to use. It is available at power up. You don't have to spend any time learning how the monitor routines work. Emulators equipped with background monitors are easier to use than those with foreground monitors when testing and debugging hardware at the early stages of a design project, before target-system code is written, or before all of the target system memory space is defined or operational.

2. Background monitor routines don't consume any of the address space of your target system.

Disadvantages Of Background Monitors

Background monitors have two main disadvantages:

1. When a processor is executing code within background memory, its ability to detect and handle target system interrupts is severely limited. Therefore, interrupts may be ignored and refresh cycles needed by DRAM's, etc., may not be generated. If a watchdog timer is operating in your system to check on system updating processes, the watchdog timer will not be updated as necessary, and therefore, it may shutdown or reset your target system. Real-time servicing of keyboards, displays, etc., will not occur as long as the emulation processor is executing in background space.

2. The routines within background monitors cannot be accessed or modified. If you'd like to add a feature to your emulator for better support of your target system, or if you'd like to change the way a monitor routine works with your target system, there's no way to do it.

Foreground Monitors

Foreground monitors are supplied as source files. They are written in the assembly language of the processor they emulate. You can edit the monitor source file, if desired. Then you must assemble the monitor program, link it, and load it in emulation memory. It is important to load the foreground monitor into emulation memory because this permits the emulation processor and the rest of the emulation hardware to communicate properly.

Advantages Of Foreground Monitors

Foreground monitors have at least three primary advantages over background monitors:

1. A foreground monitor can continue to service the needs of a target system (i.e., interrupts, memory refresh, etc.) while the monitor routines are executing. By making appropriate modifications to your foreground monitor, your emulator will be able to service real-time events, such as interrupts and watchdog timers in the target system while the emulator is executing your monitor code. For most multitasking, interrupt-intensive applications, you will need a foreground monitor.

2. You can modify the monitor routines to meet the needs of your target system. You can customize the way your emulator handles instructions. You can assign different priorities to your target system interrupts. You can modify the monitor program in cases where operation of a monitor routine conflicts with operation in your target system.

3. You can write routines to perform special tasks for your target system and then call your routines by using the structure of the monitor. For example, you might call a routine you have written that allows your target system to modify the values in coprocessor registers (if your target system uses a coprocessor).

Disadvantages Of Foreground Monitors

The disadvantage of a foreground monitor is that you need to assemble it, link it, and load it in emulation memory before you can use it. If you want to customize the monitor for your target application, you will have to understand how the monitor works. Also, you'll have to sacrifice a portion of your target processor's addressable memory space to contain the foreground monitor code. This may not be much of a problem when developing code for a 32-bit processor, but it can cause significant problems when developing target systems that use 8-bit processors with 64K of address space.

Some Emulators Have Both A Background And Foreground Monitor

When an emulator has both a background and foreground monitor, the support provided by each monitor may be different. For example, full support of emulation may be supplied by the background monitor, and interrupt handling and custom coprocessor support may be provided by the foreground monitor. If your emulator has both a background and foreground monitor, the following paragraphs will help you decide when to use each one.

When To Use The Background Monitor

Use the background monitor during the early stages of hardware development, before the target system interrupt, bus error, and other asynchronous capabilities are available. The background monitor is the ideal environment to make the first operational tests. Because designers of a background monitor make no assumptions about the status of the target system, you can execute simple tests even without any components, except the emulator probe plugged into the target system. Using an emulator with a

background monitor, you can write a standard checkerboard bit pattern to various data, address, and control lines to verify that they're wired correctly. When you start adding RAM or I/O chips to your target hardware, you can use the display/modify target memory features to send test bits and read the results.

An emulator with a background monitor can also run some simple test programs. For example, you can create an infinite loop to write alternating "ones" and "zeros" to a memory address. Then, with an oscilloscope, you can probe various points in your target hardware to check for ringing on power and ground lines, violations of setup and hold times on read/write devices, and crosstalk between lines.

When To Use The Foreground Monitor

Use the foreground monitor when the background monitor no longer supports development of your target system. For example, if your target system uses dynamic RAM, the data in the dynamic RAM will be lost when the background monitor is entered because no refresh cycles will be generated. If your target system includes I/O buffers that communicate with the host computer, communication will cease because the interrupt/handshake cycles needed by the host-based I/O drivers cannot be generated when execution is in background memory.

If the design of the target system hardware is nearly complete and the target memory system is stable and reliable, it's time to use the foreground monitor. An emulator is more transparent when it is used with a foreground monitor. Interrupts, bus errors, and other exceptions can be handled by the target system software as if the emulator were not present. All emulation and analysis functions are available. You can customize the monitor program to fit your application. Target-specific messages can be added and displayed on the emulation terminal. Display and modification of coprocessor registers (if present) can be done by adding the necessary code to the foreground monitor program.

One example of development that would need a foreground monitor is development of target system interrupt service routines (ISR's). Because interrupts do not affect the operation of monitor routines, you can analyze the execution of your ISR's with an internal analyzer while using your emulator (with foreground monitor) to watch data move between I/O and memory addresses.

You can also single step through one ISR routine while other ISR's continuously generate interrupts. An interrupt of higher priority can always take control from an interrupt of lower priority. By changing the relative priority levels of the ISR's, you can single-step through an ISR that you have given a low priority and watch the higher priority ISR's periodically take control.

For The Emulator With No Foreground Monitor

The rest of the chapter discusses information needed by users of emulators with foreground monitors. If your emulator does not have a foreground monitor, you don't need to read any more of this chapter.

Customizing The Emulation Foreground Monitor

The source file for a foreground monitor is thoroughly commented so you can understand its routines and make desired modifications. A flowchart of a typical emulation monitor is given in this chapter.

Caution

POSSIBLE LOSS OF WORK SESSION!
SYSTEM MAY BECOME UNUSABLE. Do not let the routines you customize in the emulation monitor exit the monitor. The monitor has a defined exit routine that reloads appropriate values in registers, etc. It must be used. Exiting the monitor by any other path may cause the entire system to become unusable.
Do not modify routines in the monitor unless changes are recommended and instructions are outlined in your monitor source file. Incorrect changes in some sections of a foreground monitor may cause emulation features to stop working.

Caution

LOSS OF ORIGINAL MONITOR SOURCE PROGRAM!
Do not modify the original source file for the foreground monitor. Copy the monitor source file to your own subdirectory and make modifications only to your copy.

Some emulation monitor source files are shipped with "read-only" permissions. If your monitor was shipped this way, you will need to execute "chmod 666 < monitor file name> " on your copy before you modify it. After modifying your foreground monitor, reassemble it and relink it.

How Monitors Are Structured

An emulation monitor is made up of the following major sections:

1. Processor exception vector table.
2. Entry points into the monitor.
3. Emulation command scanner.
4. Command execution modules.

Each of these sections is discussed in the following paragraphs.

Figure 3-2 is an example flow chart of a typical emulation monitor. This flow chart does not show the processor exception vector table. The processor exception vector table for this example flow chart has uncommented six entry vectors.

Processor Exception Vector Table

The emulation monitor (figure 3-1) is entered through processor exceptions. The emulation monitor program contains instructions that load the processor exception vector table with the addresses of the routines that handle the various exceptions.

```
  ORG 0  ---RESET---
     DC.L SP_TEMP
     DC.L RESET_ENTRY
* ORG 8         ---BUX ERROR---
*    DC.L BE_ENTRY
* ORG $0C        ---ADDRESS ERROR---
*    DC.L AE_ENTRY
* ORG $10        ---ILLEGAL INSTRUCTION---
*    DC.L II_ENTRY
* ORG $14        ---ZERO DIVIDE---
*    DC.L ZD_ENTRY
* ORG $18        ---CHK INSTRUCTION---
*    DC.L CI_ENTRY
* ORG $1C        ---TRAPV INSTRUCTION---
*    DC.L TI_ENTRY
* ORG $20        ---PRIVILEGE VIOLATION---
*    DC.L PV_ENTRY
  ORG $24        MONITOR SINGLE-STEP ENTRY
     DC.L MONITOR_ENTRY
* ORG $24        ---TRACE---
    DC.L T_ENTRY
* ORG $28        ---1010 EMULATOR---
*    DC.L EA_ENTRY
* ORG $2c        ---1111 EMULATOR---
*    DC.L FE_ENTRY
* ORG $34        ---CP PROTOCOL VIOLATION---
*    DC.L CPV_ENTRY
* ORG $38        ---FORMAT ERROR---
*    DC.L FT_ENTRY
* ORG $3C        ---UNINITIALIZED INTERRUPT---
*    DC.L UI_ENTRY
* ORG $C0        ---FPCP UNORDERED CONDITION---
*    DC.L FBUC_ENTRY
* ORG $C4        ---FPCP INEXACT RESULT---
*    DC.L FIR_ENTRY
* ORG $C8        ---FPCP ZERO DIVIDE---
*    DC.L FZD_ENTRY
* ORG $CC        ---FPCP UNDERFLOW---
*    DC.L FU_ENTRY
* ORG $D0        ---FPCP OPERAND ERROR---
*    DC.L FOE_ENTRY
* ORG $D4        ---FPCP OVERFLOW---
*    DC.L FO_ENTRY
* ORG $D8        ---FPCP SIGNALING NAN---
*    DC.L FNAN_ENTRY
* ORG $E0        ---PMMU CONFIGURATION---
*    DC.L PMC_ENTRY
* ORG $E4        ---PMMU ILLEGAL OPERATION---
*    DC.L PMIO_ENTRY
* ORG $E8        ---PMMU ACCESS VIOLATION---
*    DC.L PMAV ENTRY
```

Figure 3-1. Typical Processor Exception Vector Table

Activating Processor Exception Vectors

Emulation monitor programs are normally shipped from the factory with all of the exception vectors (except RESET and MONITOR SINGLE STEP) contained in comment fields. This is done to allow you to supply the addresses for your own exception handler routines, if you have written any. If you have not written any exception handlers, remove the comment delimiters (*) from those provided in the processor exception vector table. This enables the emulator to use the processor exception vector table provided with your monitor program.

If your target application has its own RESET handler, you can modify the reset vector address in the processor exception vector table to point to the routine in your target code. You will also need to disable the reset-to-monitor function by answering "no" to the appropriate emulation configuration question.

Often, the portion of the monitor containing the processor exception vector table is not relocatable, as is the rest of the monitor. If the processor exception vector table of your monitor must reside in a specific address space, be sure to map the appropriate block of memory to emulation RAM (or to target-system ROM, if permitted by your emulator).

Modifying The Processor Exception Vector Table

Use your editor to remove the comment delimiters (*) from the start of each line of code that you want to be active in your processor exception vector table. Lines that begin with comment delimiters will be ignored during execution.

End out of your edit session, making sure that you save your changes.

By removing the comment delimiters, you have made the exception vector table usable. The processor exception vector table points to addresses of the appropriate emulation monitor entry point routines.

Entry Points Into The Monitor

Emulation monitor entry points are input routines for the various entry paths into the monitor code. The following paragraphs describe typical monitor entry and input handler routines.

Break_Entry

This is the entry point taken when you use the Break command to transfer control from your target program to the monitor program. When a break to BREAK_ENTRY occurs, the program counter and status register of your emulation processor are saved on the stack, as is normally done when an exception occurs. The emulation monitor proceeds to the monitor loop where it waits for a command from the keyboard.

During the time that the monitor is in use, the emulation processor may move code into target memory, or perform some other task that affects the state of the emulation processor. By storing the entry values of the emulation processor, these same values can be restored later so that execution can be returned to the point where it was interrupted in the target program when you have finished using the monitor functions.

Bus_Error_Entry and User_Entry

These entry points are taken when the emulation processor detects either a bus error or an address error exception. The only difference between these entries and the BREAK_ENTRY described above is that some additional words required to understand these exceptions are saved in variables.

Software_Breakpoint_Entry

This entry point is taken when a software breakpoint is processed. You might use a software breakpoint when you want to begin single-stepping through your target code after several target-system processes have begun. You can set the software breakpoint at an address in your code following startup of these activities.

The emulator processes a software breakpoint by removing and storing the instruction that resides at the selected address and replacing it with a BKPT instruction. When the BKPT is found, execution enters the monitor loop and waits for a keyboard

command. At the same time, the emulator discards the BKPT instruction and restores the previous instruction that was at that address. Note that you can't set a software breakpoint at any address mapped to target ROM.

Reset_Entry

This entry point is taken when the emulation processor executes the reset exception. RESET_ENTRY returns the stack pointer to the first address in the stack, and sets all of the emulation processor's registers to default values. Then the monitor loop is entered, and the emulator waits to detect a new command.

Exception_Entry

A set of exception entry points allow the emulator to display status messages for the exception vectors after reset. These exception vectors are provided for your convenience, and may be deleted or modified. For more information on the exception vector entry points, refer to the paragraph in this chapter titled "Modifying The Exception Vector Table".

Emulation Command Scanner

Command scanning is done in the monitor loop, an idle loop that continuously tests for the existence of a keyboard command. When a keyboard command is found, the corresponding command routine is executed. Depending on which keyboard command was issued, execution may end with a return to the monitor loop to await further commands, or the state of the system may be restored to its pre-break condition, and execution may resume where it left when the break was detected.

Command Execution Modules

The following paragraphs describe typical emulation command execution modules that reside in a monitor.

Are_You_There?

An are_you_there routine is used by the emulator to determine whether the emulation processor is executing in the monitor or in the target system code. The are_you_there routine can also pass an

ASCII message to be displayed on the status line of the emulation display, such as, "Running in monitor."

Exit_Mon

This is the path to use when exiting the monitor. An exit_mon routine will reload the processor registers from the variables that were stored when the monitor was entered. The program will then exit the monitor and resume execution of the target system code.

Copy

The copy routine moves data between the monitor and target-system memory. This command is used to modify and display target-system memory.

