HEWLETT PACKARD

User's Guide

HP 64430
68030 Internal Analysis

HP 64430

68030 Internal Analyzer

User's Guide

Use this manual with the16-Bit and
32-Bit Analysis Reference Manual.

HEWLETT
PACKARD

HP Part No. 64430-97005
Printed in U.S.A.
December 1990

Edition 2

Notice

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.

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

Edition 1	64430-97002, February 1990
Edition 2	**64430-97005, December 1990**

Electromagnetic Interference

What Is Electromagnetic Interference?

All types of electronic equipment are potential sources of unintentional electromagnetic radiation which may cause interference with licensed communication services. Products which utilize digital waveforms such as any computing device are particularly characteristic of this phenomena and use of these products may require that special care be taken to ensure that Electromagnetic Interference (EMI) is controlled. Various government agencies regulate the levels of unintentional spurious radiation which may be generated by electronic equipment. The operator of this product should be familiar with the specific regulatory requirement in effect in his locality.

The HP 64000-UX has been designed and tested to the requirements of the Federal Republic of Germany VDE 0871 Level A. They have been licensed with the German ZZF as Level A products (FTZ C-112/82. These specifications and the laws of many other countries require that if emissions from these products cause harmfull interference with licensed radio communications, that the operator of the interference source may be required to cease operation of the product and correct the situation.

Reducing the Risk Of EMI

1. Ensure that the top cover of the HP 64120A Instrumentation Cardcage is properly installed and that all screws are tight (do not over tighten).

2. When using a feature set which includes cables that egress from the chassis slot of the HP 64120A, insure that the knurled nuts and ferrels, or brackets that ground the cable shields are clean and tight (Do not overtighten). The EEPROM Programmer cable has an exposed shield that must make contact with the cable clamp.

3. During times of infrequent use, disconnect the EEPROM Programmer and cables from the card cage and the target system.

4. Use only shielded coaxial cables on the four external BNC connectors on the rear of the HP 64120A

5. Use only the shielded IMB cable supplied with the HP 64120A for connection to additional HP 64120A Instrumentation Cardcages.

6. Use only shielded cables on the IEEE 488 interface connector to the host computer.

Reducing Interference

In the unlikely event that emissions from the HP 64000-UX System result in electromagnetic interference with other equipment, you may use the following measures to reduce or eliminate the interference.

1. If possible, increase the distance between the emulation system and the susceptible equipment.

2. Rearrange the orientation of the chassis and cables of the emulation system.

3. Plug the HP 64120A into a separate power outlet from the one used by the susceptible equipment (the two outlets should be on different electrical circuits).

4. Plug the HP 64120A into a separate isolation transformer or power line filter.

You may need to contact your local Hewlett-Packard sales office for additional suggestions. Also, the U.S.A. Federal Communications Commission has prepared a booklet entitled *How to Identify and Resolve Radio - TV Interference Problems* which may be helpful to you. This booklet (stock #004-000-00345-4) may be purchased from the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402 U.S.A.

Manufacturer's Declarations

U.S.A. Federal Communications Commission

Warning - This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

Federal Republic of Germany

Wenn Ihr Geraet in der Bundesrepublik Deutschland einschl. West-Berlin betrieben wird, senden Sie bitte beiliegende Postkarte ausgefuellt an Ihr zustaendiges Fernmeldeamt.

Safety

Summary of Safe Procedures

The following general safety precautions must be observed during all phases of operation, service, and repair of this instrument. Failure to comply with these precautions or with specific warnings elsewhere in this manual violates safety standards of design, manufacture, and intended use of the instrument. Hewlett-Packard Company assumes no liability for the customer's failure to comply with these requirements.

Ground The Instrument

To minimize shock hazard, the instrument chassis and cabinet must be connected to an electrical ground. The instrument is equipped with a three-conductor ac power cable. The power cable must either be plugged into an approved three-contact electrical outlet or used with a three-contact to two-contact adapter with the grounding wire (green) firmly connected to an electrical ground (safety ground) at the power outlet. The power jack and mating plug of the power cable meet International Electrotechnical Commission (IEC) safety standards.

Do Not Operate In An Explosive Atmosphere

Do not operate the instrument in the presence of flammable gases or fumes. Operation of any electrical instrument in such an environment constitutes a definite safety hazard.

Keep Away From Live Circuits

Operating personnel must not remove instrument covers. Component replacement and internal adjustments must be made by qualified maintenance personnel. Do not replace components with the power cable connected. Under certain conditions, dangerous voltages may exist even with the power cable removed. To avoid injuries, always disconnect power and discharge circuits before touching them.

Do Not Service Or Adjust Alone

Do not attempt internal service or adjustment unless another person, capable of rendering first aid and resuscitation, is present.

Do Not Substitute Parts Or Modify Instrument

Because of the danger of introducing additional hazards, do not install substitute parts or perform any unauthorized modification of the instrument. Return the instrument to a Hewlett-Packard Sales and Service Office for service and repair to ensure that safety features are maintained.

Dangerous Procedure Warnings

Warnings, such as the example below, precede potentially dangerous procedures throughout this manual. Instructions contained in the warnings must be followed.

Warning

Dangerous voltages, capable of causing death, are present in this instrument. Use extreme caution when handling, testing, and adjusting.

Safety Symbols Used In Manuals

The following is a list of general definitions of safety symbols used on equipment or in manuals:

⚠	Instruction manual symbol: the product is marked with this symbol when it is necessary for the user to refer to the instruction manual in order to protect against damage to the instrument.
↯	Indicates dangerous voltage (terminals fed from the interior by voltage exceeding 1000 volts must be marked with this symbol).
⏚ OR ⏚	Protective conductor terminal. For protection against electrical shock in case of a fault. Used with field wiring terminals to indicate the terminal which must be connected to ground before operating the equipment.
⏚	Low-noise or noiseless, clean ground (earth) terminal. Used for a signal common, as well as providing protection against electrical shock in case of a fault. A terminal marked with this symbol must be connected to ground in the manner described in the installation (operating) manual before operating the equipment.
⊥ OR ⊥	Frame or chassis terminal. A connection to the frame (chassis) of the equipment which normally includes all exposed metal structures.
∿	Alternating current (power line).
⎓	Direct current (power line).
⏦	Alternating or direct current (power line).

Note

The Note sign denotes important information. It calls your attention to a procedure, practice, condition, or similar situation which is essential to highlight.

Caution

The Caution sign denotes a hazard. It calls your attention to an operating procedure, practice, condition, or similar situation, which, if not correctly performed or adhered to, could result in damage to or destruction of part or all of the product.

Warning

The Warning sign denotes a hazard. It calls your attention to a procedure, practice, condition or the like, which, if not correctly performed, could result in injury or death to personnel.

Using This Manual

Organization

This manual is intended to be used with the 16/32-Bit Internal Analyzer Reference Manual, which discusses aspects about this analyzer regardless of where it is installed. The information provided in this manual discusses information that applies only when the internal analyzer is installed in an HP 68030 Emulation/Analysis system.

When you do the examples in the 16-/32-Bit Internal Analyzer Reference Manual, some of the manual display contents may be different from what you see on screen. The 16-/32-Bit Internal Analyzer Reference Manual was written using the 68020 to generate trace lists. Differences between the 68020 and 68030 account for some differences you will see.

Chapter 1

Discusses difficulties you will encounter when tracing activity generated by the 68030 processor because of its prefetching and its use of an instruction pipeline. This chapter shows steps you can take to overcome the difficulties when you are taking a trace, and it shows you how to simplify the task of reading trace lists of processor activity.

Chapter 2

Discusses problems you will encounter when you try to make prestore measurements of the activity generated by the 68030 processor during a program run. This chapter will help you recognize these problems and overcome them when you are making prestore measurements.

Chapter 3

Shows you how to use status values to qualify your trace commands, and how to read status information when it's displayed in your trace lists. An example at the end of this chapter shows you how status information can be used to obtain a trace list of processor activity when the 68030 responds to an interrupt.

Chapter 4 Discusses the way the analyzer solves two problems that are encountered when using 68030 memory management: (1) viewing the logical-to-physical address mappings performed by the MMU, and (2) supplying logical addresses (derived by reverse-translating physical addresses) that the analyzer can use to interpret source-file symbols when performing measurements, and when showing source-file symbols in trace lists. Read this chapter only if you are using the MMU of the 68030.

Chapter 5 Shows an example procedure where the 68030 MMU is set up and the deMMUer is configured to provide analysis in a system using the memory management capabilities of the 68030 processor.

Chapter 6 Shows you how to obtain and read dequeued trace lists. Normal trace list dequeuing is discussed, and examples are shown. Finally, this chapter shows you how to recognize errors in dequeued trace lists, and how you can correct these errors if they occur.

Chapter 7 Contains two sets of new pages: one to describe the copy MMU command syntax, and the other to describe the display MMU command syntax. These new commands support the features that let you view the details of address mappings when you have activated the memory-management feature of the 68030.

This chapter also contains a set of replacement pages for the **display trace** command, whose syntax is shown in Chapter 15 of the 16-/32-Bit Internal Analyzer Reference Manual. This is the only command that is different from the commands shown in the reference manual. The differences in this command syntax are the additional command tokens used to support trace list dequeuing.

Contents

1 **Overcoming Problems With Prefetch And Pipeline**

What Is Meant By Prefetching And Pipeline? 1-1
Reading Prefetch/Pipeline Trace Lists 1-2
  Unused Prefetches 1-3
  Fetching Two Instructions In One Cycle 1-4
  Equivalent Addresses In The Trace List 1-5
Program And Data Fetches In 68030 1-6
  When To Use The "long_aligned" Feature 1-6
  Avoiding Triggering, Enabling, Or Disabling On Unused Prefetches 1-6
  Using "in_procedure" In Place Of "long_aligned" 1-7
Cache Discussion 1-9
  Program Cache 1-9
    Disabling 1-9
    Burst 1-10
  Data Cache 1-10
    Analysis Effects 1-11

2 **Overcoming Problems In Prestore Traces**

What Are Prestore Measurements? 2-1
Module-Call And Variable-Access Common Problem 2-2
Problems When Prestoring Calls To A Program Module 2-4
Prestoring To Find Instructions That Access Variables 2-8

3 **The Status Specification**

Using Status In Trace Commands 3-1
  fcode 3-2
  access 3-4
  addr_mode 3-4
  cycle_type 3-5
  size 3-6
  bus_control 3-7
Reading Analyzer Status In Trace Lists 3-8
  Reading Mnemonic Status Values 3-8

Reading Absolute Status Values 3-16
Example Triggering On Interrupt Status 3-17

4 **Logical And Physical Analysis**

The Mapping-Tables Problem And Its Solution 4-2
The Problem Of Seeing How Addresses Are Mapped 4-2
How To See The Overall Logical-To-Physical Mappings For A Root Pointer 4-2
How To See The Table Details For Mapping A Specific Logical Address 4-4
Interpreting The MMU Tables Display 4-4
Mapping Using Function Codes 4-6
Error Messages Unique To mmu_tables Displays 4-6
Troubleshooting INVALID Mappings 4-8
The Symbols Problem And Its Solution 4-10
The Problem Of Cross Referencing Symbols In Logical Memory To Addresses In Physical Memory 4-10
When Should I Start The DeMMUer? 4-10
Startup With The Emulator 4-10
The Emulator Was Running Without The DeMMUer, And Now I Want To Use It 4-11
How To Turn On And Turn Off The DeMMUer 4-11
Turn On/Off By Using Emulation Configuration Questions 4-11
Turn On/Off By Setting The Analysis Mode 4-12
Description Of The DeMMUer Configuration Display . . . 4-13
How To Obtain Information Needed To Configure The DeMMUer 4-15
How To Access The DeMMUer Configuration Display . . . 4-16
How To Enter (Or Delete) A Range Of Addresses For The DeMMUer To Follow 4-17
How To Enter A New Value For The Translation Control Register 4-18
How To Select A Root Pointer Descriptor_Type 4-20
How To Enter Upper Address Bits In The Virtual Address Start Field 4-21
How Does The DeMMUer Work? 4-22
When Do I Need To Use The DeMMUer? 4-23
When Do I Need To Turn Off The DeMMUer? 4-23
Under What Conditions Will The DeMMUer Fail To Work? 4-24

5 Example Procedure Using DeMMUer In Memory Management

The Initial Conditions 5-1
Access The Emulator 5-2
Turn Off The DeMMUer 5-2
Turn On The 68030 MMU 5-3
Load The Operating System And Get MMU Information 5-3
Set Up The DeMMUer Configuration 5-4
Viewing The Execution Of "os" 5-7
Load The Application Program 5-13

6 Dequeued Trace Lists

How To Turn On And Turn Off The Dequeuer 6-1
When Do I Want A Dequeued Trace List? 6-2
What Does A Dequeued Trace List Show? 6-3
Dequeued Trace List Examples 6-4
  Short Jump/Branch Example .6-4
  Stack Push And Pop Example 6-6
  Operands From 8-Bit Memory Example 6-8
  Exceptions And Interrupts Example 6-10
Problems You May See When Using The Dequeuer 6-12
  Bus Cycles In The Trace List 6-12
  Jumps (JMP and JSR) 6-14
  Branch - Was It Taken Or Not? 6-16
  Correcting Branches ?TAKEN? In Trace List 6-17

7 Analyzer Commands

New Pages 7-1
Replacement Pages 7-1
copy MMU . . . . copy MMU-1
  Syntax . . . . copy MMU-1
  Function . . . . copy MMU-1
  Examples . . . . copy MMU-1
  Parameters . . . . copy MMU-4
display MMU . . . . display MMU-1
  Syntax . . . . display MMU-1
  Function . . . . display MMU-1
  Examples . . . . display MMU-1
  Parameters . . . . display MMU-4
display trace . . . . display trace-1
  Syntax . . . . display trace-1

Function . . . . display trace-1
Examples . . . . display trace-1
Parameters . . . . display trace-4

Index

Illustrations

Figure 1-1. Pipeline Diagram Of The 68030 1-2
Figure 1-2. Trace List Showing Pipeline And Prefetch .1-3

Figure 2-1. Finding Modules Calling show_discs 2-3
Figure 2-2. Error In Which Modules Called show_discs 2-3
Figure 2-3. Setup To Avoid Storing Unused Prefetches 2-5
Figure 2-4. Prestore Module Calls (No Unused Prefetch) 2-5
Figure 2-5. Detecting Who Accessed Variables 2-7
Figure 2-6. Trace List Showing Who Accessed Variables 2-7

Figure 3-1. Command Syntax For s= fcode 3-3
Figure 3-2. Command Syntax For s= access 3-4
Figure 3-3. Command Syntax For s= addr_mode 3-4
Figure 3-4. Command Syntax For s= cycle_type 3-5
Figure 3-5. Command Syntax For s= size 3-6
Figure 3-6. Command Syntax For s= bus_control . . .3-7
Figure 3-7. Composition Of Trace List Status Display .3-9
Figure 3-8. Details Of Trace List Status Displays . . . 3-10
Figure 3-9. Setup To Trigger On An Interrupt 3-17
Figure 3-10. Trace List Showing Trigger On Interrupt 3-18

Figure 4-1. Basic Mappings For A Root Pointer 4-3
Figure 4-2. How Logical Address 0 Is Mapped 4-4
Figure 4-3. Valid MMU Mappings Using Function Codes 4-7

Figure 4-4. Example Of Mapping An Address Using Fcode 4-7
Figure 4-5. Using mmu_tables To Track Invalid Mapping 4-9
Figure 4-6. Address Mapping Details In Table A 4-9
Figure 4-7. DeMMUer Configuration Display 4-13

Figure 5-1. MMU Registers Display 5-4
Figure 5-2. DeMMUer Configuration Setup 5-6
Figure 5-3. Trace List Showing Execution Of "os" . . .5-8
Figure 5-4. Trace List Resynchronized On Line 10 . . .5-9
Figure 5-5. Trace List Resynchronized On Line 20 . . 5-10
Figure 5-6. Trace List Resynchronized On Line 36 . . 5-11
Figure 5-7. Trace List Showing First Table Walk . . . 5-12
Figure 5-8. Trace List Showing Entry To Towers . . . 5-14

Figure 6-1. Short Jump Example, Not_Dequeued . . .6-5
Figure 6-2. Short Jump Example, Dequeued 6-5
Figure 6-3. Stack Push And Pop Example, Not_Dequeued 6-7
Figure 6-4. Stack Push And Pop Example, Dequeued 6-7
Figure 6-5. Operands From 8-Bit Memory, Not_Dequeued 6-9
Figure 6-6. Operands From 8-Bit Memory, Dequeued 6-9
Figure 6-7. Exceptions And Interrupts, Not_Dequeued 6-10
Figure 6-8. Exceptions And Interrupts, Dequeued . . 6-11
Figure 6-9. Problem Of Bus Cycles In Trace List . . . 6-13
Figure 6-10. Example Of Bus Cycles Fixed In Trace List 6-13
Figure 6-11. Example Of Uncertain Jumps In Trace List 6-15
Figure 6-12. Uncertain Jumps Fixed In Trace List . . 6-15
Figure 6-13. Example Branches ?Taken? In Trace List 6-17
Figure 6-14. Not-Dequeued Trace To Solve Branch ?Taken? 6-18

Tables

Table 3-1. Function Code "fcode" Specifications 3-2
Table 3-2. Definition Of Analyzer Status Bits 3-16

Table 4-1. Key To The MMU TABLES Display 4-5
Table 4-2. Definition Of The TC Register Mnemonics 4-14

Table 6-1. Shorthand Notation In Dequeued Trace Lists 6-4
Table 6-2. Notations Following Exceptions Defined 6-11

1

Overcoming Problems With Prefetch And Pipeline

This chapter discusses the difficulties involved with making traces of activity generated by the 68030 processor because of its prefetching, and its use of an instruction pipeline. These features create special problems to overcome when you are trying to take traces and read trace lists of activity generated by the processor. 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.

This chapter also discusses analysis problems encountered when the instruction and data caches are enabled and in use.

What Is Meant By Prefetching And Pipeline? (See figure 1-1.)

The 68030 prefetches instructions from memory (that is, it fetches the instructions before it is ready to execute them). It will prefetch these 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. The instruction stream is fetched in 32-bit long words. The upper 16 bits normally enter the pipeline directly while the lower 16 bits are held in the "hold" register. The hold register is then used as the source for the pipeline. Sixteen-bit words are decoded and executed while they are within the instruction pipeline. The pipelined architecture of the 68030 allows instructions to be loaded, decoded, and executed simultaneously. This increases the performance of the processor, but it makes analysis of processor activity difficult because there is a delay 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).

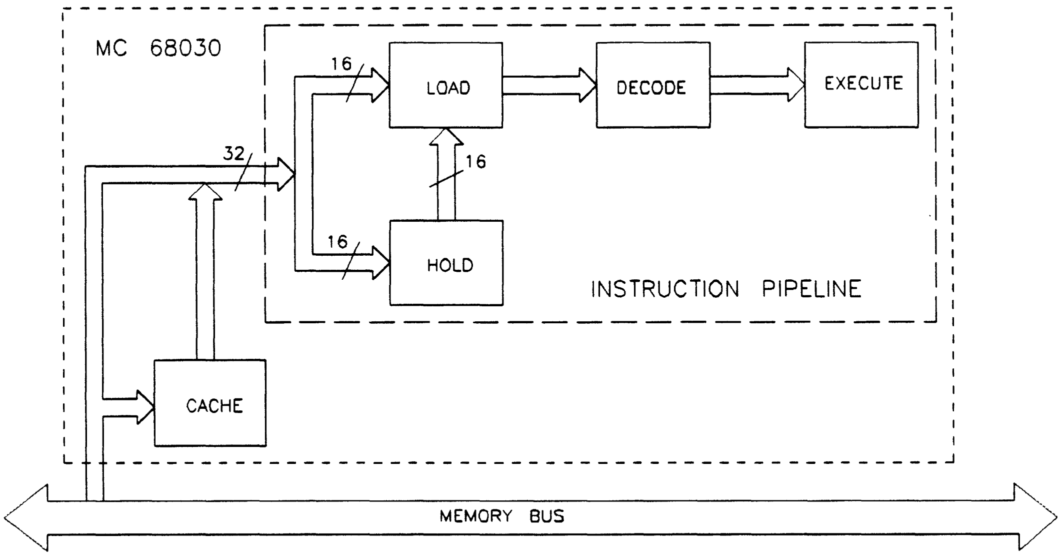


Figure 1-1. Pipeline Diagram Of The 68030

Reading Prefetch/Pipeline Trace Lists (See figure 1-2.)

We are used to reading a trace list that shows an instruction followed by the activity resulting from execution of that instruction. We don't see that order in trace lists made from the 68030. 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 1-2 shows the difficulty of reading trace lists made from processors that prefetch instructions and use an instruction pipeline. 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.