Monitor Routines You May Want To Modify

The following paragraphs describe examples of modifications you may want to make to a monitor program. Not all of these selections are available in every foreground monitor.

Read/Write Target Memory Space

You may want to change the size of the monitor transfer buffer. Some foreground monitors allow you to change the size of this buffer to obtain increased performance during target-system accesses. Monitors that allow these changes will precede the global symbol for the buffer size (typically called MON_XFR_BUF) with suggested values and value limits.

Display/Modify Registers Of A Coprocessor

You may have written a routine to load values into coprocessor registers. The monitor offers an easy way to identify the location of that routine so it will be executed by the emulator when you want to load the registers. The monitor source file will show a modification you can make so that the emulator will execute your routine when desired.

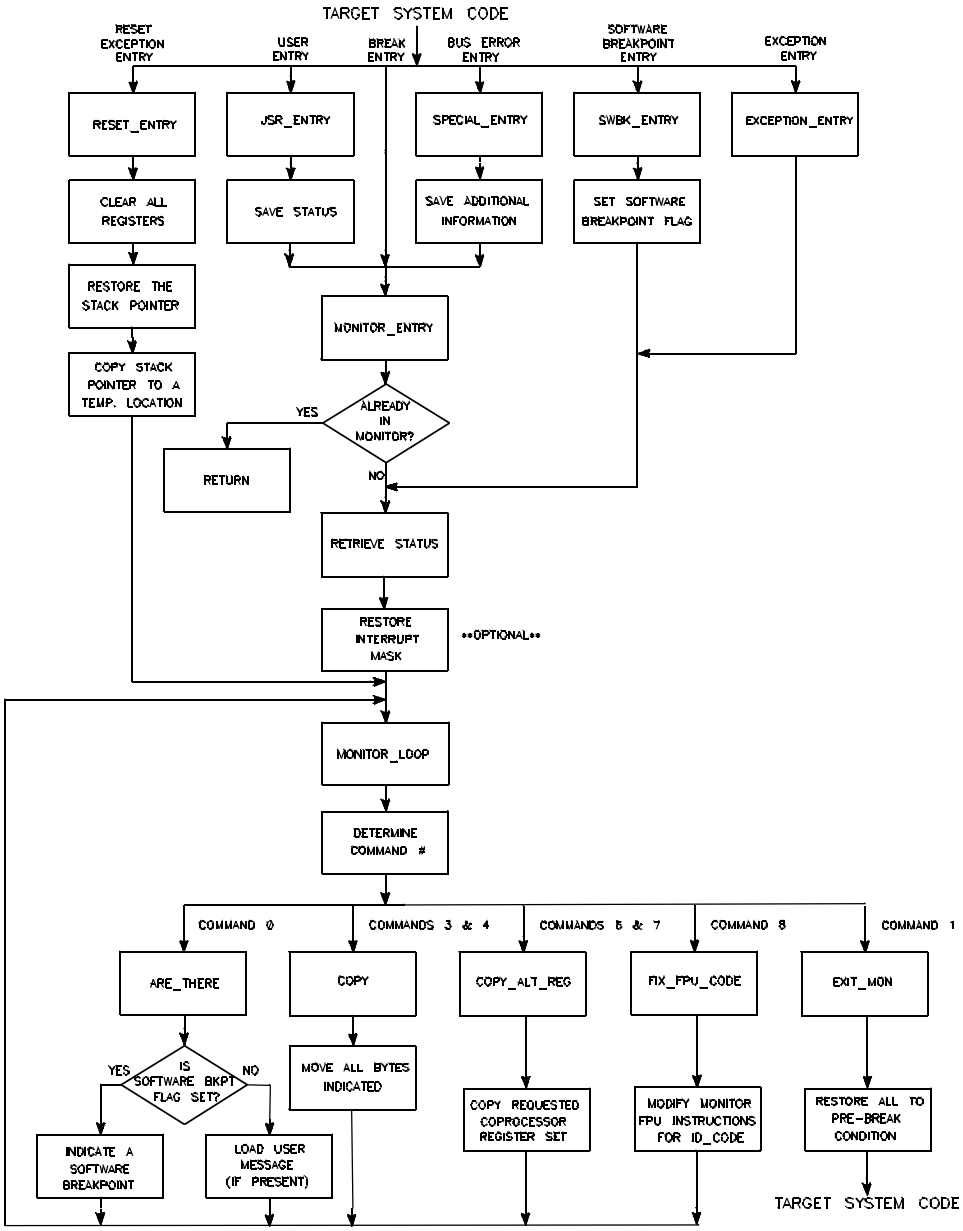

Figure 3-2. Emulation Monitor, Typical Block Diagram

Bus-Error/Address-Error Storage

You may want to modify your monitor so that it will save additional information about the state of the emulator when a bus error or address error occurs.

Messages Displayed On The STATUS Line

You may want to change the words of an existing message, or create some new messages to be displayed on the status line.

Simulated Interrupts For Out-Of-Circuit Emulation

Some monitors allow you to set up parameters so you can force the emulation processor to branch to your interrupt-handler routine and run it during execution of your target program.

Exception Vectors, Selecting Your Own

The monitor program may contain a list of exception vectors that are preceded by comment delimiters. If you would like to enable any or all of these exception vectors, simply edit your copy of the monitor source file to remove the comment delimiters.

Monitor Routines You Should Not Modify

Normally, you should never modify the monitor routines that provide the following features for the emulator:

Display/modify registers of emulation processor

Execute your target program

Break into monitor from target program

Reset into monitor

Single-step program instructions

Break away from target program

Linking The Emulation Foreground Monitor

The emulation foreground monitor must be assembled and linked before it can be used by the emulation system. Depending on the emulator you are using, there may be special constraints imposed on your foreground monitor. Below is a list of some of things you should consider before linking your emulation foreground monitor:

Some emulators allow you to link the foreground monitor with your target code to form a single executable module. Linking the monitor separate from the target code is preferred. Linking monitor programs separately is more work initially, but with separate linking, the monitor can be loaded efficiently during the configuration process.

Some emulators specify that the foreground monitor must be linked as a separate module and loaded first into memory before you can even load your target code. In these systems, one of the monitor routines performs the task of loading your target program.

In some systems, you tell your emulator the address of the foreground monitor by answering a single configuration question. With this information, the system can figure out everything else it needs to know to use the monitor routines.

In some systems, the processor exception vector table must be placed in a specific address range, but the remaining routines of the monitor can be placed in any address range you desire.

In one emulator, the foreground monitor must be placed entirely within a specific address range, and the foreground monitor for that emulator cannot be modified.

Loading The Emulation Monitor

By using options to the "load" command, you can load the monitor into emulation memory, and then load your target system code into target system memory. Load the emulation monitor code first.

Most foreground monitors reside in RAM space in emulation memory hardware. Prepare your load map so the emulation memory hardware containing the foreground monitor will be part of the same logical address space that contains the target program code. This way, the target processor will be able to run monitor code in the same space as its own target code (servicing interrupts and providing refresh cycles), even though the hardware that contains the monitor code is emulation hardware, not target-system hardware.

Memory Requirements For Emulation Foreground Monitors

The size of the relocatable portion of a foreground monitor differs from one emulator to another. You can determine the size of your emulation monitor if you look at the MODULE SUMMARY section of its linker listing file.

When you load your foreground monitor into memory, start the monitor on the first address in a new memory block. The emulation system divides the available memory hardware into blocks of equal size. Some emulators allow you to specify the size of the blocks desired (typically from 256 bytes per block to 4K bytes per block). Other emulators divide all memory into one block size (typically 256 bytes per block).

Foreground monitors should reside in RAM space in emulation memory. The emulator cannot always access target-system memory, but it can always access emulation memory. Some emulators have additional requirements (such as function codes) for the kind of memory where they reside. Refer to your emulator User's Guide for specific requirements for your monitor.

Some portions of the foreground monitor may need to be mapped to specific address ranges, such as the processor exception vector table of certain monitors. Sometimes the entire foreground monitor must be located in a specified range of address space. These restrictions vary from one emulator to another and may not apply to the foreground monitor you are using.

Emulators for processors that use memory management units (MMU) may need the monitor located in address space where logical address= physical address so that monitor routines are not affected by translations made by the MMU.

Notes

4

The Emulation Configuration

What Is An Emulation Configuration File?

The emulation configuration file is a file containing a series of questions and answers that define the kind of support to be provided by the emulator. Many aspects of the way your emulator operates are governed by its configuration file An emulation configuration file is supplied with a set of default answers to its questions. You can select different answers to these questions, if desired.

Loading the configuration file is one of the first things you do when you gain access to your emulator. The configuration file sets up the operating mode you desire. Aspects of emulator operation that are usually governed by its configuration file include:

- How your emulator to handles various conditions it may encounter during your tests.
- How your emulator manages the memory hardware it has available.
- How resources are shared between your emulator and your target system.
- How the emulator and target system interact.
- Which operations are enabled in the emulation environment.

Emulators are supplied with default configuration files that answer the questions in ways that are typical for most applications. You can modify the configuration file to match the needs of your target system by invoking the series of emulation configuration questions and providing your own answers to the questions when they appear on your workstation. After modifying the emulation configuration

file, you can save it under a filename of your own. Then you can load your new configuration file instead of the default configuration file each time you enter emulation (example: **load configuration myconfig** RETURN).

The following conditions are typically managed by the emulation configuration file:

- Selecting real-time or non-real-time emulation mode.
- Enabling breaks to the emulation monitor.
- Selecting whether to reset into the emulation monitor or to use the user reset exception vector.
- Configuring the foreground and background monitors.
- Setting the software breakpoint instruction.
- Configuring custom coprocessor functions.
- Configuring memory.
- Configuring the emulator probe.
- Configuring simulated I/O and simulated interrupts.

Modifying The Emulation Configuration File

To modify the emulation configuration file, invoke the file from within emulation by entering a command such as:

modify configuration Return

The first question in the series of questions will appear on the screen of your terminal. When you answer it, the next question will appear. Each question is displayed with its present answer. You can continue to use the present answer to a question (present setup

for that aspect of the configuration) by pressing the Return key. Optional answers to each configuration question can be selected by pressing an appropriate softkey or by typing in the desired answer on the command line, and then pressing Return.

Note

If you need to back up to a question you already answered, press the RECALL softkey. Each time you press RECALL, the emulator will back up one configuration question.

View The Configuration File And Its Present Answers

An incorrect configuration file may cause improper operation of your emulator. Review the entire configuration file to make sure all of the questions are answered correctly. If you are not sure how to answer a particular question, refer to your emulator user's guide for details of the consequences of selecting each answer to a particular configuration question.

Enter the command "**!more < configfilename> .EA**" to view the entire configuration file, along with its present answers.

The Configuration File To Use When Starting Your Target System

Target systems that can operate with the target microprocessor instead of the emulation pod should be able to start without difficulty when you load the default configuration file that was supplied with your emulator. Use the default configuration file whenever you start a new emulation session. The default configuration file enables all of the target system signals, maps all memory as target RAM, and specifies that the emulation monitor is not loaded. Verification of proper operation should be made using the emulation bus analyzer and indications from the target system. When you first connect the emulator to the target system, correct any failures you find using a minimum emulation configuration, no emulation memory or emulation monitor. Once

the default configuration works properly, you can add emulation memory and an emulation monitor.

5

Memory Mapping

This chapter discusses how to map memory for an emulation session. When you map memory, you identify the memory hardware that will be used during the emulation session, and you specify the absolute code modules that will reside in that hardware. To support these specifications, two maps are required. The use of each map is described below:

- Using the emulation memory map (part of the emulation configuration file):
  - You specify the size of each block of contiguous address space. The range of addressable memory of your target system will be divided into blocks of the sizes you specify (down to the minimum block size your emulator will support). Some emulators allow you specify their minimum block size.
  - You characterize the hardware that will support each block of contiguous address space. You might identify the first block of addresses as ROM space. The next block of addresses may reside in RAM hardware. Some emulation memory maps allow additional characterization, depending on the capabilities of the target processor.
  - You identify the location of the memory hardware that supports each address block. One block of addresses might reside in emulation memory. Another block of addresses may reside in hardware on the target system.
- Using the linker load map:
  - You identify the code modules (by name) that will reside in each of the address blocks.

The way the emulator uses the emulation memory map is:

- The emulator manages each block of addresses separately. Emulators limit the number of separate blocks they can manage. You need to select a block size that avoids exceeding this limit. Usually this limit is equal to the number of spaces available for entries on the emulation memory map.
- The emulator will allow reads and writes to addresses in blocks you've characterized as RAM space, but it will not allow writes to addresses in blocks you've characterized as ROM space.
- The emulator will turn on one set of address buffers if the present address is part of a block that resides in the target system. The emulator will turn on a different set of address buffers if the present address is part of a block that resides in the emulation memory.
- The emulator interface software uses the linker load map when it loads the target code into memory. It loads each of the target modules into the address space you specified in this map.
- Any block of memory addresses can be specified as guarded space (meaning these addresses don't exist. An attempt to read or write to guarded memory space is an illegal condition). The emulator will try to break to the emulation monitor if your target program tries to access guarded memory. This can be used to halt a runaway program.

How The Two Maps Depend On Each Other

The purpose of the following discussion is to introduce the emulation memory map and the linker load map, and show you how these two maps work together to load your target program into memory hardware.

What Do You Do When You Map Memory

Using the emulation memory map, you allocate and characterize memory space available in your hardware. The linker load map is where you specify which code module is going to reside in the memory address space available. Each of these maps is described separately in the following paragraphs.

Example Emulation Memory Map

Figure 5-1 shows a simplified emulation memory map. The exact format of the emulation memory map in your emulator may be different, but it will carry the same kind of information as shown in figure 5-1.