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 because it appeared on the bus. If your trace specification simply says trigger a trace or enable a context when this entry address appears, then your trace

```
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 MOVEM.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:   M68030--Running              Trace complete________________...R....
display  trace  disassemble_from_line_number -2


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

Figure 1-2. Trace List Showing Pipeline And Prefetch

will be triggered or your context will be enabled, even though that new function may not be active.

Figure 1-2 shows an example trace list with 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.)

Fetching Two Instructions In One Cycle

Another performance feature of the 68030 involves the processor's 32-bit data bus and 16-bit instruction format. To maximize data bus performance, the 68030 always fetches 32-bits of information when filling its instruction pipeline (figure 1-1). Because 68030 instructions are 16-bit words, the 68030 may fetch two 1-word instructions in a single bus cycle from a single address.

Equivalent Addresses In The Trace List

To help when reading the address column of a trace list, the inverse assembler emits an equivalent address for each instruction it finds in the low word of a long word. This is done to help you identify where the instruction resides in program memory. Equivalent addresses are addresses that never appeared on the address bus. Equivalent addresses are identified in the trace list by equals signs (=) that precede them. In figure 1-2, trace memory line numbers +0004, +0005, and +0006 all show equivalent addresses in their address column entries. The associated instructions were found in the low words of the 32-bit long words fetched from program memory. You cannot trigger or enable/disable directly on equivalent addresses because they never appear on the emulation bus. In order to trigger or enable/disable on an equivalent address, include the "long_aligned" feature in your trace command.

When a program fetch contains two instructions, the inverse assembler emits two lines for display in the trace list, one for each instruction. Refer to the trigger line in figure 1-2. The address of the second instruction (low word opcode) begins with an equals sign (=xxxxxxxx). This indicates that it is an equivalent address emitted by the inverse assembler to identify the memory location of the associated instruction. Notice that no trace line number or time/state count is shown beside this instruction in the trace list.

Program And Data Fetches In 68030

The addressing scheme that the 68030 uses when it addresses program instructions is different from the one it uses when it addresses data locations. Because of the differences, you'll need to use a different approach when composing a trace specification that is activated on program addresses from the approach you use when composing a trace specification activated on data addresses.

When the 68030 addresses locations in program memory:

1. It always addresses and fetches 32-bit words. These addresses are always multiples of 4, ending in 0h, 4h, 8h, and Ch (except as noted in 3 below).

2. Sixteen-bit instructions may be found in either the high word or low word (or both) of a 32-bit long word.

3. PC-relative fetches from program space will occur at the absolute address of the operand.

When the 68030 addresses locations in data memory:

It always places the absolute address of the data location on the address bus.

When To Use The "long_aligned" Feature

The function of "long_aligned" changes the specified symbolic or numeric address to the corresponding long-word address by setting the lower two address bits to 0. Always use long_aligned with symbols representing addresses in 32-bit program memory to ensure that you are specifying an address that will occur on the bus. If the instruction's address places it in 8-bit or 16-bit memory, you can ignore the long_aligned feature. Never use the long_aligned feature with symbols representing addresses in data memory.

Avoiding Triggering, Enabling, Or Disabling On Unused Prefetches

The entry address to the next function is always prefetched at the exit of the function that resides immediately before it in memory (assuming no padding exits between the two functions). To avoid triggering a trace or enabling/disabling a context on an unused prefetch of a function-entry address, you can use "long_aligned <functionname> +6" in your trace command. This will guarantee that you won't trigger or enable/disable on an unused prefetch of the entry to your function. If you use this method for enabling and disabling, you will miss up to three words of the function entry, but these words are typically only stack-frame initialization instructions.

If you are using the Hewlett-Packard 68030 "C" Compiler and using its debug option, the compiler will insert padding between each of the functions (padding is a series of no-op instructions inserted ahead of each function name). The no-ops will be prefetched at the end of a function so the specification of "<functionname>+6" will not be necessary. When using this compiler, you can define your specifications to be met on the "long_aligned" address of the function entry, without concern that the function-entry address might appear in an unused prefetch.

Using "in_procedure" In Place Of "long_aligned"

The default operation of the "in_procedure" feature does much of the work required to avoid unused prefetches of module entry addresses. Processor-specific corrections are included in the "in_procedure" algorithm to ensure that the analyzer will avoid enabling on unused prefetches of entries to functions.

The debug option of the Hewlett-Packard 68030 "C" Compiler inserts no-op instructions ahead of function entry and function exit addresses to ensure that these entries and exits will not appear in unused prefetches. If you are using this compiler, the prefetch correction provided by the "in_procedure" algorithm is unnecessary, and it should be turned off. Turn it off by entering the following command:

in_procedure_offset none

If not using the above compiler, the use of the "in_procedure" algorithm will avoid the problems of enabling and triggering on unused prefetches of entries to a function, but in_procedure won't help you prevent disabling a context on an unused prefetch of the exit of a function. The exit address may also appear in an unused prefetch. If it does, your context enable period may be ended prematurely, even though the function on which you enabled is still active. For debugging purposes, you may wish to add three no-op instructions (any three 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 context.

In the "towers.c" example program listed in the reference manual for the internal analyzer, there are two places that contain program instructions named "rts_prefetch = 0". These are only activated when the towers.c file is compiled on other than the Hewlett-Packard 68030 "C" Compiler. These are no-op instructions placed at the end of functions in towers.c. They were added to overcome a problem that occurred when the 68030 prefetched the exit address of the associated function and caused the analyzer to prematurely end context-enable periods. These no-op instructions moved the exit addresses far enough away so that they were no longer prefetched before execution of branch instructions. Before "rts_prefetch = 0" was used, some activity generated during execution of the function was missed because the context enabling on that function would disable when the exit address appeared in an unused prefetch.

Refer to the manual for the Hewlett-Packard 68030 "C" Compiler for details of how the debug option affects the compiled code.

Cache Discussion

The 68030 uses two 256-byte caches to store recently used instructions and data. By keeping recently used instructions and data in on-chip caches, the 68030 can access these items if they are used again without having to initiate external bus cycles.

When the 68030 is operating with its caches enabled, analysis of processor activity is limited because the analyzer can only trace activity on the external buses and no bus cycles are performed to fetch instructions or read data in these caches. Therefore, transactions involving the caches will either be incomplete or will not appear at all in the trace lists.

Program Cache

The program cache stores recently used instructions, making them available to be fetched again if they are needed again. When the 68030 fetches an instruction, it checks to see if that instruction is already in the program cache, and if it is, it loads the instruction into its pipeline and does not perform any external bus cycles.

If the analyzer is making a trace at this time, it will miss the cached instructions because they never appear on the buses.

Disabling

Usually, you will want to disable the on-chip caches of the 68030 when tracing processor execution of a program. In this way, all of the instructions executed by the 68030 will be fetched on the processor memory bus and can be captured in the analyzer trace.

There are two ways you can use to disable the on-chip caches. You may prefer one method over the other, depending on your mode of operation.

You may want to disable cache activity during program development so that you can see all of the activity performed by your processor. To completely disable the on-chip caches, invoke the emulation configuration questions by entering **modify configuration**, and proceed to the following questions:

Modify emulator pod configuration? **yes**

Disable on-chip cache? **yes**

To re-enable the on-chip caches, invoke the emulation configuration questions by entering **modify configuration**, and proceed to the following questions:

Modify emulator pod configuration? **yes**

Disable on-chip cache? **no**

Occasionally, you may want to enable the caches to allow your program to execute quickly, except when certain portions of the program are executing. You can go to the memory mapping portion of the configuration questions and specify that caches will be disabled only when executing within certain ranges of target memory. You cannot disable the caches within selected ranges of emulation memory.

Burst

Bursting is the name assigned to the operation performed by the 68030 when it rewrites the content of an entire line (four long words) in the cache. The way the analyzer indicates that a burst has taken place is to make the address column show the address of the long word that was requested by the processor, blank the display in the data column, and show "(burst)" in the status column of the trace list.

You can disable burst operations within selected address ranges in target memory by entering the required specifications in your memory map (emulation memory does not support the bursting feature). You can also turn off the burst capability for the entire program by turning off the caches.

Data Cache

The data cache is the same size as the program cache (16 quad long words). It can be enabled or disabled by the same emulation configuration question that enables or disables the program cache. It can be enabled or disabled for selected areas of target system memory (not emulation memory) by using command options available in the memory map.

Analysis Effects

If the data cache is enabled during a trace, and if you are making a measurement that watches changes in a selected data variable, your trace will show only the first value read, and the new values that are written. All of the reads from that data variable will be performed using the on-chip cache and will not be seen in your trace list because no bus cycles will occur.

Notes

2

Overcoming Problems In Prestore Traces

This chapter discusses problems that will be encountered when you try to make prestore measurements of the activity generated by the 68030 processor during a program run. Prestore measurements are used frequently to capture a series of calls to program modules, and to capture a series of read and/or write accesses to program variables. Calls to modules and accesses to program variables have problems in common, and problems that are unique to each kind of measurement. This chapter will help you recognize these problems and overcome them when you are making prestore measurements.

What Are Prestore Measurements?

Prestore measurements store events of interest in the trace memory, and store transactions related to those events in the prestore memory. A trace list derived from a prestore measurement will show states that are numbered (events of interest), and states that are identified by "pstore" (related events) in the trace memory line number column.

The numbered states are executions that met the STORE qualification. The "pstore" states are events that occurred before the stored states; they failed to meet the STORE qualification, but did meet the "pstore" qualification.

When a state meets the STORE qualification, it is stored in the trace memory, and the present content of the "pstore" register is stored in the prestore memory. When a trace list is prepared at the end of a prestore measurement, each transaction stored in the trace memory is preceded in the list by the last state that was stored in the prestore memory before the store-qualified state was found.

Module-Call And Variable-Access Common Problem

Each "pstore" state is expected to be the function call or read/write instruction that caused the next store-qualified state to occur, but that's not exactly what gets prestored in a 68030 trace measurement. The 68030 performs a prefetch from program memory after it fetches, but before it executes, the function-call or read/write instruction. The prefetched instruction usually meets the prestore qualification and replaces the function-call or read/write instruction in the prestore memory. When entry to the function or access of the variable finally occurs (the store-qualified state), the prestore memory saves the instruction that replaced the function-call or read/write instruction in the prestore memory.

The actual instruction that made the call or accessed the variable will rarely appear in a trace list. Even so, by reading the address of the prestored event, you can identify the code module that was active when the function was called or the read/write was initiated. By taking a new trace of activity using specifications such as:

```
TRIGGER1      a= <address of prestored state>
STORE1        any_state
```

you can quickly find the function-call or variable-access instruction. It will appear in the trace list just before the state that was prestored in the first measurement.

The remainder of this chapter discusses details you need to know to overcome the difficulties involved with prestoring calls to a program module and prestoring to find instructions that access variables.

```
Trace Specification                         A/14   D/16   S/15

GLOBAL CONTEXT    all_activity
TRIGGER POSITION  center_of_trace
PRESTORE          on s= PROGRAM READ

   TRIGGER1     a= long_aligned towers.c:ask_for_number

   STORE1       a= long_aligned towers.c:show_discs

   COUNT1       time




STATUS:   M68030--Running                  Trace complete________________...R....


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

Figure 2-1. Finding Modules Calling show_discs

```
Trace List
Label:              Address                 Opcode or Status       time count
Base:               symbols               mnemonic w/symbols        relative
trigger PROG|/towers.c:_main.+00000090 RTS                            0.56us
       =tower:ask_for_number.+00000000 LINK.W          A6,#$****
pstore  tower:main.whileLoop1+0000001A JSR             (A3)           311.ms
       =tower:main.whileLoop1+0000001C PEA             $****
+0001   |towers.c:show_discs.+00000000 MOVEM.L         rm=$3C38,-(A     0
pstore  towers.:_pause.break4+00000004 MOVE.L          ($75,A0,D4.L   1.12s
+0002   |towers.c:show_discs.+00000000 MOVEM.L         rm=$3C38,-(A     0
pstore =|towers.c:_move_disc.+00000016 MOVE.L          ($****,A6),-   29.6us
+0003   |towers.c:show_discs.+00000000 MOVEM.L         rm=$3C38,-(A   0.72us
pstore  towers.:_pause.break4+00000004 MOVE.L          ($75,A0,D4.L   1.35s
+0004   |towers.c:show_discs.+00000000 MOVEM.L         rm=$3C38,-(A     0
pstore =|towers.c:_move_disc.+00000016 MOVE.L          ($****,A6),-   10.6us
+0005   |towers.c:show_discs.+00000000 MOVEM.L         rm=$3C38,-(A   0.72us
pstore  towers.:_pause.break4+00000004 MOVE.L          ($75,A0,D4.L   2.02s
+0006   |towers.c:show_discs.+00000000 MOVEM.L         rm=$3C38,-(A     0
STATUS:   M68030--Running                  Trace complete________________...R....
 display  trace


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

Figure 2-2. Error In Which Modules Called show_discs

Problems When Prestoring Calls To A Program Module

The prestore capability of the analyzer is useful when you want to obtain a list of the modules that call a subroutine. This list will show each of the calling modules. You can see how often each calling module makes a call, and observe the order of calls made by the calling modules.

Figures 2-1 and 2-2 make a prestore measurement to find the modules that call the show_discs function in the towers.c program. There appear to be at least three modules that call the show_discs function. Actually, one of the calls shown in the trace list (towers.c:_pause.break4) is not really a call at all.

The problem of prefetching has affected the content of the trace list in figure 2-2. The module entry address appeared on the emulation bus in an unused prefetch during the measurement. The analyzer stored the unused prefetch of the module entry address, and prestored the last program read to occur before the unused prefetch. The trace list shows several entries to the module that were actually unused prefetches, preceded by opcodes that weren't even close to module calls. This creates confusion when trying to read a trace list to find out who is calling the module.

The displays shown in figures 2-3 and 2-4 illustrate a way to get around this problem of having your trace list filled with meaningless prestores preceding unused prefetches. Simply change your store specification to ensure that module execution has already begun by storing an address of "long_aligned module_entry+6". Now you won't have to worry about storing unused prefetches of the module entry.

Instead, you'll have to worry about prestoring the first opcode in the called module, and completely missing the calling module. To get around this problem, further qualify your prestore specification to prevent any opcodes from being stored if they are within the address range of the called module. In this way, the prestore memory will contain the last program read from the calling module, and not the first opcode from the called module. You may still want to make a new trace to see which opcode in the calling module actually made the call, but at least you'll know the identity of the calling module.