The emulation memory map shows a list of address ranges. Beside each address range, the emulation memory map shows whether the memory hardware that will support that range is located in the emulator or in the target system. Finally, it shows whether that hardware is to be treated as RAM or ROM.

```
Entry    address range          location/description
1          0 - FF                 EMUL/ROM
2        100 - 1FF                TARGET/RAM
3       1000 - 11FF               EMUL/RAM
4       1200 - 12FF               EMUL/RAM
5       2000 - 23FF               TARGET/ROM
```

Figure 5-1. Simplified Emulation Memory Map

Managing The Emulation Memory Hardware

In figure 5-1, some emulation memory is specified as ROM space, and some as RAM space. All of the hardware on the emulation memory board is RAM hardware, but your map specification will control the way the emulator manages access to the memory. If you specify an entry as RAM space, the emulator will allow reads from and writes to that space during runs of your target program. If you specify an entry as ROM space, the emulator will allow reads from that space during a run of your target program, but not allow writes to that space. If a write is attempted to space you've mapped as ROM space, the emulator will signal that an illegal condition has occurred.

Emulation Memory Architecture

Figure 5-2 is a simplified schematic showing the logical architecture of the emulation memory board assembly. The memory mapper hardware is set up to support each entry in the emulation memory map (figure 5-1 in this example). Each entry is explained below:

Entry 1 in the emulation memory map is the 256-byte range of addresses from 0 through 0FFH. For this example, it has been mapped as Emulation ROM. When the mapper hardware receives any address in this range, it performs an address translation to map the address into the appropriate space in emulation RAM. It also causes the attribute memory to supply the appropriate control signals to the emulation run control so that the emulation cycle will be treated as an access to ROM (no write transactions allowed) and as a read from the emulation memory.

Entry 2 in the emulation memory map is the 256-byte range of addresses from 100 to 1FFH. It is target ROM. The mapper hardware will not perform an address translation. The attribute memory will send the signal "target ROM" to the emulation run control. The emulation run control will turn on the appropriate buffers to send the address to hardware in the target system. The emulator will manage each address according to its rules for accesses to ROM space.

Entries 3 and 4 are mapped to emulation memory. The mapper hardware will perform the appropriate address translations to map each of the addresses to the appropriate spaces in emulation RAM. The attribute memory will send the appropriate signals to the

emulation run control so that each address cycle will be managed as an access to RAM (as specified in the emulation memory map).

Entry 5 in the emulation memory map is a 1-Kbyte range of address space in target ROM. Again, the mapper performs no address translation. The attribute memory sends the code for "target ROM" to the emulation run control, and it sends the address to target-system hardware. The emulator manages each address according to its rules for accesses to ROM space.

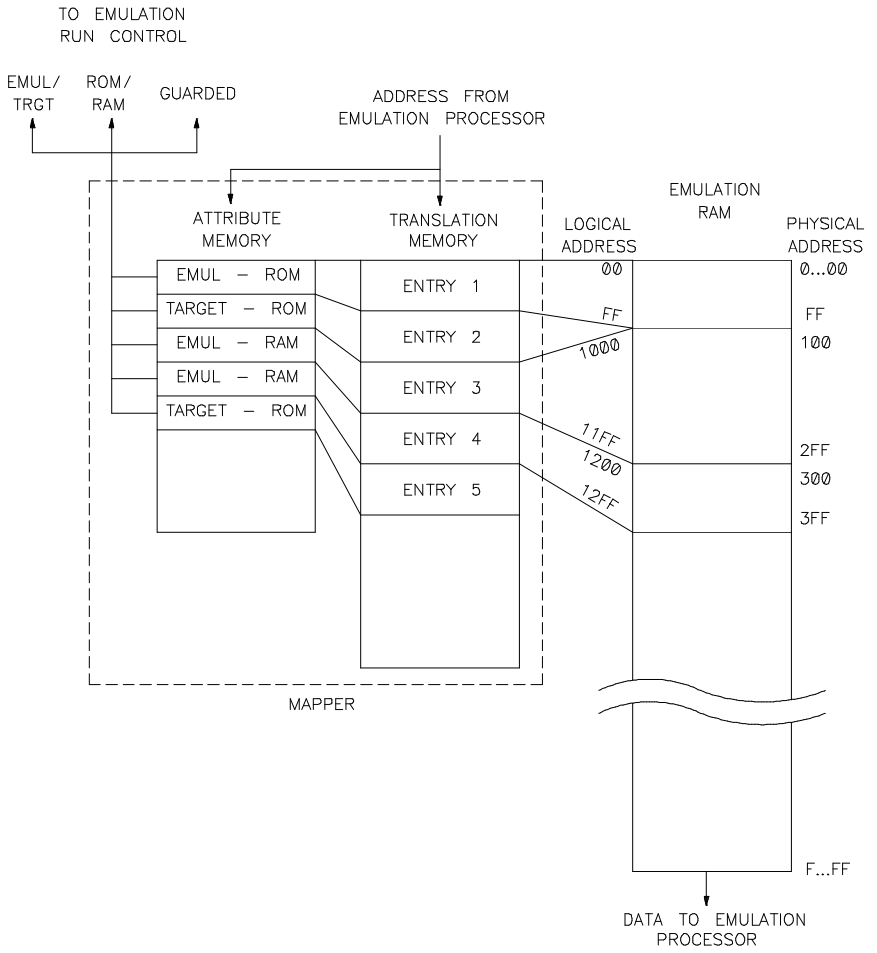

Figure 5-2. Emulation Memory Architecture

Note

The preceding discussion explains why emulation memory is expensive. It has to be faster than target memory to allow time for the translations that must be performed by the mapper. If the target memory must respond in 80 nsec, then the emulation memory must typically respond in 35 to 40 nsec.

Example Linker Load Map

Figure 5-3 shows an example linker load map. This map specifies the addresses where each module of your target code will be loaded in memory. Many linker load maps show only the address of the first byte of code of each module. This example assumes that module 1 will be no larger than the 256-byte space available in entry 1 on the emulation memory board (figure 5-2). Module 3 will be no larger than the target memory space allocated to entry 5 on the emulation memory map (figure 5-1). The monitor is no larger than the 512-byte space made available in entry 3 of the emulation memory map.

Note that the mapper can use any block of emulation memory hardware to support any range of addresses. This flexibility is not available when mapping target memory. The addresses supported in target hardware depend on how you wired your target system.

The linker load map must be in agreement with the emulation memory map. If you set up the linker load map to place a code module at addresses 1000H through 10ffH, then you need to make sure your emulation memory map allocated memory hardware to support that address range. If you decide to place data within an

```
Code Module Name         Load Address
Module 1                    0000
Module 2                    0100
Module 3                    2000
Monitor                     1000
Module 4                    1200
```

Figure 5-3. Simplified Linker Memory Map Content

address range, and your program is intended to write to that data space, then you need to specify that address range as RAM space in your emulation memory map.

Note

For complex programs, it may be helpful to have a copy of your linker load map when you are setting up your emulation memory map. You can use it to make sure you provide hardware for every software module in your linker load map.

Special Considerations For The Emulation Memory Map

The following considerations apply when filling out most emulation memory maps.

Selecting The Best Memory Hardware To Use

As soon as target system memory becomes available, you should include its address space in the emulation memory map so you can load your program into it and use it for project development. Normally, target system memory hardware will have response characteristics that are different from the response characteristics of the emulation memory hardware.

Dividing Memory Into Blocks

A single address block (entry on the emulation memory map) can be as small as a single page. (The page size used in the example in this chapter is 256 bytes.) An address block can also be as large as the entire memory space available.

Each address block can have one memory characterization. An emulator cannot manage two different memory characteristics within one block of addresses (it can't treat one-half of an address block as ROM and the other half of the same address block as RAM).

The ability to select from a variety of choices for page sizes can be useful in emulators that limit the number of separate entries that can be maintained in an emulation memory map. The page size you specify will apply to all memory in the map.

By specifying a particular block size (having one or more pages), you determine how many separate address spaces must be managed by the emulator. Each address in a block of address space will be treated as though it resides in the type of hardware you specify for the block: RAM or ROM.

When you enter a specification on your emulation memory map, the emulation mapper will assign at least one page of hardware addresses to support that specification. This is true, even if there is only one or two bytes in the code module that will occupy that block. The emulator cannot allocate memory space smaller than the page size you specify.

Three Kinds Of Memory In The Emulation Memory Map

The emulator, can manage three kinds of hardware: RAM, ROM, and guarded.

- RAM space is any memory address space that allows random access for both read and write transactions by the emulation processor.
- ROM space is any memory address space that allows random access for read transactions, but no access for write transactions. If the emulation processor (processor in the probe that replaces the target processor) tries to write to space you've characterized as ROM, the emulator will detect this event as an illegal transaction, and place a warning message on the status line of the display. Note that the emulation control board can write your program into ROM space. Only the emulation processor (in the emulation probe) is prevented from writing to memory you've characterized as ROM space.
- Guarded space is any memory address space you are not using (whether or not memory hardware is available to

support that space). Address space characterized as "guarded" should never be accessed by the emulation processor. This designation normally identifies non-existent space. If the emulation processor ever tries to read from or write to memory space that is characterized as guarded, the emulation control board will detect this as an illegal event and place a warning message on the status line of your display. (Note that no emulation memory hardware is required to support any address space designated as guarded space.)

One of the questions in the emulation-configuration set of questions asks if you want the emulator to break from your target program and begin executing in the monitor if the target program attempts to write to an address you've characterized as ROM space. If you answer yes, any attempt to write to ROM will cause a break to the monitor. If you answer no, the write instruction will have no effect on the execution of your target program. In either case, the content of emulation ROM space will not change. The content of ROM space in your target system may or may not change, depending on the design of your target system.

Caution

In some target systems, there is risk involved with electing to break to the monitor on an attempted write to ROM. If you have some target-system activity that should not be abandoned until it is complete, (such as a moving hydraulic arm that could over-travel if your target program suddenly jumped to the monitor) you might want to answer "no" to the configuration question that asks, "break on write to ROM?". (If using a foreground monitor, you might overcome this problem by modifying your monitor program.)

You can set up the emulation bus analyzer (if available) to trace the activity preceding and following an attempted write to ROM. This can be done by setting the trigger to occur on any address in the range you've specified as ROM space, and then specifying that the trigger be in the center of the trace memory.

If you have activated the monitor and are using non-real-time mode, an attempted access to guarded memory will cause execution to break from your target program to the monitor.

Any address ranges not mapped when the mapping session is terminated are asigned the memory default. The default attribute can be target RAM, target ROM, or guarded. Unless otherwise specified, the system defaults all unmapped address space to guarded.

Mapping A Foreground Monitor

Most foreground monitors can be mapped to any address space having the RAM attribute as long as its located in emulation memory hardware. A few foreground monitors require special considerations when mapping memory (such as placing them in a specified range of addresses).

Deleting Map Entries

Some emulation memory maps will not allow you to delete all of the map entries. They may reserve one range of addresses for a processor-specific usage, such as an exception-vector table for a foreground monitor.

Overlay

If you want a block of hardware memory to respond to two or more address ranges, use the overlay capability in the emulation memory map. When you overlay one address range onto another, you can characterize the emulation memory differently for each address range, if desired.

A typical use of the overlay feature would map a single 256-byte block to support two address ranges: 1000H thru 10FFH as ROM, and 1800H thru 18FFH as RAM. When the emulation processor addresses any location from 1000H to 10FFH, the emulator will allow reads but not writes. When the emulation processor addresses any location from 1800H through 18FFH, the emulator will allow both reads and writes. Therefore, the emulation processor could overwrite the content at any address in the 256-byte block if it addressed it using its 18XX address, but could not write to it at all if it addressed the same memory hardware using its 10XX address.

You can only specify memory overlay using emulation memory hardware, not hardware on your target system, but you can map a

block of memory space in your target system and then overlay it with a block of emulation memory hardware. The range of your overlay specification must be the same size as the range it is overlaying.

Additional Characterizations Available In Some Emulators

Some emulators allow you to specify a wide variety of attributes that can be included when characterizing blocks of memory. Your emulation memory map may allow you to include one or more of the following descriptions to further characterize your memory hardware:

- program code/data code
- processor function codes
- synchronous or asynchronous cycles
- 8-bit, 16-bit, or 32-bit data widths
- emulation memory whose cycles are interlocked with target system memory
- cache disables to turn of a processors cache when execution is in a particular address range

Problem When Specifying Certain Address Ranges

The problem discussed in this paragraph generates the message, "ERROR: Lower address in range greater than upper address." It results from the way an emulator processes the highest address in an address range specification. For most range specifications, the emulator accepts your specification as you enter it. The problem occurs when the highest address in an address range is the lowest address of an address block (e.g. xxxx00h), In this case, the emulator rounds down the address by one byte (e.g. xxxxffh).

If you attempt to specify an address range by using the same value for both the start and end addresses, and the address you choose is the first address in a new memory block (map 100h thru 100h emulation ram), the emulator will round down the ending address. When rounded down, the ending address (0FFH) is lower than the starting address (100H).

Notes

6

Command Files

This chapter provides the following information about command files:

- what is a command file?
- how are command files useful?
- methods used to create command files
- editing an existing command file
- how to execute a command file
- conventions for naming command files
- nesting and chaining command files
- making a command file fit into a script
- executing scripts automatically upon login
- things to remember when using command files
- what to do if your command file doesn't work

This chapter does not provide details of how to make a command file execute in every kind of system. For example, you can quote strings of characters to be passed as parameters during execution of your command file. Some systems allow you to use several different characters for quoting characters. Other systems allow you to use only quotation marks for quoting characters. You'll have to refer to your system user's guides and/or reference manuals to find out which quoting characters you can use in command files for your system.

What Is A Command File?

A command file is simply a file that contains commands. When you type the name of the command file on your interface, the commands are pulled one at a time from the file and executed in your system. Command files are useful because they allow you to create a setup or test that involves several command entries, and then perform that setup or test any time you want by simply typing the name of the command file on the command line and pressing the RETURN key.

Command files are similar to HP-UX scripts. The difference is that the commands in your command file must be native to the interface you are using. Also, command files must not have execute permissions.

How Are Command Files Useful?

Command files minimize the need to type on the keyboard, and they speed up the process of emulation because each new command is entered as soon as the system is ready to accept it.

You can use command files to duplicate measurements. For example, every time you want to assemble, link, load, and run a program, you can type the name of a command file that contains the appropriate assemble, link, load, and run commands. All of the steps will be executed automatically.

The following is a list of tasks that are easy to implement in a command file:

1. Compile, assemble, and link programs.

2. Enter emulation and do emulator tasks.

3. Invoke an external analyzer (State, Timing, or Software Performance), and perform analysis tasks, such as:

   a. Specify trigger conditions.

   b. Trace.

   c. Copy trace list to printer.

4. To run an emulation and analysis session, you enter a series of commands that gain access to emulation, load the different memories with your absolute code, run your code from selected addresses, and trace program execution using selected trigger states and storage qualifications. The commands you enter to perform these actions can be placed in a command file under a name you select.

   All you have to do to repeat the same session later is type the name of the command file on the command line. The system will execute the commands in the order they appear in your command file.

When you configure the emulator for an application, you answer a series of configuration questions. These questions and the answers you give are stored in a command file generated by the emulator.

Methods Used To Create Command Files

There are two methods to create command files: (1) automatic creation by recording the commands you use during a test, and (2) opening a file and typing in a series of command lines. Both methods are described in the following paragraphs.

Automatic Creation Of A Command File While You Use The Emulator/Analyzer

You can have the emulation system keep a record of the commands you execute while performing a test. (This is called "logging commands to a file"). In this way, you can concentrace on running your test without having to think about command files. When you finish your test, turn off the logging process. The file may be edited to view or alter the command list. Creation of a command file by logging commands ensures that the syntax of the commands is correct.

To turn on the process of logging commands, some interfaces let you enter a command such as:

log_commands to < filename> RETURN

Once you have done this, all of the commands you use will be put into a file whose name is the < filename> you specified in the command. Each command will be saved on a separate line.

When you finish your tests, enter the command to stop the logging process. Some interfaces use the command:

log_commands off RETURN

The command file is in your present working directory, ready for edit.

Things To Correct During The Edit

If you ended your session with the "end" command, that command will be the last command in your log file. You will probably want to delete this command. For other editing considerations, refer to the paragraph titled, "Editing An Existing Command File", later in this chapter.

Logging To Enlarge An Existing Command file

If you already have a command file with the same name as the file you are logging commands to, your new commands will be appended to the previous command file. To avoid this, accompany your "log_commands" entry with "noappend". When you specify noappend, the previous log file will be overwritten (if you have permission to write to the file).

Logging Calls To Other Command Files

If you call an existing command file to perform some activity while you have the "log_commands" process turned on, the logging process will turn off while the called file executes. Only the call to the command file will be logged, not the commands within the called file.

Logging Commands That Spawn New Processes

You cannot log commands that spawn new processes. For example, "!vi" in the command file will invoke the "vi" editor, but will not log any of the "vi" editor commands. When the "vi" editor is terminated, commands will again be logged to your file.

You can log single shell commands, such as "!ls" or "!asm -o sample.s > sample.o".

Using An Editor To Create Command Files

You can use the Softkey Driven Editor or any HP-UX editor to create command files. The syntax diagrams in your emulation and analysis user's or reference manuals will be helpful when composing your command lines. Proceed as follows:

1. Invoke the editor you want to use to create the command file.

2. Type in the desired commands. Make sure the commands you type are in the form that appear on the command line, not the abbreviations that appear on some of the softkeys. Also, make sure you use the correct command syntax.

Editing An Existing Command File

Before you can use an existing command file, it may need to be edited. Edit the command file just as you would edit any other file:

Scroll through the text. If you ended the log process by using the "end" command, the last line of your file will be "end". Delete this command.

Edit out any lines containing "!sh" in a command file. Otherwise, your command file will become suspended until you press < CNTL> d. After pressing < CNTL> d, your command file will resume.

Edit out any control characters in command files.

Note

Do not make command files executable. One difference between command files and HP-UX scripts is that scripts must have the file permissions set to "executable" before you can use them. Command files must not have execute permissions.

How To Execute A Command File

To execute a command file, you must be in an emulation/analysis session. You cannot start a command file from a shell prompt. However, you can create a shell script (an HP-UX file of commands that can be invoked from the HP-UX shell) that gains access to your emulation/analysis session, and then run a command file once that access has been obtained. You can also create an HP-UX script that enters an emulation/analysis session and then performs the actions of an embedded command file to complete a series of emulation and analysis tasks. These methods of execution are discussed in the following paragraphs.

Executing A Command File Alone

The most simple case of executing a command file is to type in the commands that gain access to your emulation/analysis system interface, and then type the name of the command file.

Using A Script And Command File Together

This case is a little more complicated because you must create two files. This case provides greater benefits because it reduces the number of commands you need to type in when performing your test. To do this:

1. Make a command file.
2. Make a script whose last entry accesses the emulation/analysis interface where you want to start your command file. Make the script executable (**chmod + x** < script_name>).
3. Type the name of your script at the system prompt.
4. Type the name of your command file when your script finishes executing and the desired interface is on screen.

Conventions For Naming Command Files

A command file name can have up to 14 characters. A name with more than 14 characters will be truncated. If your command file name begins with an alpha character, simply type the command file name and press RETURN to execute the file.

Startup8 RETURN

A command file name can begin with a number (e.g. 35test). When the first character of a command file name is a number, you must precede the invocation with a backslash.

\35test RETURN

Characters after the first in a command file name can be alpha or numeric. Most characters can be used in a command file name, except # | > < ; ( ) / and ’.

Nesting And Chaining Command Files

Nesting command files means calling a command file from within another command file. You can nest command files up to 50 levels deep.

Nesting Command Files

A command file call is similar to a subroutine call.

When a command file calls another command file, the calling command file must supply any parameters needed by the called command file. Otherwise, execution will stop. Once execution has started, you will not be prompted for missing parameters.

Chaining Command Files

Command files are said to be chained when the last line of a command file calls another command file. You can chain as many command files as you like. Chaining command files is preferred to nesting because nesting command files is not supported on some host systems.

Executing Scripts Automatically Upon Login

You can set up a script to execute automatically when you log in to the system. To do this:

1. Make the script executable.
2. Execute the script to make sure no errors occur. Fix any errors that occur. Then proceed.
3. Invoke your .profile file in an editor.
4. Add the script name to a line in your .profile file.
5. Save the .profile file, and exit the editor.
6. Log off the system. Log on again to verify that the script executes correctly.

Things To Remember When Using Command Files

The following general information will help you use command files effectively:

1. A command file that is started in an emulation/analysis session will continue to execute until an end-of-file is found, or until a syntax error occurs.
2. You can stop a command file by pressing < CNTL> c or the BREAK key.
3. Command files can contain shell variables if they begin with "$" and use an identifier. An identifier is a sequence of letters, digits, or underscores beginning with a letter or underscore. The identifier may be enclosed by braces "{}" or entered directly following the "$" symbol. Braces must be used when the identifier is followed by a letter, digit, or underscore that is not part of its name.

   For example, assume a directory named /users/softkeys and a shell variable "S" whose value is "soft". By specifying the

directory as "/users/${S}keys", the correct result (/users/softkeys) is obtained. However, if you specify the directory as "/users/$Skeys", the host will look for the value of "Skeys".

4. You can examine the current values of all shell variables defined in your environment with the command "env". Positional shell variables, such as $1, $2, and so on, are not supported. Special shell variables, such as $@, $*, are also not supported.

5. Command file lines can be longer than one screen width. Break long command lines by ending the line with a backslash (\). A line terminated by "\" is concatenated with the line that follows it. This continues until a line is found that does not end with a backslash. A line constructed in this manner is recognized and executed as one single command line. If the last line in a command file is terminated by "\", it appears on the command line but is not executed. Normally, the line feed is recognized as the command terminator.

6. Commmands in shell scripts can also be longer than one screen width. Do this by quoting the command. The HP 64000-UX environment recognizes three quoting characters for shell commands: double quotes ("), single quotes (’), and the backslash symbol (\).

   For example, the following three lines are treated as a single shell command. The two hidden line feeds are ignored because they are inside the two single quotes (’):

   ```
   !awk ’/$/ { blanks+ +  }

   END { print blanks }

   ’ an_unix_file
   ```

7. An HP-UX hosted linker may not work if it is invoked from a command file and has no argument specified. The linker looks for input from the keyboard. It cannot read the input from the command file.

8. Do not set the execute permission of a command file. If permission is set (**chmod + x** commandfile), the command file will not work.

If The Command File Does Not Work Properly

Command files execute properly when they have no syntactical or semantic errors. If an error is found, execution of the command file stops, and a message is displayed.

Command file execution will stop if:

1. The command file contains a syntax error.
2. An error is detected by the application program.
3. You press < CNTL> c or the BREAK key.

Notes

7

Coordinated Measurements Through CMB And IMB

High-speed communication is used to coordinate triggering, enabling and disabling of functions, and synchronization of execute and halt between two or more emulators and/or analyzers involved in a measurement. The coordinating signals pass through the CMB and/or IMB listed below:

CMB (The Coordinated Measurement Bus).

IMB (The InterModule Bus).

These two buses have differences, but the functions they perform are the same.

Differences include:

- The number of lines and types of signals carried.
- An emulator on CMB must use a background monitor. An emulator on IMB can use either a background or foreground monitor.
- You can interconnect a great many emulator/analyzer sets with CMB and only a few with IMB.
- The maximum overall length of the bus cable depends on which bus you are using.
- When using the CMB, external analyzers, such as the HP 1630 Logic Analyzer, can provide control signals to other emulator/analyzers on the bus. With the IMB, analyzers outside the card cage can only receive signals from the bus, not supply signals to the bus.
- When using the CMB, an analyzer outside the card cage can cause all emulators on the bus to break to their monitor programs. No such capability is offered on the IMB.

The CMB and IMB interconnect certain measurement functions so that activity in all of the participating emulators and analyzers can respond to the same control sources. Coordination offered when using the measurement buses includes:

- Synchronized runs of your target program.
- All analyzers accept the present state as their trigger when a designated analyzer recognizes its trigger event.
- One emulator on the bus can stop execution of your target program and break to its monitor program when an event is recognized by an analyzer in a different emulator/analyzer set.

IMB Information

The IMB is a 6-conductor cable that interconnects control signals for emulation and analysis through a special IMB plug at the top of all control boards. The coordinating signals on the IMB are described in the following paragraphs:

Master Enable

All analyzers that are to participate in a coordinated measurement receive this signal. This is the execute/halt signal. When it is true, it enables all analyzers that receive it. When it is false, it disables the analyzers. There can only be one driver for the master enable line: either the execute/halt function (the default configuration), or one of the analyzers assigned to participate in the measurement.

The purpose of the master enable line is to synchronize measurement start in all analyzers. At measurement start (execute key pressed or true state from designated master enable driver), each analyzer tries to start. The master enable line remains false until all analyzers are ready. Then it switches true, releasing all analyzers to start together.

When the master enable line is driven by one of the analyzers, it can alternate between true and false during a measurement to exclude unwanted activity from the coordinated measurement. When an analyzer receives this line from another analyzer, all of its analysis functions are disabled during disable periods.

Emulation_start The purpose of the emulation_start command is to couple an analyzer into a measurement with other emulators and analyzers that do not have, or are not using, any other IMB lines. When the analyzer you coupled is driving the emulation_start line or driving/receiving any other IMB line, it will start before any emulators in the coordinated measurement.

Trigger enable The trigger-enable line carries a logic level. When it is true, it enables the receiving analyzers to recognize their internal triggers, if they occur. When it is false, it disables trigger recognition in the receiving analyzers. The trigger enable line can alternate between true and false during a measurement to allow the controlling analyzer to window the activity where trigger recognition can occur. If no analyzer is designated to drive this line, it will default to the true state.

Trigger The trigger line carries a transition from false to true. When the trigger line switches from false to true during a measurement, it remains true for the rest of the measurement. An analyzer can drive the trigger transition when it recognizes its trigger specification. An analyzer can receive the trigger transition from another analyzer on the bus. A receiving analyzer will identify the present state as its trigger state when the trigger line switches true. The trigger line can have more than one designated driver. When more than one driver is designated, the first driver to recognize its trigger becomes the trigger source for the measurement in process.

Storage Enable The storage enable line carries a logic level. An analyzer can drive this line, controlling when the receiving analyzers can store information. If an analyzer receives this line, it will store states that meet its store specification when this line is true. The analyzer that controls this line can switch it between true and false during a measurement to window activity that can be stored. If no analyzer is designated to drive this line, it defaults to the true state.

Delay Clock The delay clock line carries a stream of clock pulses. Certain analyzers can generate this stream of clocks. Other analyzers (such as timing analyzers) can receive these clocks and use them to count delays when taking their measurements (delay by numbers of clocks).

CMB Information

The CMB consists of three bi-directional signal lines. These lines are available through a connector on the exterior of the card cage. The signals that coordinate measurements on the CMB are:

CMB Trigger line

READY line

EXECUTE line

Emulators and analyzers that coordinate measurements over the CMB offer commands you can use to:

- Enable and disable interaction on the READY line.
- Allow EXECUTE to start running your target program in emulation.
- Drive or received the CMB trigger line by any unit on the CMB.

Note

The CMB trigger line also goes true briefly following receipt of an EXECUTE signal. Because of the short-duration true state on the CMB trigger line, do not use it to trigger external instruments if the EXECUTE function is also being used.

CMB Trigger Line

The CMB Trigger line is low true. It can be used directly as a break source to the emulator, or it can be used indirectly as a break source through the internal trigger lines. When used as a break source, the driving function must be cleared before the emulator can resume running.