```
Trace Specification                          A/12    D/16    S/15

GLOBAL CONTEXT    all_activity
TRIGGER POSITION  center_of_trace
PRESTORE          on a= long_aligned not range towers.c:show_discs thru
                  towers.c:show_discs end s= PROGRAM READ

   TRIGGER1      a= long_aligned towers.c:ask_for_number

   STORE1        a= long_aligned towers.c:show_discs+6

   COUNT1        time




STATUS:    M68030--Running                  Trace halted__________________...R....
 display   trace_specification


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

Figure 2-3. Setup To Avoid Storing Unused Prefetches

```
Trace List
Label:              Address                    Opcode or Status        time count
Base:               symbols                  mnemonic w/symbols         relative
trigger PROG|/towers.c:_main.+00000090    $4E754E56      supr prgm        0
pstore  tower:main.whileLoop1+0000001A    $4E934878      supr prgm      1.54s
+0001    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
pstore   |towers.c:_move_disc.+00000014   $FE782F2E      supr prgm      1.12s
+0002    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
pstore   |towers.c:_move_disc.+00000014   $FE782F2E      supr prgm      1.42s
+0003    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
pstore   |towers.c:_move_disc.+00000014   $FE782F2E      supr prgm      1.42s
+0004    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
pstore   |towers.c:_move_disc.+00000014   $FE782F2E      supr prgm      1.69s
+0005    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
pstore   |towers.c:_move_disc.+00000014   $FE782F2E      supr prgm      1.41s
+0006    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
pstore   |towers.c:_move_disc.+00000014   $FE782F2E      supr prgm      1.41s
+0007    |towers.c:show_discs.+00000004   $200D0680      supr prgm        0
STATUS:    M68030--Running                  Trace halted__________________...R....
 display   trace


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

Figure 2-4. Prestore Module Calls (No Unused Prefetch)

Figures 2-3 and 2-4 show an example that corrects the problems of figures 2-1 and 2-2. This time, only valid calls to the show_discs module are stored. The store specification ensures that execution has begun in show_discs. The prestore specification in this measurement uses the "not range" capability of the analyzer to prevent any address within the range of the called function from being stored in the prestore memory. Therefore, the content of the prestore memory is the last opcode prefetched from the range of the calling module.

The above algorithm assumes the first few instructions in the show_discs module are stack-frame initialization instructions, with no looping involved. If the module is a simple function with no parameters, and if it also contains looping constructs, this method will not help.

If you are using the Hewlett-Packard 68030 "C" Compiler with its debug option, the compiler will insert padding in the form of no-op instructions ahead of each function name. This makes most of the preceding precautions unnecessary because only valid calls to the show_discs module will be stored. You will still need to use the "not range" capability in your prestore specification to keep the analyzer from prestoring one of those no-ops.

```
Trace Specification                         A/13   D/16   S/15

GLOBAL CONTEXT    all_activity
TRIGGER POSITION  center_of_trace
PRESTORE          on s= PROGRAM READ

   TRIGGER1     a= long_aligned towers.c:ask_for_number

   STORE1       a= range towers.c:free_level thru towers.c:free_level+3*4-1

   COUNT1       time




STATUS:   M68030--Running                  Trace complete________________...R....
 display  trace_specification


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

Figure 2-5. Detecting Who Accessed Variables

```
Trace List
Label:              Address                              Opcode or Status
Base:               symbols                            mnemonic w/symbols
pstore  towers.:init_display.continue8 ADDQ.L       #1,D3
       =towers.c:init_display.forTest6 MOVEQ        #$00000003,D0
+0003   |towers.c:_free_level+00000008   $00000003    supr data long wr (ds32)
pstore  init_di.functionExit8+00000002 CMPI.B       #$5E,A0
+0004   DATA|m6802/towers.c:free_level   $00000000    supr data long wr (ds32)
pstore  towe:remove_disc.functionExit5 MOVE.L       (A7)+,D3
       =_remove.functionExit5+00000002 MOVEA.L      (A7)+,A2
+0005   DATA|m6802/towers.c:free_level   $00000000    supr data long wr (ds32)
pstore  towers.c:_place_disc.+00000028 MOVE.W       (A0),-(A6)
       =towers.c:_place_disc.+0000002A LSL.L        #4,D0
+0006   |towers.c:_free_level+00000004   $00000003    supr data long rd (ds32)
pstore  towers.c:_place_disc.+00000044 MOVE.B       -(A0),-(A6)
       =towers.c:_place_disc.+00000046 OR.B         -(A0),D0
+0007   |towers.c:_free_level+00000004   $00000003    supr data long rd (ds32)
pstore  tower:place_disc.functionExit6 MOVE.L       (A7)+,D3
STATUS:   M68030--Running                  Trace complete________________...R....
 display  trace


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

Figure 2-6. Trace List Showing Who Accessed Variables

Prestoring To Find Instructions That Access Variables

A typical use for the "prestore" capability is to find out which functions are accessing variables. You may have a variable that is getting wrong information written into it during a program run. If you have several places in your code that write to this variable, you'd like to see which one of these places is writing the bad data. A prestore measurement can quickly give you a trace list showing the accesses to the variable and the prestored write instructions.

Figures 2-5 and 2-6 show a trace specification and trace list used to prestore a series of reads and writes to a range of variables. The prestore measurement was made using the towers.c program which appears in the internal analyzer reference manual.

In figure 2-5, the trace specification was set up to store all accesses in the range of the free_level array. No steps were taken to avoid capturing unused prefetches of the array variables because these are data locations. Unused prefetches are only a problem in program memory.

Figure 2-6 shows the desired reads and writes to the variables. The prestore memory failed to retain the actual read and write instructions that caused the variables to be accessed. The prestore memory was affected by processor prefetching. Each prestored state was one or two instructions after the read or write instruction that caused the variable to be accessed, but the prestored state did identify the area of program that accessed the variable. The actual read or write instructions can easily be seen in a trace list by making a new "STORE_ON any_state" trace, if desired, as was explained earlier in this chapter.

The display width of figure 2-6 was not great enough to include the time count column because a very wide space was allocated to display of the address information. To observe the time count column in such a situation, you can use the Control-f and Control-g (^ f and ^ g) keyboard keys.

The Status Specification

This chapter shows you how to use status values in your trace commands, and how to read status information in your trace lists. An example at the end of this chapter shows you how status information can be used to obtain a trace list showing how the 68030 processes an interrupt.

Using Status In Trace Commands

The entries available for specifying measurement parameters on status conditions are specific to the 68030 processor. These specific 68030 status conditions are discussed in the following paragraphs.

You can qualify measurement parameters on states found on any (or all) of the processor status buses and status bits. You can also qualify measurement parameters on states found on status bits generated by the emulator. The following pages show details of the status qualifications available. The status bits are listed in table 3-2, later in this chapter.

If you know which status specifications you want to identify, you can type in their names without identifying the name of the associated bus or group where they reside. This will save space in your command line. The individual specification will be recognized by the analyzer whether or not you include the name of the status bus or group where the identifier resides. Example:

trace TRIGGER_ON s= fcode PROGRAM access READ

is the same as

trace TRIGGER_ON s= PROGRAM READ

The following paragraphs show you how the analyzer can process your status specifications. Specifications for the status states can be entered in any order on the command line.

fcode

When you enter a specification involving **fcode**, your specification must be met by the three bits of the processor function code. Figure 3-1 shows the command syntax for entry of a function code specification. Refer to table 3-1 for definitions of the function codes available.

Table 3-1. Function Code "fcode" Specifications

IDENTIFIER	FC2-FC0	CYCLE TYPE
SPACE0	0	Undefined, Reserved
USER_DATA	1	User data space
USER_PROG	2	User program space
SPACE3	3	Undefined, Reserved
SPACE4	4	Undefined, Reserved
SUPER_DATA	5	Supervisor data space
SUPER_PROG	6	Supervisor program space
CPU_SPACE	7	CPU space
DATA	1 or 5	Either user or supervisor data space
PROGRAM	2 or 6	Either user or supervisor program space

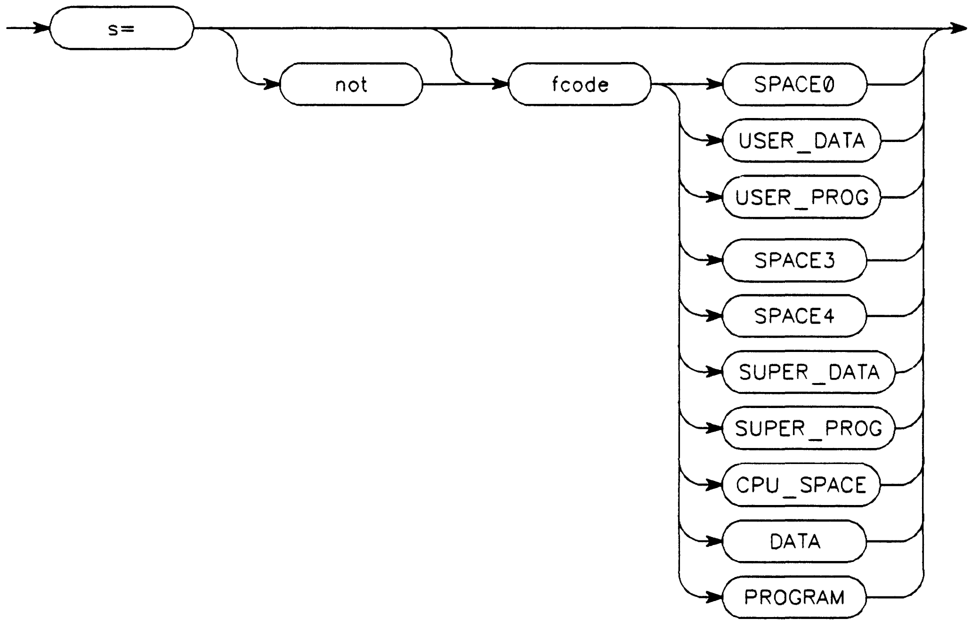


Figure 3-1. Command Syntax For s= fcode

access Figure 3-2 shows the command syntax used to enter a specification involving the R/W and RMC bits of the 68030 processor.

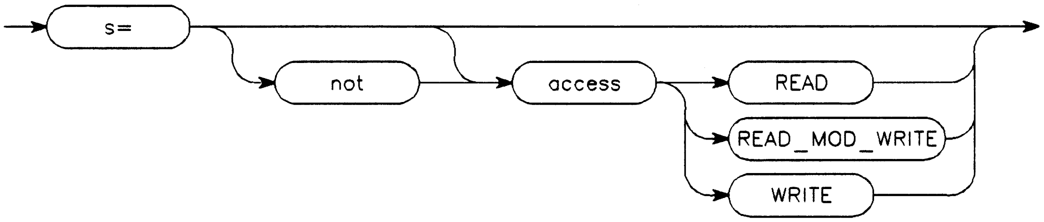


Figure 3-2. Command Syntax For s= access

addr_mode Figure 3-3 shows the command syntax used to enter a specification involving the physical/logical signal generated by the emulator.

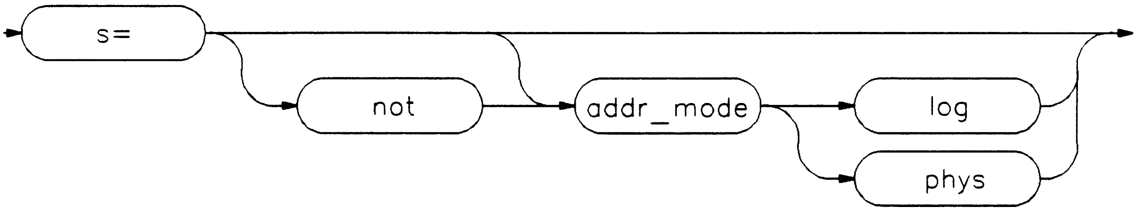


Figure 3-3. Command Syntax For s= addr_mode

cycle_type Figure 3-4 shows the command syntax used to enter a specification involving the BURST signal generated by the emulator, and the STERM, DSACK0, and DSACK1 status bits of the 68030 processor.

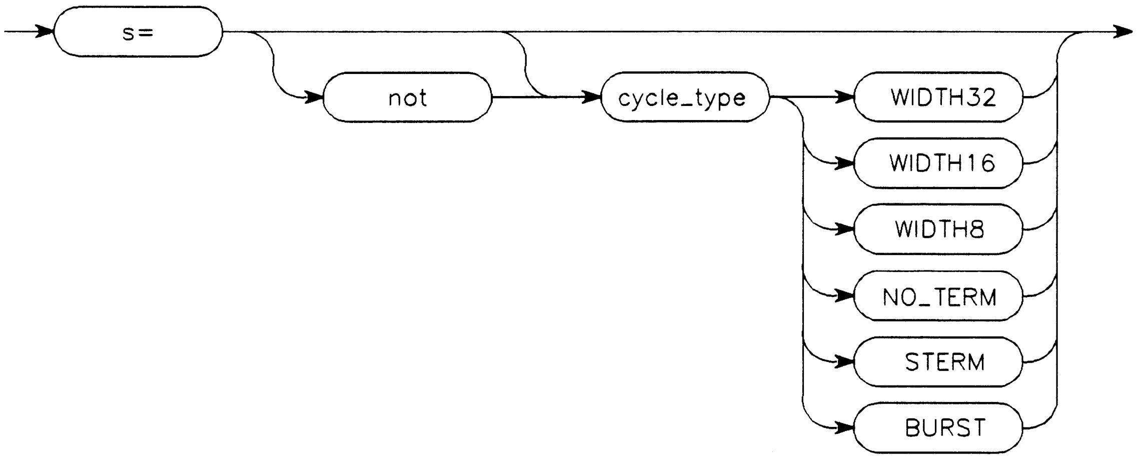


Figure 3-4. Command Syntax For s= cycle_type

size Figure 3-5 shows the command syntax used to enter a specification involving the transfer size bits SIZ0 and SIZ1 of the 68030 processor.

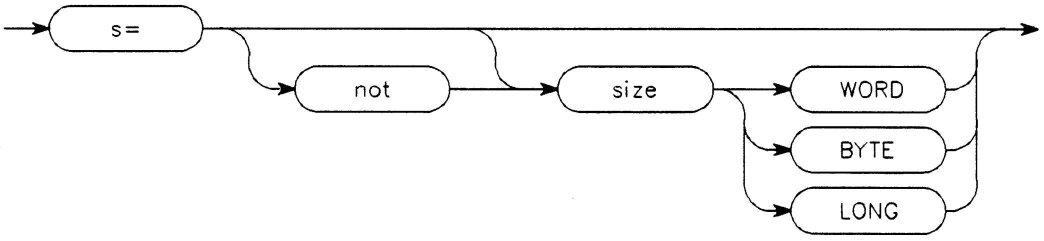


Figure 3-5. Command Syntax For s= size

bus_control

Figure 3-6 shows the command syntax used to enter a specification involving the bus-control bits TABLESEARCH, RETRY, BERR, and DMA which are emulator-generated signals. The DMA transactions are detected by the 68030 emulator. This status specification allows ANDing several selections because the bus_control selections represent individual status bits.

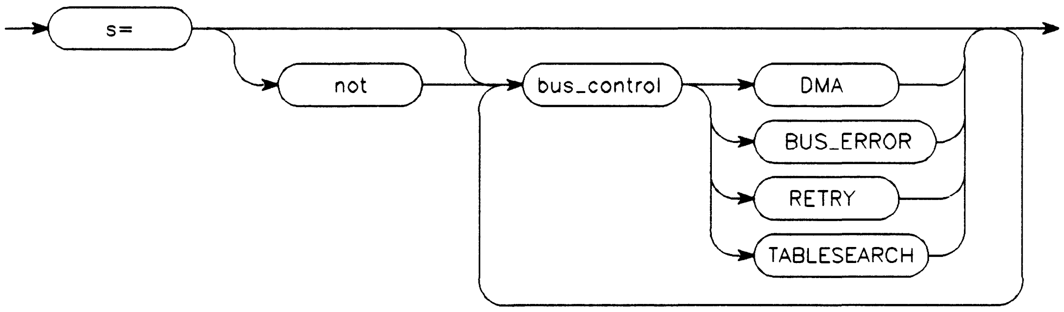


Figure 3-6. Command Syntax For s= bus_control

Reading Analyzer Status In Trace Lists

The following paragraphs show you how to read and interpret processor status in absolute hexadecimal values, and in the mnemonic display forms presented in the trace lists of the 68030 internal analyzer.

Reading Mnemonic Status Values

Figures 3-7 and 3-8 will help you read the content of the trace list that shows status information in mnemonic form. The following lines show examples of mnemonic status information:

```
user data long log wr (ds32)
user prgm long phys rd (ds32)
```

The above statements are composed of the notations available in the internal analyzer, as outlined in figure 3-7. The interpretation of each of these notations is shown in figure 3-8. The two example lines above were composed from notations in blocks A, C, E, and G.

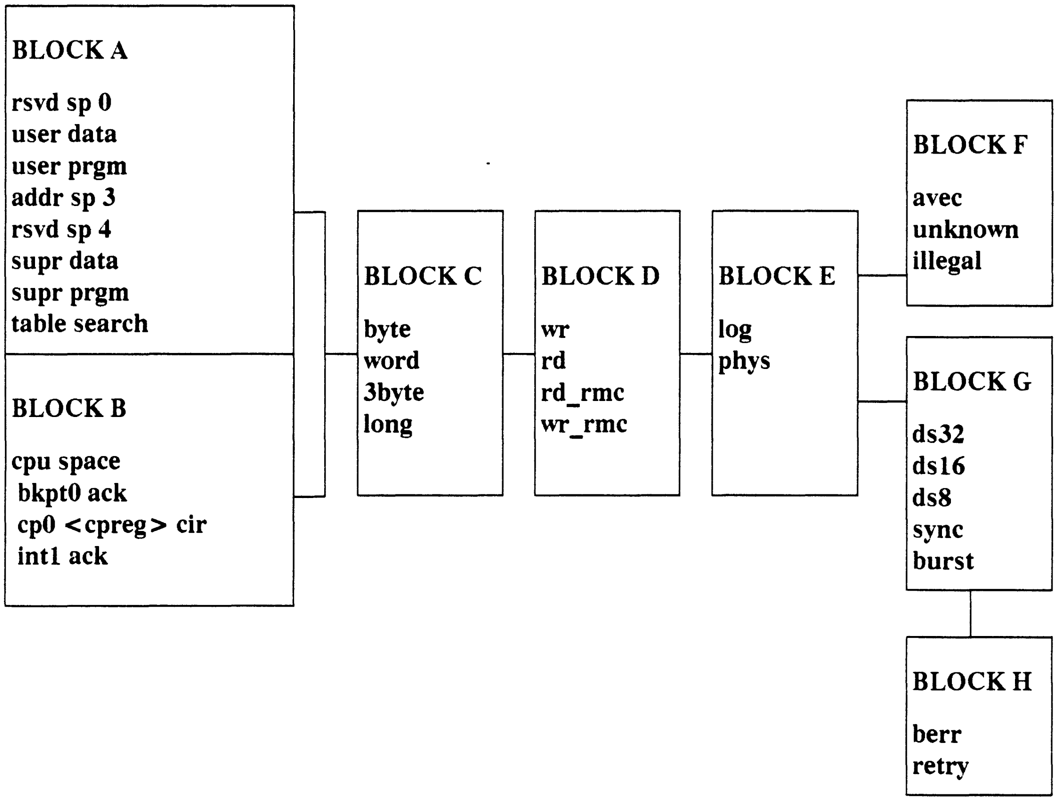


Figure 3-7. Composition Of Trace List Status Display

BLOCK A:

This block shows the results of function-code decoding, according to the following table:

$\overline{\text{TABLESEARCH}}$	FC2	FC1	FC0	Inverse Assembler Display
1	0	0	0	rsvd sp 0
1	0	0	1	user data
1	0	1	0	user prog
1	0	1	1	addr sp 3
1	1	0	0	rsvd sp 4
1	1	0	1	supr data
1	1	1	0	supr prog
1	1	1	1	cpu space (see also block B)
0	X	X	X	table search

BLOCK B:

These displays result from decoding address information during CPU space cycles (FC2-0 = 111B). If none of following are true, "cpu space <NO.>" appears (where "<NO.>" is determined by CPU space address bits A19 thru A16).

"bkpt0 ack"

Displayed when CPU address bits A19-A16 = 0000B and CPU function codes FC2-FC0 = 111B. The breakpoint number acknowledged (0-7) is decoded from CPU address bits A4-A2.

Figure 3-8. Details Of Trace List Status Displays

BLOCK B: (cont'd)

"**cpX <cpreg> cir**" (X can be 0 thru 7)

Displayed when CPU address bits A19-A16 = 0010B, and CPU function codes FC2-FC0 = 111B. This cycle corresponds to an access to a "coprocessor interface register" (cir). The ID of the coprocessor being accessed (0-7) is decoded from address bits 15-13. The "<cpreg>" field is decoded from CPU address bits A4-A0 according to the following table:

A4-A0	Inverse Assembler Display	Coprocessor Register
0000x	cpX response cir	Response
0001x	cpX control cir	Control
0010x	cpX save cir	Save
0011x	cpX restore cir	Restore
0100x	cpX op word cir	Operation Word
0101x	cpX command cir	Command
1001x	cpX rsvd0 cir	(Reserved)
0111x	cpX condition cir	Condition
100xx	cpX operand cir	Operand
1010x	cpX reg sel cir	Register Select
1011x	cpX rsvd 1 cir	(Reserved)
110xx	cpX inst addr cir	Instruction Address
111xx	cpX op addr cir	Operand Address

"**int0 ack**"

Displayed when CPU address bits A19-A16 = 1111B and CPU function codes FC2-FC0 = 111B. The interrupt number being acknowledged (0-7) is decoded from CPU address bits A3-A1.

Figure 3-8. Details Of Trace List Status Displays (Cont'd)

BLOCK C:

This block shows the results of decoding the CPU SIZ1 and SIZ0 signals according to the following table:

SIZ1	SIZ0	Inverse Assembler Display
0	1	byte
1	0	word
1	1	3byte
0	0	long

BLOCK D:

This block shows the results of decoding the CPU R/$\overline{W}$ and $\overline{RMC}$ signals according to the following table:

$\overline{RMC}$	R/$\overline{W}$	Inverse Assembler Display
1	0	wr
1	1	rd
0	0	wr_rmc
0	1	rd_rmc

Figure 3-8. Details Of Trace List Status Displays (Cont'd)

BLOCK E

This block shows the results of decoding the emulator-generated physical/logical status bit according to the following table:

physical/logical	Inverse Assembler Display
1	phys
0	log

BLOCK F:

One of two messages is output by this block when DSACK1, DSACK0, STERM, and BERR CPU signals and the BURST and retry signals from the emulator are all 1.

"**avec**"

Autovector is displayed if these signals are observed all high for an interrupt-acknowledge cycle, indicating that a low value on the CPU AVEC input was the probable cause for the termination of this bus cycle.

"**unknown**"

Displayed if these signals are observed all high for any cycle other than an interrupt acknowledge cycle. This indicates that the inverse assembler was unable to determine the cause for termination of the bus cycle, and usually is the result of the DSACK1 and DSACK0 signals going high prior to the low-to-high transition of the CPU AS signal. (This is usually a violation of 68030 CPU and/or emulator electrical specification #28.) This message indicates you may have a hardware timing problem. Contact the HP Sales/Service Office.

Figure 3-8. Details Of Trace List Status Displays (Cont'd)

BLOCK G:

This block shows the results of decoding the CPU $\overline{\text{STERM}}$, $\overline{\text{DSACK1}}$, and $\overline{\text{DSACK0}}$, signals and the emulator-generated $\overline{\text{BURST}}$ signal according to the following table:

$\overline{\text{BURST}}$	$\overline{\text{STERM}}$	$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$	Inverse Assembler Display
1	1	1	1	(see blocks F and H)
1	1	1	0	ds8
1	1	0	1	ds16
1	1	0	0	ds31
1	0	1	1	sync
0	0	1	1	burst
x	0	?	?	illegal (if ?? is 00, 01, or 10)

When "illegal" appears, more than one of the following signals was asserted at the time the bus cycle was terminated: $\overline{\text{DSACK0}}$, $\overline{\text{DSACK1}}$, $\overline{\text{STERM}}$. This is usually a violation of 68030 CPU and/or emulator electrical specification #28, #60, or #61. This message indicates you may have a hardware timing problem. Contact the HP Sales/Service Office.

BLOCK H:

This block shows the results of decoding the emulator-generated $\overline{\text{BERR}}$ and $\overline{\text{RETRY}}$ signals, according to the following table:

$\overline{\text{BERR}}$	$\overline{\text{RETRY}}$	Inverse Assembler Display
1	0	retry
0	1	berr
1	1	(see blocks F and G)

Figure 3-8. Details Of Trace List Status Displays (Cont'd)

In addition to the above status decoding, displayed values for a particular data cycle are also shown differently, depending on the size of the data bus, as indicated by the DSACK lines. Although shown graphically by the display, the data bus width is also indicated by the presence of "ds32", "ds16" or "ds8." Note the transfer size is independent of the data bus size.

An STERM cycle transfers 32 bits of data (one long word).

During burst cycles, the 32-bit address of the long word requested by the processor is shown in the address column (addresses of the other three long words are not shown). No data is shown with the burst address.

Examples:

data	**FC2-FC0 (function codes)**	**SIZ1-SIZ0 (transfer size)**	**R/$\overline{\text{W}}$ (read/write)**	**$\overline{\text{DSACK1}}$-$\overline{\text{DSACK0}}$, $\overline{\text{STERM}}$, $\overline{\text{BURST}}$ BERR, RETRY**
01	supr data	long	rd	(ds8)
0123	supr data	long	rd	(ds16)
01234567	supr data	long	rd	(ds32)
01xx	supr data	byte	rd	(ds16)
xx23	supr data	word	rd	(ds16)
01xxxxxx	supr data	byte	rd	(ds32)
xx23xxxx	supr data	byte	rd	(ds32)
xxxx45xx	supr data	byte	rd	(ds32)
xxxxxx67	supr data	byte	rd	(strm)
	supr data	long	rd	(brst)

Figure 3-8. Details Of Trace List Status Displays (Cont'd)

Reading Absolute Status Values

When the "display trace absolute" option is selected within the trace list, four-digit, hexadecimal values replace disassembled information. These values can be used in conjunction with normal disassembly to learn more about a particular cycle. Table 3-2 identifies the meaning of each bit in the status values. Figure 3-9 will help you interpret the meanings of status displays in absolute, hexadecimal values.

Table 3-2. Definition Of Analyzer Status Bits

BIT	Definitions
0	**TABLESEARCH** signal generated by the emulator. Indicates a table search cycle.
1	physical/logical signal generated by the emulator.
2	**RETRY** signal generated by the emulator.
3	**BURST** signal generated by the emulator. Indicates burst cycle.
4	SIZ0 signal from 68030.
5	SIZ1 signal from 68030.
6	**BERR** (LBERR) signal generated by the emulator.
7	HCPDMA signal generated by the emulator. When this status bit is high, a DMA cycle has occurred. More precisely, this bit is high between the high-to-low transition of LBG and the low-to-high transition of LBGACK, indicating cycles where a device other than the 68030 CPU is the bus master. Note also that if address strobes occur between the two signal transitions mentioned above, no cycles with this status will appear in the tracelist.
8	FC0 signal from 68030.
9	FC1 signal from 68030.
10	FC2 signal from 68030.
11	**DSACK0** signal from 68030.
12	**DSACK1** signal from 68030.
13	**R/W** signal from 68030.
14	**RMC** signal from 68030.
15	**STERM** signal from 68030.

Example Triggering On Interrupt Status

Figures 3-9 and 3-10 show an example of how the status information can be used in a specification to trace a processor interrupt. The status of the 68030 processor is always "CPU_SPACE" when an interrupt is recognized. The address bus always carries all "f's", except for the least-significant hexadecimal digit. Its value will indicate the level of the interrupt.