CMB READY Line

The CMB READY line is high true. It is open collector and performs ANDing of the ready state of all enabled emulators on the CMB. Each emulator on the CMB releases this line when it is ready to run. This line goes true when all enabled emulators are ready to run, providing a synchronized start. When CMB is

enabled, each emulator is required to break if CMB READY goes false, and will wait for CMB READY to go true before returning to the run state. When an enabled emulator breaks to the monitor, it will drive CMB READY false. The emulator that drives CMB READY false holds it false until it is ready to resume running the target program. When the emulator is reset, it also drives CMB READY false.

CMB EXECUTE Line

The CMB EXECUTE line is low true. Any emulator or analyzer on the CMB can drive this line. It serves as a global interrupt, and is accepted by analyzers and emulators.

BNC Internal-To-External Connection

The BNC input on the card cage can either supply the analyzer trigger to external equipment, or receive an analyzer arm, an analyzer trigger, or an emulator break request from external equipment.

CMB Interaction With External Analyzers

You can make measurements that include an external logic analyzer or oscilloscope connected to the CMB bus through a connector on the card cage (some card cages use BNC connectors, others have special connectors for this purpose). The lines on the CMB are bi-directional so you can use them to drive an external device, or receive arming or triggering signals from an external device.

Through command choices, you can specify which of the emulators and analyzers will drive and receive the trigger signals. This allows flexible interconnections between the emulation and analysis systems installed in the card cage and analyzers outside the card cage that are connected through CMB or BNC ports.

Notes

8

Introduction To Analysis

An analyzer is an instrument that captures signal activity synchronously with a clock signal. An analyzer can display a history of the signal activity over the period of the measurement. If the period of the measurement was greater than the capacity of the analyzer memory, the most recent signal activity will be shown. There are several types of analyzers used with emulators. These types are described below.

Types Of Analyzers

Emulation Bus Analyzer

An emulation bus analyzer captures bus cycle information from the address, data, and status buses of an emulation processor in sync with the emulation processor clock. The states captured show a history of activity on the emulation processor bus. This history can be presented as a list of states expressed in numerical values (hexadecimal, binary, etc), or it can be inverse assembled into a list similar to an assembly language program listing.

External Analyzer

An external analyzer captures activity on signal nodes that are external to the buses of the emulation processor. To use an external analyzer, you connect the external analyzer probe cable to signal nodes in your target system. The external analyzer probe cable is a set of signal lines, each with an individual probe connection. External analyzers can make state measurements and/or timing measurements, as described below.

External State Analyzer

An external state analyzer can use the same clock as the emulation bus analyzer so that the states at the signal nodes can correspond to the states captured by the emulation bus analyzer.

External Timing Analyzer

An external timing analyzer can record logic levels at the nodes you selected at each occurrence of a sampling clock within the analyzer. You select a clock rate that provides the measurement resolution desired. The clock rate must be faster than the data rate to prevent the analyzer from detecting a glitch on the occurrence of a single data bit. (A glitch is both a positive and a negative transition occurring between any two sampling clocks.)

Basis Branch Analyzer

This is a testing method that determines the extent to which branches are executed during runs of your target program. The measurement tells you if each possible path of the branch was executed during the test. You can use this type of analyzer to ensure that your tests do a thorough job of program testing. This analyzer identifies branches that are not completely tested so you can write new tests to verify paths not taken. HP offers its HP Branch Validator to make basis branch analysis tests.

Coverage Analyzer

A coverage analyzer measures the percent of memory accessed by a target program. A coverage analyzer reserves one bit in its memory for each byte of HP emulation memory. When a byte of emulation memory is accessed (written to or read from), the corresponding bit in the memory of the coverage analyzer is set. After a test, the coverage analyzer can show you the memory locations that were accessed.

Software Performance Analyzer

A software performance analyzer measures performance characteristics of code modules. You can use software performance measurements to compare code modules with each other to see how much execution time and state activity each one uses in your program. These measurements help you identify code modules that need to have their performance optimized.

Activity measurements record information about memory activity and program activity to tell how much execution time is spent and how many states are executed by each function and variable defined in the program under test. Activity measurements are also used to compare these functions and variables with one another to determine their relative use of time and states. Duration measurements record the time spent executing selected functions and procedures. You can use duration measurements to determine the average time it takes to execute a program module, and detect events when the duration of a module is abnormally different from its average duration.

Specifications Needed To Set Up A State Analyzer

Analysis functions may include trigger, storage, and count directives. The analyzer can capture a number of states in its memory, depending on the capacity of its memory. These states will include address, data, and conditions of the status bits.

Trigger

A trigger is a point of reference within the trace memory. It may be some unique state and you want to see the activity that led up to its occurrence. It may be some starting event and you want to observe the activity that follows after its occurrence. If your measurement is not keyed on the capture of some significant event, specify a trigger of any_state.

Store

The store function sets up the analyzer to store states of interest in its trace memory. The default of the store specification (any_state) sets up the analyzer to store every state, regardless of its content. When you are interested in analyzing the activity associated with a particular address range, you can store-qualify just that address range, and the analyzer will store only states executed within that address range. If you want to observe a series of writes to a variable, you can store-qualify write transactions to the address of the variable, and the analyzer will store only those write states.

Count

The count function measures periods of time or numbers of state transactions between states stored in memory. You can set up the

analyzer count function to count occurrences of a selected event during the trace, such as counting how many times a variable is read between each of the writes to the variable during a trace. The analyzer can also be set up to count elapsed time, such as counting the time spent executing within a particular function during a run of your target program.

Special Considerations

Chapter 9 of this manual discusses the way your measurements may be affected by operation of an internal cache (for processors equipped with a cache). Chapter 11 discusses difficulties you may have, and ways to overcome those difficulties, when analyzing the trace lists obtained from processors that prefetch instructions and use an instruction pipeline. Chapter 10 discusses a special measurement that some analyzers can make to help you find the cause of problems that may appear during execution of your target program.

9

How An On-Chip Cache Affects An Analyzer

What Is An On-Chip Cache?

An on-chip cache is an area of memory that is used to store recently used instructions and/or data on a microprocessor. Some microprocessors are equipped with caches for instructions only, while others have caches for both instructions and data. The purpose of an on-chip cache is always the same - to improve efficiency of program execution by the microprocessor.

By keeping recently used instructions and/or data in an on-chip cache, a microprocessor can access these items if they are used again without having to initiate external bus cycles. When the microprocessor fetches an instruction or accesses some data, it checks to see if that instruction or data is already in the cache, and if it is, the microprocessor loads it from the cache and does not perform any external bus cycles.

How Does A Cache Affect Analysis?

An emulation bus analyzer can only trace activity that appears on external processor buses. When the microprocessor is operating with its cache(s) enabled, analysis of processor activity is limited because no bus cycles are performed to fetch items already in the on-chip cache. Therefore, transactions involving the cache will either be incomplete or will not appear at all in trace lists.

Disabling And Enabling The Cache

Usually, you will want to disable the on-chip cache of a microprocessor when tracing execution of a program. In this way, all of the instructions executed by the microprocessor will be fetched on the processor memory bus and can be captured in the analyzer trace. Typically, emulators built for microprocessors that have on-chip caches will have a selection within the emulation configuration that you can use to disable or enable the cache(s).

Sometimes you may want to perform analysis with the cache(s) enabled. This will be the case when you want to measure speed of execution because on-chip caches can greatly increase the speed of execution of some programs.

10

Prestore Trace Measurements

The word "prestore" sounds like something is being stored before it happens. Obviously, that's not possible. What's "prestore" really mean? Prestore is a measurement mode that lets the analyzer capture the last occurrence of some event every time a new state is stored in the trace memory (you might prestore the last write instruction before data is written to a selected variable). Here's how prestore works.

To make a prestore measurement, the trace memory of the analyzer is divided in half. One half of the memory stores states that meet the store qualification. The other half of the memory stores states that meet the prestore qualification.

Figure 10-1 shows imaginary men working inside your analyzer to make prestore measurements. Each state that enters the analyzer

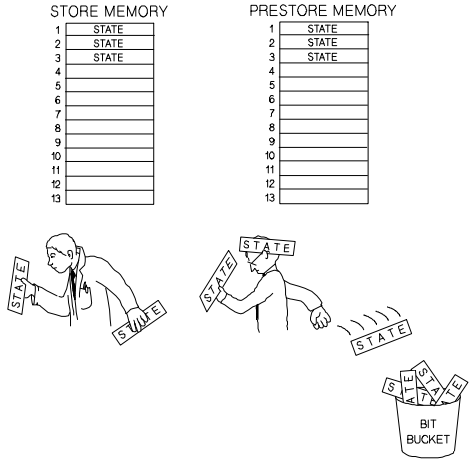

Figure 10-1. Making The Prestore Trace Measurement

is examined by Store Man. He looks to see if the state meets the store qualification you set up before the trace began. If the state meets the store qualification, it is stored in the trace memory, as is described later. If the state doesn't meet the store qualification, the state is passed to Prestore Man. Prestore Man examines the state to see if it meets the prestore qualification. If it does, Prestore Man places the state behind his ear and throws away any state that he had been holding there before. If it doesn't meet the prestore qualification, he throws the state in the bit bucket.

This process continues, with Store Man looking at each state, and then passing the state on to Prestore Man. Finally, a state comes in that meets Store Man's qualification. He places that state in the store memory. When this happens, Prestore Man places the state he has behind his ear into the prestore memory.

Note 

Each time Store Man places a state in store memory and Prestore Man places the last state he held behind his ear in prestore memory, then a new measurement begins. If the first state examined by Store Man meets the store qualification, no prestore state will be stored. Therefore, no prestore state will be shown in the trace list preceeding it.

When the trace is complete and you call for a trace list, the content in store memory and prestore memory are arranged as shown in figure 10-2.

```
TRACE LIST
PSTORE       STATE  (state 1 in the prestore memory)
+0001        STATE  (state 1 in the store memory)
PSTORE       STATE  (state 2 in the prestore memory)
+0002        STATE  (state 2 in the store memory)
PSTORE       STATE  (state 3 in the prestore memory)
+0003        STATE  (state 3 in the store memory
```

Figure 10-2. Making A Prestore Trace List

How Is A Prestore Trace Useful?

A prestore trace is useful when some error is being made to a point in your program, and there are several possible sources of the error. For example, assume there are several program modules that write to a variable, and sometime during execution of your program, that variable is getting bad data written to it. Using a prestore measurement, you can find out which module is writing the bad data. You can store-qualify writes to the variable. Using prestore, you can capture the instructions that caused those writes to occur.

The prestore measurement is used to answer such questions as, "Who is writing to this variable?", and "Which modules are calling this module?"

The information available in a prestore trace list can also be found by reading a default trace list, but you might have to read many thousands of trace list lines to find the number of transactions of interest that are shown on just one screen of a prestore trace list.

Setting Up The Analyzer To Make A Prestore Trace

When you set up the analyzer to make a prestore trace, you store-qualify some event of interest. and then prestore-qualify a related event.

Store Qualification

You might store-qualify events such as:

1. Store the entry to a selected code module.
2. Store the address of a variable.
3. Store an interrupt request.

Several different events can be store-qualified during a single prestore trace. This lets you test for the causes of several different errors in the same trace.

Prestore Qualification

In the prestore memory, you qualify a kind of action that affects the store-qualified event. This may be:

1. Store calls to a code module.

2. Store fetches of instructions. (These can show which module accessed a variable, or where code was executing when the event was stored).

Here, again, several kinds of transactions can be prestore-qualified during a trace.

Reading Prestore Trace Lists

When the prestore trace list is displayed, it will be a list of trace memory line numbers, preceded by prestored events. The trace memory line numbers will identify the store-qualified states, and the "pstore" lines preceding them will show the last prestore-qualified state that was captured before the stored event.

With a specification such as "STORE_ON a= 4000H", and "prestore on program read", you'll get a trace list that shows all the accesses to address 4000H. Preceding each access to 4000H, the list will show the most recent program read instruction.

```
+0001  00004000 ............
pstore ........ ............  (last program read before +0002)
+0002  00004000 ............
pstore ........ ............  (last program read before +0003)
+0003  00004000 ............
pstore ........ ............  (last program read before +0004)
+0004  00004000 ............
```

Note

It's possible to see a prestore trace list with no "pstore" events. Because each state is examined to see if it meets the store qualification first, prestore memory will never save any states if "prestore" and "store" both have the same qualification (or if prestore has a more specific qualification than store). Make sure your prestore and store qualifications are exclusive.

11

Tracing Processors That Prefetch Instructions And Use An Instruction Pipeline

This chapter discusses the difficulties you encounter when making traces of activity generated by microprocessors that prefetch instructions and use an instruction pipeline. This chapter shows steps you can take to overcome the problems when you are taking a trace, and it shows you how to simplify the task of reading trace lists of processor activity.

What Is Meant By Prefetching And Pipeline?

Figure 11-1 shows the instruction pipeline used in a Motorola 68020 microprocessor. While pipelines vary from one processor to another, figure 11-1 can be used to understand the concept.

A processor is said to be prefetching instructions, when it fetches the instructions before it is ready to execute them. A processor will prefetch instructions from the instruction cache if the instructions are resident there and the instruction cache is active, or it will prefetch the instructions from external memory.

A pipeline is made up of two or more stages that store, decode, and execute instructions simultaneously as they move through the pipeline. This increases the performance of the processor because several instructions can be at different stages of the pipeline at any time, but it makes analysis of processor activity difficult. An analyzer captures states from processor buses, and there may be a delay of several cycles between the time an instruction is fetched (the instruction appears on the bus) and the time it is executed (the resulting operand cycles appear on the bus).

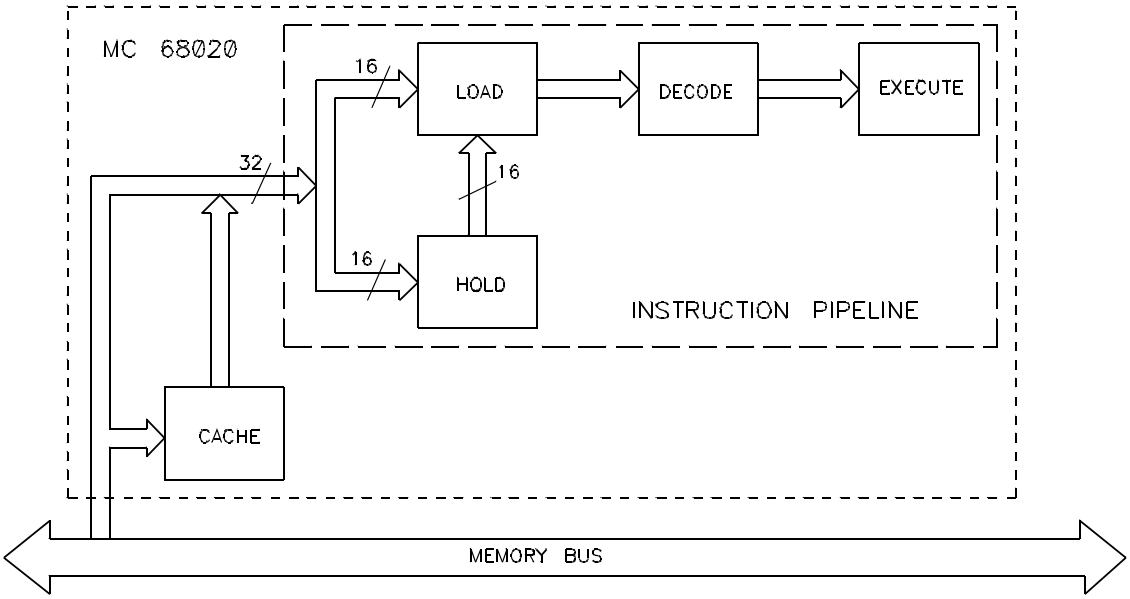

Figure 11-1. Pipeline Diagram Of The 68020

Reading Prefetch/Pipeline Trace Lists

We are used to reading a trace list that shows an instruction followed by the activity resulting from the execution of that instruction. We don't see that order in trace lists made from microprocessors that prefetch instructions and use a pipeline. Instead, we may see an instruction fetch, then activity generated by execution of instructions that were fetched earlier, then prefetches of instructions to be executed in the future, and finally, the execution of the instruction of interest (denoted by its operand cycles).

Figure 11-2 shows the difficulty of reading trace lists made from processors that prefetch instructions and use an instruction pipeline. It was made by tracing activity of a Motorola 68020 microprocessor. While addressing schemes, etc., vary from one processor to another, figure 11-2 does demonstrate the difficulty of

reading the trace lists. Trace memory line number + 0004 contains an instruction. It requires an add to be performed. Notice that the resulting operand cycles of the add instruction are not performed until trace memory line numbers + 0007 and + 0008. The activity on lines + 0005 and + 0006 have nothing to do with the instruction on line + 0004. They are simply instruction prefetches that were pushed into the pipeline after the instruction on line + 0004 was fetched, and before it was executed.

```
Trace List
Label:       Address                 Opcode or Status             time count
Base:        symbols                mnemonic w/symbols            relative
-0002   _move_.+00000000 LINK.W        A6,#$0000                    0.24us
-0001   s:stack+00007F40   $000011E8     supr data long wr (ds32)    0.16us
trigger _move_.+00000004 MOVE.L        A2,-(A7)                     0.20us
       =_move_.+00000006 LEA            ($000051B0,PC),A2
+0001   s:stack+00007F3C   $7FFFFF60     supr data long wr (ds32)    0.28us
+0002   _move_.+00000008   $81700000     supr prgm long rd (ds32)    0.20us
+0003   s:stack+00007F38   $FFFEA194     supr data long wr (ds32)    0.28us
+0004  =_move_.+0000000E ADDQ.L        #1,($8010,A5)                0.36us
+0005  =_move_.+00000012 JSR           ($FE78,PC)                   0.56us
+0006  =_move_.+00000016 MOVE.L        ($****,A6),-(A7)            0.28us
+0007   towers.:move_num   $00000005     supr data long rd (ds32)    0.20us
+0008   towers.:move_num   $00000006     supr data long wr (ds32)    0.24us
+0009   _show_.+00000000 MOVE.L        rm=$3C38,-(A7)               0.28us
+0010   s:stack+00007F34   $00001092     supr data long wr (ds32)    0.20us
+0011   _show_.+00000004 MOVE.L        A5,D0                        0.16us

STATUS:   M68020--Running                Trace complete_____________...R....
display  trace  disassemble_from_line_number -2


  run     trace     set      step     display   modify    end    ---ETC--
```

Figure 11-2. Trace List Showing Pipeline And Prefetch

Unused Prefetches

Sometimes, instructions are prefetched and placed in the pipeline, and never executed at all. For example, this may happen when there is a branch instruction at the end of a function, just before the entry to a new function. When the program nears the end of the function that has the branch instruction, the processor prefetches the entry to the next function (because of its close location in memory). When the branch instruction is executed, the next function is not entered, but instead, execution jumps back to some point specified by the present function. When the processor branches, it flushes the content of its pipeline and begins execution at the "branch-to" address.

The problem the analyzer has with this operation is that it records the fetch of the entry to the new function when it appears on the processor bus. If your trace specification calls for some analyzer response when this entry address appears (trigger, etc.), then the analyzer will react as specified, even though that new function may not be active.

Figure 11-2 also shows an unused prefetch on line + 0006. The "MOVE.L" instruction was prefetched by the processor and placed in the pipeline, but it was never executed. That's because the instruction that was prefetched before it was a JSR. When the JSR was executed, it caused the program to jump to a new address and flush the MOVE.L instruction from the pipeline. (Note that the MOVE.L instruction contains asterisks in the trace list because the end of the instruction was not prefetched before the JSR was executed, and the pipeline was flushed.)

Note

Certain HP analyzers are able to perform trace list dequeuing. Dequeued trace lists do not show unused prefetches. They are much easier to read because they show logical processor activity instead of bus cycle executions.

How To Avoid Triggering, Enabling, Or Disabling On Unused Prefetches

The entry address to the next function is always prefetched at the exit of the function immediately before it in memory (assuming no padding exists between the two functions). To avoid triggering a trace or enabling/disabling a measurement window on an unused prefetch of a function-entry address, you can use a specification such as "trigger_on < functionname> + 6" in your trace command. The "+ 6" specification is enough if you are tracing a Motorola 68020. If tracing a different processor (such as an Intel 80386), you may need to select a different offset. Select a value that ensures that you won't trigger or enable/disable a measurement window on an unused prefetch of the entry to your function. If you use this method for enabling and disabling, you will miss the first few words of the function entry, but these words may only be stack-frame initialization instructions.

The latest versions of Hewlett-Packard "C" Compilers for processors that prefetch instructions have a "debug" option that inserts padding between each of the functions (padding is a series of no-op instructions inserted ahead of each function name). The no-ops are prefetched at the end of the preceding function so the specification of "< functionname> + 6" is not necessary. When using one of these compilers, you can define your specifications to be met on the address of the function entry, without concern that the function-entry address might appear in an unused prefetch.

The exit address may also appear in an unused prefetch. If it does, a measurement window you've specified may be ended prematurely, even though the function on which you enabled is still active. For debugging purposes, you may wish to add some no-op instructions (any instructions that do not affect the functional results) at the end of your function, just preceding the exit instruction. In this way, the no-ops will be prefetched instead of the exit instruction, and you will be able to obtain a full-length enable period in your measurement window.

In an example program used for demonstrating certain Hewlett-Packard emulators and analyzers, there are two places where program instructions named "rts_prefetch = 0" have been added. These no-op instructions are only activated when the example program is compiled on other than a Hewlett-Packard

compiler that offers the debug options. These instructions were placed at the end of the functions to overcome a problem that occurred when the microprocessor prefetched the exit address of the function and caused the analyzer to prematurely end a measurement-enable period. These no-op instructions moved the exit addresses far enough away so that they were no longer prefetched before the nearby branch instructions were executed. Before "rts_prefetch = 0" was used, some activity generated during execution of the function was missed because the measurement window that enabled on that function would disable when the exit address appeared in an unused prefetch.

12

What To Do If Your Emulator Doesn't Work

This chapter discusses things you can try if you are having trouble with your emulator.

Debugging The Connection To The Target System

When your emulator is connected to a target system, emulator operation is complicated by the hardware of the target system and the probe and cable assembly. The following paragraphs will help you find solutions to problems that may occur when an emulator is connected into your target-system hardware.

Hardware Problems

If you need to place a target-system board assembly on an extender board to gain access to the target processor connector, the extender board may cause problems. With the clock rates of some processors exceeding 20 MHz, and with some processors using burst mode, the delay caused by the extender-board hardware may be too great for proper operation of the target system with the emulator. Try to find a way to access the processor socket without using an extender board.

If you have circuit components mounted close to your target microprocessor, you may need to use stacked sockets to allow proper connection of the emulation probe. Stacked sockets can affect the impedance and ground pathways at the processor pins, degrading the performance of your target system.

Electrical Problems

Be careful to design your target system hardware to operate within the timing specifications of the components you are using. Some target systems will operate even when part of their components are operating outside of their timing specifications. In these target systems, the same components may fail to operate once the emulation probe is connected. Timing problems of this sort may appear as improper accesses to emulation memory and target-system memory, as well as failures during DMA cycles.

The emulator may increase the time required to access memory because of the delays associated with sending a signal through the emulation probe cable, and turning on the cable buffers. If you suspect this may be a problem you are having, consider replacing the memory hardware in your target system with faster hardware that can compensate for this delay.

Wire-wrapped prototype hardware in a target system may be difficult to interface with an emulator. This is especially true if that hardware is poorly grounded. The emulation probe uses high-current drivers for its signal lines to maintain the integrity of its signals. These high currents can create voltage pulses within wire-wrapped circuitry; some of these voltage pulses may even exceed the switching thresholds of components in your target system.

Sometimes the problem discussed above can be solved by creation of a damping socket (processor socket with resistors connected in some of the critical signal lines). Usually resistor values between 10 and 100 ohms are used. Twenty-two ohms is a good starting value. In many cases, damping sockets have provided great improvements in the way a target system operates with an emulator.

Architectural Problems

Some target systems fail only in one or two modes when an emulator is connected. These failures may be due to certain target-system circuitry that is only active during these operating modes. Consider the case of a target system that uses a watch-dog timer that must be refreshed periodically or else it will shut down the system. In target systems using circuitry like this, you may solve the problem by simply disabling this kind of circuit during emulation.

Make Sure The Emulation Configuration Is Correct

An incorrect configuration file can cause improper operation. Review the entire configuration file and make sure all of the questions are answered correctly for your target system. If you are not sure how to answer a particular question, refer to your emulator user's guide for details of your configuration file and information about the target system interface. Enter the command "!more < your_configfile_name> .EA" to view the entire configuration file, along with its present selections.

Target systems that can operate with the target processor but not the emulation processor should be able to start with the default configuration file. This is the configuration file that was shipped with your emulator. The default configuration file enables all of the target system signals, maps all memory as target RAM, and specifies that the emulation foreground monitor is not loaded.

Isolate plug-in failures with the default configuration before attempting to use configurations that include emulation memory or an emulation foreground monitor. Once the default configuration works properly, add emulation memory and the foreground monitor, if applicable.

Use The Emulation Bus Analyzer

The emulation bus analyzer can be used with any configuration without interfering with the emulation processor. It passively monitors each bus cycle that the processor executes. All of the analyzer data can be displayed without disrupting the emulation process. The analyzer can be used to verify the proper operation of the program being executed and the proper operation of the hardware.

When debugging hardware failures with the emulation bus analyzer, start with a "trace TRIGGER_ON a= anystate" specification before allowing the processor to run. This will capture all bus cycles starting with the reset address. Pay particular attention to the bus size bit and the data field of the first few cycles. The triggering capability of the analyzer can be used to capture conditions that are the result of a failed interface by using the "trace TRIGGER_ON < failure_condition> " specification. These

conditions are usually incorrect code branches or status conditions such as halt or shutdown.

Failures can sometimes be debugged using the trigger of the emulation bus analyzer to drive external test equipment. Set the emulation bus analyzer to trigger on the bus cycle in error. Use the trigger output with timing analyzers or oscilloscopes to monitor the target system. When observing the data, consider delays. The trigger pulse may actually occur between one and two clock cycles after the end of the bus cycle.

Use Status Messages

The user's manual for your emulator has a complete list of status-line messages and the problems that cause them to appear. Status messages such as "Write to ROM fc= < code> ", "halted" and "slow device fc= < code> " provide address or status information that you can use as a trigger for making a trace with the analyzer.

Run Performance Verification (PV)

The service manual for your HP emulator includes instructions for running performance verification on your emulation system. The PV tests will identify failures in the emulation and analysis hardware, if any.

If All Else Fails

If the emulator is configured properly, and the target program and foreground monitor are loaded, unexplained behavior may still exist. This is frequently due to foreground monitor interaction with the target software and/or hardware.

In the software category, check that it is appropriate to disable interrupts while in the foreground monitor. Some systems with delta-time-interrupt structures for real-time clocks, operating system functions, etc., will crash if the delta-time-interrupt is not serviced within a preset time limit. The foreground monitor can be customized to enable or disable interrupts, as required.

It is possible to "disable" the normal target system function of the non-maskable interrupt through vector table modifications, and a small amount of additional foreground monitor code.

Ensure that the program being executed is not accidentally overwriting the foreground monitor or vise versa.

Obtain a listing of the foreground monitor and the program being executed, and use the analyzer to verify proper operation of both.

Set the analyzer to trigger on the foreground monitor entry address and set the trigger position to the center of the trace. This will allow you to examine CPU activity before and after entry to the foreground monitor. You can observe the stacking activity of the non-maskable interrupt, as well as emulator generated jam cycles. This will enable you to determine if the foreground monitor is being entered properly.