```
Trace Specification                           A/15    D/16    S/15

GLOBAL CONTEXT     all_activity
TRIGGER POSITION   center_of_trace

   TRIGGER1     a= 0fffffffxh s= CPU_SPACE

   STORE1       any_state

   COUNT1       time




STATUS:   M68030--Running in monitor      Trace complete________________...R....
 display  trace_specification

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

Figure 3-9. Setup To Trigger On An Interrupt

```
Trace List                                     Mode:logical data
Label:   Address                  Opcode or Status                          time count
Base:      hex                        mnemonic                                relative
-0007    0000010A   $0000         supr prgm word rd log addr (ds16)          0.40us
-0006    00000100   $7000         supr prgm long rd log addr (ds16)          0.60us
-0005    00000102   $0640         supr prgm word rd log addr (ds16)          0.40us
-0004    00000104   $0001         supr prgm long rd log addr (ds16)          0.40us
-0003    00000106   $4EFA         supr prgm word rd log addr (ds16)          0.36us
-0002    00000108   $FFFA         supr prgm long rd log addr (ds16)          0.44us
-0001    0000010A   $0000         supr prgm word rd log addr (ds16)          0.36us
trigger  FFFFFFFF   $             int7 ack byte rd log addr (avec)           0.44us
+0001    0000CF7C   $2000----     supr data word wr log addr (ds32)          0.92us
+0002    0000007C   $0000C100     supr data long rd log addr (ds32)          0.36us
+0003    0000CF7E   $----0000     supr data long wr log addr (ds32)          0.36us
+0004    0000CF80   $0106----     supr data word wr log addr (ds32)          0.40us
+0005    0000C100   $4AF90000     supr prgm long rd log addr (ds32)          0.36us
+0006    0000CF82   $----007C     supr data word wr log addr (ds32)          0.48us
+0007    0000C104   $CD8F6A02     supr prgm long rd log addr (ds32)          0.36us
STATUS:    M68030--Running in monitor      Trace complete________________........
 display  trace

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

Figure 3-10. Trace List Showing Trigger On Interrupt

4

Logical And Physical Analysis

Read this chapter only if you are using the on-chip Memory Management Unit (MMU) of the 68030. If the 68030 MMU is not enabled, you won't need the information in this chapter.

Two problems arise when you use the MMU of the 68030:

1. The processor sets up tables that it uses to map addresses in logical memory to addresses in physical memory. This mapping process can be difficult to use because you can't see the details of these tables and how they manage their address mappings.

2. The analyzer cross references the symbols you used in your source files to addresses in logical (virtual) memory that contain the corresponding code. The analyzer is unable to cross reference symbols in your source-file to addresses in physical memory. Therefore, physical address values can only be expressed in numerics when they are used in analyzer specifications and shown in trace lists.

Both of the above problems are solved by analyzer features that are discussed in this chapter. The solution to seeing how logical addresses are mapped through the tables to physical addresses is discussed first in this chapter. The remaining portion of this chapter is devoted to discussing the solution for cross referencing source-file symbols to physical addresses.

The Mapping-Tables Problem And Its Solution

The following paragraphs discuss the features that let you see how the 68030 uses tables to map addresses in logical memory to addresses in physical memory. You can see a list of all present address mappings. These paragraphs also show you how to see the detailed mapping structure for any logical address you choose.

Note

The **mmu_mappings** and **mmu_tables** features discussed below only work when you are using the background monitor of your emulator. The memory accesses that are required to support these features are not implemented in the foreground monitor.

The Problem Of Seeing How Addresses Are Mapped

When the 68030 is using its MMU to map memory through table searches, there are two kinds of information you might need to see:

1. The overall logical-to-physical address mappings under a particular root pointer (figure 4-1).

2. The details of the mapping tables used to map a specific logical address (figure 4-2).

Each of the above topics of information is discussed in the following paragraphs.

How To See The Overall Logical-To-Physical Mappings For A Root Pointer

The host program can read the address mappings and provide a display like the one shown in figure 4-1. To obtain a list of address mappings for a particular root pointer, enter a command such as:

display mmu_mappings root_ptr CRP

You can select **mmu_mappings** for the CRP, SRP, or any root pointer value of your choice (e.g., **root_ptr 080000002000f4000h**).