Ensure that the foreground monitor exits and returns to the normal program properly. Set the analyzer to trigger on the foreground monitor exit address, and observe the unstacking process when a return instruction is executed. Be sure that the stack contents are not corrupted, and that the program returns to the expected location.

Notes

Glossary

absolute file This file contains machine-readable instructions and/or data. The instructions and/or data are stored at absolute addresses. Absolute files are generated by the compiler/assembler/linker. These files are loaded into memory for execution by the target processor.

analyzer This is an instrument that captures activity of signals synchronously with a clock signal. An emulation bus analyzer captures emulation bus cycle information. An external analyzer captures activity on signals external to the emulator. Refer to Chapter 8 for a discussion of the types of analyzers.

arm condition A condition that reflects the state of a signal external to the analyzer. The arm condition can be used in branch or storage qualifiers. External signals can be from another analyzer or an instrument connected to the CMB or BNC.

assembler A program that translates symbolic instructions into object code.

background A memory that parallels the emulation processor's normal address range. Entry to background can only take place under system control, and cannot be reached via a user's program.

background monitor A set of fixed, general-purpose routines that do not occupy processor address space. A background monitor is designed to support several of the test and measurement capabilities of an emulator in a wide variety of applications. Refer to the chapter on Monitor Concepts in this manual.

BNC connector A connector that provides a means for the emulator to drive/receive a trigger signal to/from an external device (such as a logic analyzer, oscilloscope, or HP 64000-UX system).

basis branch analysis This is a testing method that determines the extent to which branches are executed during runs of your target program. These tests are performed by HP's Branch Validator.

breakpoint A point at which emulator execution breaks from the target program and begins executing in the monitor. (See also Hardware Breakpoint and Software Breakpoint.)

cache control Cache control is an emulator capability that can turn on and turn off the cache of the emulation processor. You might want to turn on the cache to determine how fast your target system will run when the cache is enabled. You will want to turn off the cache when you are making measurements with an internal analyzer so that the analyzer will have access to all of the processor activity on the emulation bus.

code coverage analysis This form of analysis records every access to every address during a run or your program. It is used to find program portions that have not been tested during exhaustive test procedures.

code segment See executable segment.

command files A file containing a sequence of commands to be executed. Command files are discussed in detail in a chapter of this manual.

compiler A program that translates high-level language source code into object code, or produces an assembly language program with subsequent translation into object code by an assembler. Compilers typically generate a program listing which may list errors displayed during the translation process.

configuration file A file in which configuration information is stored. Typically, configuration files can be modified and reloaded to configure instruments (such as an emulator) or programs (such as the PC Interface).

context A context is a measurement window above selected measurement parameters. A context can be set up to limit the scope of a single measurement parameter, such as enabling trigger recognition to occur only when interrupts are pending. A context can be set up to limit the entire analyzer, such as allowing the analyzer to trace only during execution of a selected code module.

control flow transfer Any change in the normal sequential progress of a program. JMP, CALL, RET, IRET, and INT instructions, as well as exceptions and external interrupts, can cause a change in control flow.

coordinated measurements These are measurements performed by two or more emulators and/or analyzers in which measurement parameters are shared over an interconnecting bus. This is discussed in detail in the chapter titled, "Intermodule Buses (IMB And CMB)."

coprocessor support Some emulators contain an on-board floating-point processor that provides support for custom coprocessors. In those emulators, you can display and modify the content of custom coprocessor registers.

count The count function measures periods of time or numbers of state transactions between states stored in memory. You can set up the analyzer count function to count occurrences of a selected event during the trace, such as counting how many times a variable is read between each of the writes to the variable during a trace. The analyzer can also be set up to count elapsed time, such as counting the time spent executing within a particular function during a run of your target program.

coverage analysis A measure of the percentage of memory accessed by a target program. For each byte of emulation memory, there is one bit of analyzer memory, and it is set when that byte is accessed (written to or read from).

cross trigger The situation in which the trigger condition of one analyzer is used to trigger another analyzer. Signals internal to HP emulation/analysis systems can be used to cross trigger between emulation and external analyzers.

data communications equipment (DCE). A specific RS-232C hardware interface configuration. Typically, DCE is a modem.

data segment A segment that contains data (other than immediate data) for an executable segment. A data segment is identified by a specific type code in the descriptor of the segment.

dequeued instructions A set of instructions and executions presented in the order of execution, which is different from the order captured by the emulation bus analyzer.

downloading The process of transferring absolute files from a host computer into the emulator.

DTE (Data Terminal Equipment) A specific RS-232C hardware interface configuration. Typically, DTE is a terminal or printer.

emulation/analysis system (subsystem) A set of hardware and software capable of performing emulation functions on a target system that uses a particular microprocessor.

emulation bus The emulation bus contains all of the signals on the pins of the emulation microprocessor.

emulation bus analyzer A component in the emulation system that captures the emulation processor's address, data, and status information.

emulation command scanner This is a function in the monitor. After the monitor is invoked and processor information is stored, the emulation command scanner is executed. Here, the emulator continuously tests to see if a new command has been issued. The new command may be to perform one of the monitor routines or to exit the monitor and return to execution of a target program.

emulation data base (EDB) Data base used during the first ten years (approximately) of production of HP's emulation and analysis products. This data base maintained a cross reference between symbols defined in source files and the absolute addresses where the related code was placed in memory. The Emulation Data Base (EDB) has been replaced by the Symbol Retrieval Utility (SRU) in HP's recent emulation and analysis products.

emulation memory This is memory space that resides in your emulator hardware.

emulation memory map The emulation memory map defines the addresses supported by memory hardware during emulation. You set up this map during the emulation-configuration process. In it, you assign hardware memory in your emulator, and/or in your target system, to support ranges of addresses. In this map, you also define the behavior of the memory so that it will act as RAM hardware or ROM hardware to emulate the type of memory you intend to install in your target system when its design is complete.

emulation processor The emulation processor is the processor that replaces the target-system processor during an emulation session. The emulation processor is part of the emulation probe.

emulator A tool that replaces the processor in your target system. The goal of the emulator is to operate just like the processor it replaces. The emulator gives information about the bus cycle operation of the processor and allows control over target system execution. Using the emulator, you may view contents of processor registers, target system memory, and I/O resources.

emulator probe The cable that connects the emulator to the target system microprocessor socket.

entry point An executable-segment offset that identifies the starting point for execution, as when the segment is invoked via a gate.

error message Any message placed on the screen of your emulator terminal to notify you of a problem (or potential problem) with your emulator setup or most recent command.

exception A processor-detected condition that requires software intervention. Many microprocessors communicate exceptions to software by means of the interrupt mechanism.

emulation bus analysis The process of capturing states from the emulation bus, using the emulation processor clock. These states are shown in list form. They show the details of the execution of a target program.

escape sequence (transparent mode) A keyboard input consisting of a special sequence of characters, beginning with the escape character (1C hexadecimal). This sequence is used to access an emulator while in transparent mode. When using multiple emulators and transparent mode to access the different emulators, each one must be given a unique escape character.

external analyzer An analyzer that captures activity on signal nodes external to the emulation processor buses.

external timing analyzers These analyzers capture timing information by recording the logic levels at selected nodes at each occurrence of an analyzer clock. The clock rate must be selected so that it is faster than the data rate to avoid detecting a glitch on the occurrence of a single data bit.

external analyzer probe A set of signal lines that connect the external analyzer to target-system signals.

external clock Any clock other than the clock source of the emulator. Typically, the clock of the target system is used as an external clock for emulation tests and measurements.

flag One of several Booleans maintained by the CPU, including arithmetic flags, control flags, and nested task flag.

foreground The directly addressable memory range of the emulation processor.

foreground monitor Foreground monitors are supplied as source files. They are written in the assembly language of the processors they emulate. You can edit forground monitors to customize them for support of your target system. Before you can use a foreground monitor, you must assemble it and link it and load it in a portion of your emulation memory. Refer to the chapter on monitor concepts in this manual for further discussion of foreground monitors.

fork Temporary diversion from one program or process to start another program or process.

form The part of the PC Interface screen that allows you to enter data for modifying various parameters.

function codes Some emulators support the use of processor function codes. In these emulators, emulation memory can be mapped to any of the functional address spaces (CPU, supervisor or user, program or data, or undefined). Function codes can be used as additional qualifications when referencing memory.

glitch This is the name assigned to the detection of at least one transition in both directions between any two sampling clock pulses during a timing measurement.

guarded Any memory address space that you are not using (whether or not memory hardware is available to support this address space). This identifies address space that should never be accessed by the emulation processor, either for a write or read transaction. Guarded normally identifies non-existent memory space. If the emulation processor ever tries to read from or write to memory space mapped as guarded, the emulator will detect this event and place an error message on the status line of your display. If you are operating your emulator in a mode that allows breaks to the monitor (a mode other than real-time mode) execution will break from the target program and begin executing the monitor program. If you have an internal analyzer, you may be able to trace activity leading up to an attempted access to guarded memory space. (Note that no emulation hardware memory is used to support an address space designated as guarded space.)

handshaking A process of receiving and/or sending control characters that indicate a device is ready to receive data, that data has been sent, and that data has been accepted.

hardware breakpoint Hardware breakpoints are generated by the emulation bus analyzer when it recognizes a specific state, or any state in a range of states, you specified. To achieve a hardware breakpoint, the emulation bus analyzer uses the non-maskable interrupt signal to force the emulator to transfer execution to its monitor program.

host computer A computer to which an HP emulator can be connected. A host computer may run interface programs that control the emulator. Host computers may also be used to develop programs to be downloaded into the emulator.

host processor The host processor (also called emulation-control processor) is the processor that controls the emulator, and controls the emulation processor during an emulation session.

hybrid foreground/background monitors Refers to an emulation design that allows the use of background and foreground monitor programs together.

in-circuit/out-of-circuit emulation In-circuit emulation is emulation performed with the emulation processor connected into the target system hardware. In this mode, you can use and test the resources of the target system, such as target-system memory and target-system I/O ports. Out-of-circuit emulation is emulation performed with the emulation processor disconnected from the target system hardware. In this mode, you must use the resources of your emulator, such as emulation memory and simulated I/O.

index The field of a selector that identifies a slot in a descriptor table.

initialization See hardware/software initialization.

instruction pipeline This is a set of logic hardware designed to store and decode a series of instructions that are waiting to be executed in a microprocessor.

instrumentation card cage The hardware frame and power supply built to accept installation of emulation and analysis board assemblies, and to provide interconnections for boards and interconnections for development stations that control the development hardware.

intermodule bus (IMB) The bus connecting two or more HP emulators/analyzers or connecting an HP 64000-UX emulator/analyzer and an HP IMB/CMB Interface to allow coordinated measurements.

internal analysis (also called emulation bus analysis) An analysis performed by capturing states executed by the emulation processor and appearing on the emulation bus.

interrupt This can have at least two meanings: (1) the electrical or logical signal that indicates an event has occurred, and (2) the mechanism by which a computer system responds quickly to events that occur at unpredictable times.

interrupt handler A routine that is invoked by the occurrence of an interrupt.

inverse assembler A program that translates absolute code captured by an emulation bus analyzer into assembly language mnemonics.

labels A label identifies a data input or set of data inputs to be monitored. You might assign a label called ADDRESS to identify all of the lines connected to the processor address bus. Normally, you will define the logic polarity of the data inputs included in a label, and define the logic thresholds of those data inputs. Not all analyzers allow you to define your own labels.

linker A program that combines relocatable object modules into an absolute file which can be loaded into the emulator and executed.

linker map The linker map identifies code modules that will be loaded into address space that was made available in the emulation memory map.

load memory The name of the activity that places a copy of your target program in the memory available in your emulator or your target system, or both.

locked exit One of two methods used to leave the PC Interface and return to a host computer operating system. A locked exit command allows you to exit the PC Interface and re-enter later with the current configuration. (See also Unlocked Exit.)

logging commands The process of automatically storing each entered command into a file. This file can later be used as a command file. A chapter in this manual discusses the details of command files.

logical address range The range of addresses that store the entire program in a system employing memory management. Logical address range is the same as virtual address range.

logical address space The amount of virtual memory that a microprocessor is capable of accessing. Virtual memory techniques (for example, using pages or segments to address memory consisting of a limited amount of high-speed physical memory and a large capacity memory device such as a disc) can

allow a processor to access more memory than permitted by the size of the address bus.

macros Custom made commands that represent a sequence of other commands. Entire sequences of commands defined in macros will be automatically executed when you enter the macro name. Macro nesting is permitted; this allows a macro definition to contain other macros.

maximum clock speed This is the fastest clock speed that still allows hardware to operate correctly.

memory management unit A memory management unit is used to achieve apparently large addressable memory in a system with limited addressable memory. This is done by placing the target program into a large, usually slow, logical (virtual) memory space. As portions of the program are needed for execution by the target processor, the memory management unit of the processor swaps blocks of code from the slower, logical memory space to the faster, physical memory. The processor fetches its instructions and performs its data writes in physical memory space. When all of the physical memory space is filled and additional code is required for execution, blocks from the existing physical memory space are written back out to the logical memory space where they are stored until they are need again for future executions.

memory mapping This is the process of assigning hardware memory to support addresses occupied by software. For a detailed discussion of this process, refer to the memory mapping concepts chapter in this manual.

memory mapper term A number assigned to a specific address range in the memory map. Term numbers are consecutive.

monitor The monitor is a collection of routines that perform many of the functions needed in an emulator, such as displaying the content of registers or loading code into memory so that the code can be executed during a test. For a detailed discussion of monitors, refer to the monitor concepts chapter in this manual.

monitor program A program executed by the emulation processor that allows the emulation system control to access target system resources. For example, when you enter a command that requires access to target system resources, the system controller writes a command code to a storage area and breaks the execution of the emulation processor from the target program into the monitor. The monitor program then reads the command from the storage area and executes the processor instructions that access the target system. After the target system resources have been accessed, execution returns to the target program.

multiprocessing Using more than one CPU to execute a multi-tasking system.

multi-tasking The capability to support more than one task either simultaneously (by using more than one CPU) or virtually simultaneously (by multiplexing one CPU among several tasks).

non-maskable interrupt (NMI) An external interrupt presented to the NMI pin of a microprocessor. Interrupts presented on the NMI pin cannot be ignored by the microprocessor, regardless of the present conditions of any interrupt masks in force.

object module format (OMF) A standard for the structure of object code files.

offset The address of a location within a segment, expressed as a quantity to be added to the base address of the segment.

operating system Software that controls the execution of computer programs and the flow of data to and from peripheral devices.

out-of-circuit emulation See in-circuit/out-of-circuit emulation.

overlay (used when creating an emulation memory map) This parameter allows you to map two or more address ranges to the same physical memory hardware. The two or more address ranges can even have different hardware characteristics: one can be RAM and another can be ROM, if desired.

parity setting The configuration of parity switches. Depending on the configuration of the parity output switch and the parity switch, a parity check bit is added to the end of data to make the sum of the total bits even or odd. A parity check is performed after data has been transferred, and is accomplished by testing a unit of the data for either odd or even parity to determine whether an error has occurred in reading, writing, or transmitting the data.

path Also referred to as a directory (for example **\users\projects**).

pass through mode See transparent mode.