```
VALID M68030 MMU MAPPINGS:
    LOGICAL ADDRESS        PHYSICAL ADDRESS
    Lower      Upper       Lower      Upper
         0       FFFF    00010000   0001FFFF
     10000      1FFFF    00020000   0002FFFF
     20000      2FFFF    00030000   0003FFFF
     91000      91FFF    000A0000   000A0FFF
     94000      94FFF    000D0000   000D0FFF
     F0000      FFFFF    000F0000   000FFFFF
  7FFF0000   7FFFFFFF    00040000   0004FFFF
  FFE00000   FFE0FFFF    00068000   00077FFF
  FFE11000   FFE11FFF    00070000   00070FFF
  FFFE0000   FFFEFFFF    00050000   0005FFFF
  FFFF0000   FFFFFFFF    00060000   0006FFFF




STATUS:    M68030--Running in monitor      Trace complete________________........
 display   mmu_mappings   root_ptr   CRP


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

Figure 4-1. Basic Mappings For A Root Pointer

Note

If you enter a root pointer value instead of specifying either the CRP or SRP, you'll also need to specify the value of the TC register. Refer to the detailed explanation of the TC register later in this chapter.

display mmu_mappings root_ptr 080000002000f4000h translation_control 8c0c440h

Your display will show one line of information for each mapped block of addresses. If you are using a small page size such as 256 bytes per page, your display may be several screens long. In this case, you may want to view just a portion of the mmu_mappings list. You can begin the list at any desired logical address, ignoring all of the lower (preceding) addresses (e.g., **show_map_from logical_address 08000000h**). You can also scroll the display window using the up/down arrow keys of your keyboard.

```
M68030 MMU TABLES:
              LOGICAL ADDR(hex)      0    0    0    0    0    0    0    0
                          (bin)   0000 0000 0000 0000 0000 0000 0000 0000
                  [Table Level]   AAAA AAAA AAAA BBBB CCCC PPPP PPPP PPPP

         BASE                                          --STATUS---
LEVEL    ADDR      INDEX LOCATION     CONTENTS      L/U LIMIT S CI  M U WP DT
CRP                                80000002 000F4000  L  0000                SHORT
A      000F4000     0H   000F4000           000F8002                  0 0    SHORT
B      000F8000     0H   000F8000           00010001            0   0 0 0    EARLY
PAGE   00010000




STATUS:   M68030--Running in monitor     Trace complete________________........
 display  mmu_tables  root_ptr  CRP  logical_address  0


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

Figure 4-2. How Logical Address 0 Is Mapped

How To See The Table Details For Mapping A Specific Logical Address

The host program can read the details of the mapping for a particular logical address and provide a display like the one shown in figure 4-2. To obtain a display of the mapping for a particular logical address, enter a command such as:

display mmu_tables root_ptr CRP logical_address 0

The display will show you the path taken through the tables to map the logical address in your command to its corresponding physical address. The last entry in an mmu_tables display might point to a page in physical memory where the code for the logical address you specified resides. Early terminations and indirect terminations may also show in the mmu_tables display, depending on techniques used in your operating system.

Interpreting The MMU Tables Display

The content of an MMU TABLES: display is interpreted in Table 4-1 of this chapter.

Table 4-1. Key To The MMU TABLES Display

MMU TABLES NOTATION	MEANING
LOGICAL ADDR(hex)	The logical address whose mapping is shown on the display.
LOGICAL ADDR(bin)	The binary value of the logical address - shown so that you can see how each of its bits is distributed among the table maps.
[Table Level]	FCODE=Func. Code (MOT RSV=Motorola RSV), A=Table A, B=Table B, C=Table C, D=Table D, P=Page, I=Ignore.
LEVEL	CRP=CPU root pointer, SRP=Supervisor root pointer, your own root value, A=Table A, B=Table B, C=Table C, D=Table D, IND=indirection, PAGE=page in physical memory where code for logical address resides.
BASE ADDR	Base address of the table specified in the preceding table or root pointer, plus the index.
INDEX LOCATION	Index is expressed both in its logical value and its absolute value.
CONTENTS	The hexadecimal content of the root pointer or specified location in the table. Interpretation of the content is next in the table.
L/U	Interpretation of the value under LIMIT (L=lower, U=upper).
LIMIT	Unsigned index limit.
--STATUS---	The status bits.
S	S=1 when only supervisor access is allowed to this address.
CI	CI=1 when the cache is inhibited during access to this address.
M	M=1 when the content of the page where this address resides has been modified by a write or read-modify-write instruction.
U	U=1 when the associated address has been accessed.
WP	WP=1 when the associated address is write protected.
DT	Descriptor type of the associated table or root pointer. May be SHORT, LONG, PAGE, EARLY termination, or INVALID.

Mapping Using Function Codes

Figure 4-3 shows an example of an MMU Mappings display for a 68030 processor that is using the function codes as its first level in the memory mapping process. The analyzer adds"Function Code" headings above each separate block of addresses to identify the addresses governed by each function code.

Figure 4-4 shows an example of an MMU Tables display that tracks the mappings that apply to a single address through a set of tables that begin with function-code mapping.

Error Messages Unique To mmu_tables Displays

The following error messages will only appear when you are using an active MMU, expecially examining its operating details with the mmu_tables and mmu_mappings displays:

ERROR: Invalid translation control value.
ERROR: Invalid translation control (sre) value.
ERROR: Invalid translation control (fcl) value.
If one of the three error messages above appear, they indicate that the TC register value that governs this mapping is incorrect. Two of the messages further specify that the "sre" or "fcl" components of the translation control register contain the problem. Refer to the detailed explanation of the TC register later in this chapter.

ERROR: Invalid root pointer value. This message appears when the root pointer value you selected to govern this mapping is incorrect.

ERROR: Root pointer does not match function code value. If you are using the supervisor root pointer (SRP), your function code selections must always be in supervisor regions: SUPER_PROG, or SUPER_DATA. This message indicates that you combined the SRP with a non-supervisor function code.

ERROR: Invalid table level. This message indicates that you've asked for a table level that the emulator/analyzer can't find because the translation to that table is undefined. Either you have asked to see a table that is not activated according to your translation control register value, or an invalid condition in the mapping of the MMU tables has occurred before the path gets to the level you requested.

```
VALID M68030 MMU MAPPINGS:
    LOGICAL ADDRESS       PHYSICAL ADDRESS
    Lower      Upper      Lower      Upper
Function Code =  USER DATA
  ED124000   ED127FFF   80000000   80003FFF
  ED128000   ED12BFFF   80000000   80003FFF
  ED138000   ED13BFFF   80000000   80003FFF
  ED13C000   ED13FFFF   80000000   80003FFF
  ED164000   ED167FFF   80000000   80003FFF
  ED168000   ED16BFFF   80000000   80003FFF
  ED170000   ED173FFF   80000000   80003FFF
  F0000000   F0FFFFFF   FF000000   FFFFFFFF
  F4000000   F4FFFFFF   FF000000   FFFFFFFF
  F6000000   F6FFFFFF   FF000000   FFFFFFFF
  F8000000   F8FFFFFF   FE000000   FEFFFFFF
  FF000000   FFFFFFFF   FE000000   FEFFFFFF
Function Code =  USER PROG
         0   FFFFFFFF   00000000   FFFFFFFF

STATUS:    M68030--Running in monitor      Trace complete________________........
 display  mmu_mappings  root_ptr  CRP  show_map_from  fcode USER_DATA  logical_address
0ED124000h


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

Figure 4-3. Valid MMU Mappings Using Function Codes

```
M68030 MMU TABLES:
      LOGICAL ADDR(hex)   MOT RSV 0      0    2    0    0    0    0    0    0
                  (bin)    000         0000 0010 0000 0000 0000 0000 0000 0000
          [Table Level]    FCODE       AAAA BBBB CCCC CDDD DDPP PPPP PPPP PPPP

         BASE                                           --STATUS---
LEVEL    ADDR       INDEX LOCATION      CONTENTS      L/U LIMIT S CI  M  U  WP DT
CRP                                 80000002 FFFF1000  L  0000                 SHORT
FCODE  FFFF1000      0H  FFFF1000          FFFF1022                     0  0   SHORT
A      FFFF1020      0H  FFFF1020          FFFF1062                     0  0   SHORT
B      FFFF1060      2H  FFFF1068          FFFF10A2                     0  0   SHORT
C      FFFF10A0      0H  FFFF10A0          3C400001              0   0  0  0   EARLY
PAGE   3C400000




STATUS:    M68030--Running in monitor      Trace complete________________........
 display  mmu_tables  root_ptr  CRP  fcode USER_DATA  logical_address  02000000h

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

Figure 4-4. Example Of Mapping An Address Using Fcode

ERROR: Attempt to read guarded memory, addr = <physical address>. This message appears when the logical address you selected is mapped to a non-existent physical address. You will need to review the table structure leading up to this invalid address. Perhaps your operating system created this invalid mapping. If your table structures seem correct, you may want to invoke the memory-mapping portion of your emulation configuration and map memory to support the address that generated this message.

Troubleshooting INVALID Mappings

The following paragraphs show you how to detect and troubleshoot invalid address mappings in translation tables. Figure 4-5 is an MMU Tables display for logical address 0. It shows that an invalid condition exists in Table A at its base address 0FFFF14A0H.

Figure 4-6 displays the details of the address mappings in Table A, beginning with its base address 0FFFF14A0H. The first eight entries in Table A are invalid.

With the above information, you can use the "modify memory" capability of the emulator to write correct contents into the first eight locations in Table A (address 0FFFF14A0H needs contents that correctly identify the next level in the mapping scheme, Table B, Page, etc.). In the case of figures 4-5 and 4-6, the following command would write a correct value:

modify memory long physical 0FFFF14A0H to 8000FC02H, FFFF1350H

```
M68030 MMU TABLES:
      LOGICAL ADDR(hex)  SUPER PROG      0    0    0    0    0    0    0    0
                  (bin)     110       0000 0000 0000 0000 0000 0000 0000 0000
          [Table Level]    FCODE      AAAA BBBB CCCC CDDD DDPP PPPP PPPP PPPP

          BASE                                         --STATUS---
LEVEL     ADDR       INDEX LOCATION      CONTENTS     L/U LIMIT S CI M U WP DT
CRP                                 80000002 FFFF1000  L  0000               SHORT
FCODE   FFFF1000      6H  FFFF1018           FFFF14A3                 0 0  LONG
A       FFFF14A0      0H  FFFF14A0           00000000                     INVALID
PAGE    ------




STATUS:   M68030--Running in monitor      Trace complete________________........
 display  mmu_tables  root_ptr  CRP  fcode SUPER PROG  logical_address  0h