PC Interface A program that runs on the HP Vectra and IBM PC/AT compatible computers. This is a friendly interface used to operate an HP 64700-series emulator.

performance measurements Performance measurements are taken to help you improve software performance; they involve measurements of program activity and module duration. Activity measurements record information about memory activity and program activity to tell how much execution time is spent and how many states are executed by each function and variable defined in the program under test. Activity measurements are also used to compare these functions and variables with one another to determine their relative use of time and states. Duration measurements record the time spent executing selected functions and procedures. You can use duration measurements to determine the average time it takes to execute a program module, and detect events when the duration of a module is abnormally different from its average duration.

performance verification (PV) A program that tests the emulator to determine whether the emulation and analysis hardware is functioning properly.

physical address In an 80286 processor, a 24-bit address, such as that used as a base address, capable of encompassing the entire address space of the 80286. In a system using memory management, physical addresses are addresses where the target processor will fetch its code and perform its address reads and writes. See the discussion called "memory management unit" in this glossary for further details.

physical address space The amount of real address space that a microprocessor is capable of addressing. A processor's physical address space can be determined by the number of address lines (for example, a microprocessor with 16 address lines has an address space of 2 to the 16th power, or 64K memory locations.

prefetch The ability of a microprocessor to fetch additional opcodes and operands before the current instruction is finished executing.

prestore This is the capture of prestore-qualified states that precede store-qualified states. Prestore measurements are discussed in detail in a chapter of this manual.

privilege The right to access certain portions of memory or to execute certain processor instructions.

privilege level (PL) A measure of privilege. In the 80286 architecture, privilege is measured by integers in the range 0-3, where 0 is the most privileged and 3 the least.

protection A mechanism that limits or prevents access to areas of memory or to instructions.

RAM space Any memory address space assigned to act as Random-Access Memory space during a run of program on your emulator. Emulation RAM space will allow the emulation processor to read from it or write to it.

ranging The term used to describe measurements where any value or address within a range of values or addresses will satisfy the specification. A label is assigned to represent the range of values or addresses (for example, RANGE1 can represent 100H through 1FFH). Then a measurement could be specified to be qualified on any address in RANGE1. If address 104H appeared, it would be qualified because it is within the range of RANGE1. Specifically, it would be shown as RANGE1+ 0004 (assuming you specified that values within RANGE1 would be counted beginning with the lowest value and proceeding to the highest value).

real-time vs. non-real-time emulation You can operate your emulator in one of two modes: real-time mode, or non-real-time mode. In real-time mode, you can run your target program continuously at full rated processor speed, but all of the emulation features that interfere with target program execution are disabled. In non-real-time mode, execution of your target program may be interrupted at any time, but the full feature set of your emulator is always available.

Many emulator features are implemented by routines in the monitor. Whenever the emulator is running a monitor routine, it is (of course) not executing your program in real time. When using non-real-time mode, breaks to the monitor may be initiated by commands you enter or by conditions detected by the emulator. When using real-time mode, features implemented by monitor routines are not available.

real-time execution Refers to the emulator configuration in which commands that temporarily interrupt target program execution (for example, display/modify target memory or processor registers) are not allowed.

Real-time mode capabilities Here is a list of features that are typically available in emulators running in real-time mode:

- run
- some display commands
- some modify commands
- specify
- execute
- trace
- load trace
- stop_trace

Non-real-time mode capabilities All of the emulation features available in real-time mode are also available in non-real-time mode. Additionally, the following emulation features are also available:

- target memory accesses - display, copy, load, modify, and store.
- register accesses - display, copy, and modify.
- software breakpoints - set and reset.

In order to use the emulation features listed above when you have real-time mode in effect, you must manually stop the real-time execution by breaking into the emulation monitor; use a keyboard "break" command, or a software break.

relocation Changing the physical location of a segment.

remote configuration The configuration in which an HP emulator is directly connected to a host computer via a single port. Commands are entered (typically from an interface program running on the host computer) and absolute code is downloaded into the emulator through that single port.

ROM space Any memory address space assigned to act as Read Only Memory space during a run of program in your emulator. Emulation ROM space will allow the emulation processor to read from it, but not write to it. (Note that you can have your emulator load code you've developed into emulation ROM space. This space only acts as ROM hardware when your emulation processor is running your target program.)

RS-232-C A standard serial interface used to connect computers and peripherals.

sequencer An analyzer state machine that searches for execution of states in a particular sequence. In some analyzers, a sequencer runs continuously during a trace (find state A, then state B, then state C, then state D, then start over and find state A, then ....). In some analyzers, a sequencer simply runs until a trigger is found; it is used to enable trigger recognition (find state A, then state B, then state C, then the trigger state). Normally a sequencer includes a restart term (find state A, then state B, restart at state A if state Z appears, then find state C, then find state D).

single step The execution of one microprocessor instruction. Single-stepping the emulator allows you to view program exeution one instruction at a time.

softkey interface The host computer interface program used in the HP 64000-UX environment. The softkey interface is a friendly interface used to control HP 64700 emulators.

software breakpoint Software breakpoints are accomplished by removing an instruction from memory and placing it in temporary storage, and then installing in its place a trap instruction that transfers execution to the monitor. When the software breakpoint is executed, the original instruction replaced in memory is restored by the emulator. Software breakpoints are used to stop target program execution at a particular point so that you can view the state of the processor or target system, or begin program execution from a preselected point in the target program.

Some processors, like the 680x0 family, can process software breakpoints quite easily. Others, like the Z80, need hardware to assist in this process. One limitation of the software breakpoint is that you must be able to replace the original instruction you removed as soon as the software breakpoint is executed. Therefore, you can't set a software breakpoint in target ROM, unless you set up your memory map to overlay the target ROM address with emulation memory.

standalone configuration The configuration in which a data terminal is used to control the HP 64700-series emulator, and the emulator is not connected to a host computer.

stderr An abbreviation for "standard error output". Standard error can be directed to various output devices connected to HP ports.

state/timing analysis Refer to analyzer.

status bus The set of lines that carry information about microprocessor status.

stdin An abbreviation for "standard input". Standard input is typically defined as your computer keyboard.

stdout An abbreviation for "standard output". Standard output can be directed to various output devices connected to HP ports.

store The store function sets up the analyzer to store states of interest in the trace memory. The default of the store specification (any_state) sets up the analyzer to store every state, regardless of its content. When you are interested in analyzing the activity associated with a particular address range, you can store-qualify just that address range, and the analyzer will store only states executed within that address range. If you want to observe a series of writes to a variable, you can store-qualify write transactions to the address of the variable, and the analyzer will store only those write states.

step Refer to single step.

segment A variable-length area of contiguous memory addresses not exceeding 64K bytes.

symbols Symbols are used to represent values. For example, you may have assigned the symbol LOOP to identify the first instruction in one of your code modules. The emulator will show you an address associated with the symbol LOOP. This is the memory address that stores the first byte of code in the instruction beside your symbol LOOP. Symbols can also be used to identify data values and states found on status buses, if desired.

symbol retrieval utility (SRU) Data base used by the most current HP emulation and analysis products. This data base maintains a cross reference between symbols defined in source files and the absolute addresses where the related code is placed in memory. It replaces the Emulation Data Base (EDB) which was used in earlier HP emulation and analysis products.

The SRU allows HP 64000-UX emulation software to read several different file formats. With SRU, an emulator can use HP-MRI IEEE-695 format files and get full symbol support. SRU provides symbol information using the data in the object module format (executable) file.

The IEEE-695 file format provides a sophhisticated view of the executable file and its symbols. This view is more appropriate when dealing with symbols in high level languages (such as C and Ada) than when using assembly language.

The IEEE-695 file format uses symbol relationships that include the concept of modules. Modules (known as packages in the Ada language) reside at the highest level in the symbol hierarchy. For the C language, the IEEE-695 file format will create a module for each C source file. Modules reside at a higher level than source files; a module "owns" the source file and any procedures/globals the source file generates.

This symbol hierarchy can be seen when using IEEE-695 files and accessing and displaying symbols in emulation. A source file is the child of a module symbol. Local symbols are accessed as children of file name symbols or children of modules symbols, depending on the type of local symbol:

- Local symbols that are line numbers are accessed through the file name symbol.
- Local symbols that are not line numbers are accessed through the module name.

The most reliable results will be obtained when using syntax that tells the emulator whether the local symbols reside in the module symbol or in the source file symbol.

synchronous execution The execution of multiple emulator/analyzers at the same time (i.e., multiple emulator start/stop).

syntax The way in which expressions are structured in command languages (the order of entries in a command line). Syntax rules determine which forms of command language syntax are grammmatically acceptable.

target system memory This is memory space that resides within your target system hardware.

target processor The target processor is the microprocessor that controls execution in your target system. You remove the target processor and replace it with the emulation probe to perform an emulation session.

target system The system under development. The intended product of your development efforts.

terminal interface The command interface present inside the HP 64700-series emulators that is used when the emulator is connected to a simple data terminal. This interface provides on-line help, command recall, macros, and other features that provide for easy command entry from a terminal.

timing diagram A timing diagram presents the trace memory in a waveform display. The diagram may appear as a graphics diagram on high or medium resolution monitors, and as an ASCII character diagram on standard terminals.

trace A measurement that collects a series of states. Each state describes the conditions at several points in a system under test. The series of states shows how conditions changed with time at each of the monitored points.

trace list The trace list displays the trace memory contents in list format. The trace memory can be displayed in binary, octal, decimal, and hexadecimal formats, along with a time tag which indicates when the samples were captured.

trace memory This is the memory that stores states captured during a trace. This memory is read by the analyzer when it composes trace lists.

transparency To perform its function properly, the emulator probe must look like the target microprocessor, from the point of view of your target system hardware. The functions, signal quality, signal timing, loading, drive capacity, and any other processor-specific characteristics must be indistinguishable at the plug-in connector. This is called emulator transparency.

- Functional transparency is the ability of the emulator to function the same as the target processor when the emulator is connected to your target system. Total functional transparency means the emulator can execute your program, generate outputs, and respond to inputs in exactly the same way as the target processor.

- Timing transparency refers to the timing relationships between signals on the pins of the emulation probe and signals on the pins of the target processor. The timing relationships of signals at the emulation probe are designed to be equivalent to, or better than, the timing relationships of signals on the target system microprocessor.

- Electrical transparency refers to the electrical characteristics of pins on the emulation probe compared to pins of the actual target processor. These characteristics include rise and fall times, input loading, output drive capacity, and transmission line considerations. The electrical parameters of the emulation probe pins are designed to be equivalent to, or better than those of the target system microprocessor.

transparent configuration The configuration in which the emulator is connected between a data terminal and a host computer. When the emulator is in the transparent (pass through) mode, the data terminal acts like a normal terminal connected to the computer. In this configuration, you can develop code on the host computer and download absolute code into the emulator for debugging and testing.

transparent mode The emulator mode in which all characters received on one port will be copied to the other port. This mode allows a data terminal (connected to one emulator port) to access a host computer (connected to the other emulator port) through the emulator.

trigger A trigger is a point of reference within the trace memory. It specifies when a trace measurement is to be taken. Trigger also refers to the analyzer signal that becomes active when the trigger condition is found.

uploading The transfer of emulation or target system memory content to a host computer.

unlocked exit One of two methods used to leave the PC Interface and return to a host computer operating system. An unlocked exit command allows you to exit the PC Interface and re-enter later with the default emulation configuration. (See also Locked Exit.)

viewport See window.

virtual address An address that consists of a selector and an offset value. The selector chooses a descriptor for a segment; the offset provides an index into the selected segment.

virtual address space The set of all possible virtual addresses that a task can access, as defined by the GDT and the task's LDT. The maximum possible virtual address space for one task is one gigabyte.

virtual memory A style of memory management that permits the virtual address space to exceed the physical address space of RAM. With the help of processor features, the operating system simulates the virtual address space by using secondary storage to hold the overflow from RAM.

wait states Extra microprocessor clock cycles that increase the total time of a bus cycle. Wait states are typically used when slower memory is implemented.

window A specified rectangular area of virtual space shown on the display in which data can be observed. Note that there are also measurement windows. Don't confuse measurement windows with multiple display areas on a computer terminal. A measurement window is a period in state flow when analyzer tracing is enabled. Analyzer tracing is disabled during the portion of state flow that is outside the window.

word count A field of a gate descriptor that specifies the number of words of parameters to be copied from the calling procedure's stack to the stack of the called procedure.

Index