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

Figure 4-5. Using mmu_tables To Track Invalid Mapping

```
M68030 MMU TABLES:  Show Level A
      LOGICAL ADDR(hex)  SUPER PROG      0    0    0    0    0    0    0    0
                  (bin)     110       0000 0000 0000 0000 0000 0000 0000 0000
          [Table Level]    FCODE      AAAA BBBB CCCC CDDD DDPP PPPP PPPP PPPP

                                         --STATUS---                      TABLE
  INDEX LOCATION        CONTENTS        L/U LIMIT  S CI M U WP DT            ADDR
    0H  FFFF14A0    00000000  00000000                         INVALID
    1H  FFFF14A8    00000000  00000000                         INVALID
    2H  FFFF14B0    00000000  00000000                         INVALID
    3H  FFFF14B8    00000000  00000000                         INVALID
    4H  FFFF14C0    00000000  00000000                         INVALID
    5H  FFFF14C8    00000000  00000000                         INVALID
    6H  FFFF14D0    00000000  00000000                         INVALID
    7H  FFFF14D8    00000000  00000000                         INVALID
    8H  FFFF14E0    8000FC02  FFFF1360   L  0000   0      0 0 SHORT    FFFF1360
    9H  FFFF14E8    8000FC02  FFFF1360   L  0000   0      0 0 SHORT    FFFF1360
    AH  FFFF14F0    8000FC02  FFFF1360   L  0000   0      0 0 SHORT    FFFF1360

STATUS:   M68030--Running in monitor      Trace complete________________........
 display  mmu_tables  root_ptr  CRP  fcode SUPER PROG  logical address 0h
show_table_level A

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

Figure 4-6. Address Mapping Details In Table A

The Symbols Problem And Its Solution

The following paragraphs discuss how the emulator can place physical addresses on the emulation bus, and the analyzer can accept symbols defined for addresses in logical memory and cross-reference them to those physical addresses.

The Problem Of Cross Referencing Symbols In Logical Memory To Addresses In Physical Memory

The Problem: Symbols are used by the analyzer to identify addresses. The emulator cross references the symbols you defined in your source files with the addresses where the related code is stored in logical (virtual) memory. The relationship of the symbols to addresses works fine until you turn on the Memory Management Unit (MMU) of the 68030. The MMU translates logical addresses to physical addresses and places the physical addresses on the processor address bus. The analyzer has no way to cross reference logical address symbols to physical addresses so it can't use symbols.

The Solution: The deMMUer translates the physical addresses back to logical addresses and supplies these reverse-translated addresses to the analyzer. The analyzer ignores addresses on the emulation bus when the MMU is enabled; it uses the addresses it receives from the deMMUer.

The remaining pages of this chapter show you how to set up the HP 68030 deMMUer Analysis Bus Generator (deMMUer) to supply logical addresses to the internal analyzer when the 68030 MMU is placing physical addresses on the emulation bus. These pages also describe how the deMMUer operates, and restrictions you should observe when using the deMMUer.

When Should I Start The DeMMUer?

You can start the deMMUer at the same time as the 68030 MMU starts, or you can turn on the deMMUer after the MMU has been operating. Each case is discussed in the following paragraphs:

Startup With The Emulator

The best time to start the deMMUer is just before beginning a run of program. The deMMUer flushes its reverse translations as part of the processor reset procedure. This ensures that the translation tables within the deMMUer contain no old translations. Then the deMMUer waits to detect the first table search performed by the

68030 processor. Logical addresses are available from the deMMUer immediately after reset. All table searches are monitored, keeping the deMMUer physical-to-logical address translations up to date.

The Emulator Was Running Without The DeMMUer, And Now I Want To Use It

You can enable the deMMUer any time during operation of the 68030 Emulator/Analyzer, and it will output current logical address translations. The physical-to-logical reverse translations are built up and maintained from reset, even when the deMMUer is disabled (outputting physical addresses).

How To Turn On And Turn Off The DeMMUer

There are two ways to turn on and turn off the deMMUer: one is by setting the analysis mode, and the other is by invoking the emulation configuration set of questions. Each is described below.

Note

You may turn on the deMMUer and still have only physical address information. The deMMUer can only translate physical addresses to logical addresses after you have (1) enabled the MMU of the 68030 processor, (2) set up a valid deMMUer configuration, and (3) enabled the deMMUer. The way to set up the deMMUer configuration and enable the deMMUer is discussed later in this chapter.
You will not need address translations when the 68030 MMU is off.

Turn On/Off By Using Emulation Configuration Questions

Invoke the emulation configuration questions by using the **modify configuration** command. Proceed through the questions until the following configuration questions can be answered:

Modify memory configuration? **yes**

In the memory mapping display, enter the following command:

configure_deMMUer

In the deMMUer configuration display, enter the following command:

enable_deMMUer

Even though you have activated the deMMUer, it will still provide physical address information for analysis until it has been loaded with a valid configuration (discussed next in this chapter).

Once turned on, the deMMUer will track the MMU activity, and update its translation tables each time the MMU makes a change to its translation tables. (Note that the MMU is turned on or off by a different emulation configuration question that appears after the memory mapping display:

"MMU enabled during session? **yes**"

disable_deMMUer

This turns off the deMMUer. Only physical addresses will be supplied to the analyzer. Therefore, only the physical analysis mode will be available.

Turn On/Off By Setting The Analysis Mode

If you have the 68030 MMU enabled, and you have a valid value in the TC register of the deMMUer configuration, and you have enabled the deMMUer, then you can turn on the deMMUer from within an emulation session, by entering the following commands:

set analysis mode logical

This turns on the deMMUer, providing logical addresses to the analyzer. The analyzer uses these addresses to perform symbol searches to satisfy trace specifications and show symbols in trace lists.

set analysis mode physical

This turns off the deMMUer. Physical addresses from the 68030 MMU will be supplied to the analyzer. The trace lists will show the physical addresses, but the analyzer will not accept or display source-file symbols.

Description Of The DeMMUer Configuration Display

The deMMUer configuration display is shown in figure 4-7. You must set up this configuration with valid entries before the deMMUer can perform its reverse address translations. Setup instructions are described later in this chapter.

Figure 4-7 shows a typical display rather than a default display. The component parts of the deMMUer configuration display are described in the following paragraphs.

Figure 4-7 shows the deMMUer hardware is enabled. The deMMUer will follow changes in the 68030 MMU. If the deMMUer were disabled, only physical addresses from the 68030 MMU would be supplied to the analyzer.

```
deMMUer configuration
----------------------------------------------------------------------------
deMMUer hardware enabled


Translation Control - 82CF5000H
        <e      sre  fcl   ps    is   tia   tib   tic   tid>
         1       1    0    4 Kb  15    5     -     -     -
    * deMMUer cannot follow changes in ignored address bits


Virtual address start - 00000000


Root Descriptor Type -> long Descriptor



Range List -       Start          End
  A             00000000     00010000
  B             10000000     10800000
  C             0C000000     0C040000
  D             undefined range
STATUS:   Configuring DeMMUer__________________________________________...R....



_range__ _enable_ _disable_ __set___         display_ _return_ ________ ________
```

Figure 4-7. DeMMUer Configuration Display

The content of the 68030 Translation Control (TC) register is shown. It is decoded in this display so you can see its 68030 MMU mnemonics. The mnemonics are defined in table 4-2.

Virtual Address start - You can enter an address pattern here in cases where a number of upper address bits are being ignored by

Table 4-2. Definition Of The TC Register Mnemonics

TC Register Identifier	Definition And Description
e	68030 MMU Address Translation Enabled. 1 = MMU enabled to translate addresses from logical to physical. 0= MMU disabled. Logical addresses = physical addresses. Note that the 68030 MMU can be disabled by hardware, and by the **MMUDIS** pin.
sre	Supervisor Root Pointer Enable. 1 = 68030 supervisor root pointer enabled to point to supervisor function code translation table. 0 = 68030 supervisor root pointer disabled. Both user and supervisor accesses will use the translation table defined by the CRP.
fcl	Function Code Lookup. 1 = first level in address translation table structure is indexed by the function code. 0 = function codes are not part of translation process. The first level of translation tables within the translation table structure is indexed by the bits identified by tia.
ps	Physical Memory Page Size. Figure 4-1 shows page size is 4 kilobits. The least significant 12 address bits are not translated to find a page in physical memory.
is	Initial Shift. This is the number of upper address bits ignored during table searches.
tia - tid	Address Translation Table Indexes Numbers of logical address bits used as indexes for each level of the translation tables (not including optional level indexed by function codes).

the 68030. These upper address bits will be used instead of the values of the ignored bits to determine which symbols the analyzer will show on its display. For example, if the 68030 is ignoring upper address bits 31 and 30, those logical address bit values are not detectable by the deMMUer. Therefore, the deMMUer will assume the bits are 0 unless specified by the Virtual Address Start field. If you know that all executions are being performed in the logical address space whose upper bits 31 and 30 are 01, then enter 40000000H in this field.

Root Descriptor Type - This tells the deMMUer whether the descriptor types in the root pointers are short format, long format, or page descriptors. Only one of these formats can be selected even though there are two root pointers. Refer to How To Select A Root Pointer Descriptor Type later in this chapter.

Range - Up to four ranges of physical memory can be defined for the deMMUer to translate. Three ranges were defined in the example display. The ranges must begin and end on 64K byte boundaries. A range can have any size from 64K bytes through 4 megabytes (starting and ending on 64K byte boundaries).

How To Obtain Information Needed To Configure The DeMMUer

You will need to enter valid information into the deMMUer configuration before the deMMUer can translate physical addresses to logical addresses. Some of the information you will need to set up the deMMUer configuration is present in the 68030 MMU: the value in the translation control register, and the descriptor type of the root pointer. There are two ways to obtain this inforamtion:

1. Start the system with internal analysis set to trigger on the first table search. Then when the 68030 does its first table search, you can look at the trace list a few states before the trigger and see what value was written into the TC register of the MMU. You will want to write the same value into the TC register of the deMMUer.

2. In some cases, the MMU is enabled long before the first table search, and its TC register and root pointer will have already been automatically set up. In these cases, you can select

display registers mmu

Here you can see the present value of the TC register of the MMU, and also the Root Pointer Descriptor Type. Copy these values. You will need to enter them in the TC register and Root Pointer Descriptor Type fields of the deMMUer configuration.

When you have the value of the translation control register in the 68030 MMU, and you know the root pointer descriptor type, you are ready to access the deMMUer configuration and set up the fields in the display.

Some of the information you will need to enter into the deMMUer configuration display is information specific to use of the deMMUer. The way to obtain this information is discussed later in this chapter.

How To Access The DeMMUer Configuration Display

Invoke the emulation configuration questions by using the **modify configuration** command. Proceed through the questions until the following configuration questions can be answered:

Modify memory configuration? **yes**

In the memory mapping display, enter the following command:

configure_deMMUer

In the deMMUer configuration display, you can turn the deMMUer on or off and define values and ranges to be used by the deMMUer during its operation. The procedures you follow to make these entries are discussed in the following paragraphs.

When you are finished configuring the deMMUer, return to the memory mapping display by using the **return** command. With a valid configuration setup, the deMMUer will be able to perform its reverse address translations.

How To Enter (Or Delete) A Range Of Addresses For The DeMMUer To Follow

Access the deMMUer configuration display, and enter the following commands:

range A <value> **thru** <value>
range B <value> **thru** <value>

These commands tell the deMMUer which ranges of physical addresses to translate to logical addresses. You may not want the entire range of physical addresses to be translated to logical addresses. The deMMUer will only perform translations of addresses within the physical address ranges you specify here.

You can enter up to four physical address ranges. Each range can be as small as 64K bytes, or as large as 4 megabytes. The ranges will be rounded up to 64K byte boundaries, automatically. The maximum range of physical addresses that the deMMUer can translate is 16 megabytes. If you try to enter a single range larger than 4 megabytes, the present range specification will not change, and an error message will be displayed on the STATUS line.

You can delete a range specification by entering a command such as:

range A clear

This restores the entry to the default condition: "undefined range".

Note

If the values of 68030 MMU registers TT0 and TT1 are changed by your program during a run, they may identify ranges of addresses to be untranslated within ranges that you defined in the deMMUer configuration to be translated. The deMMUer has no way of detecting changes to TT0 and TT1. As a result, if such changes occur, the deMMUer will continue to output logical addresses that are not 1:1 mappings of the physical addresses which are guaranteed by the TT0 and TT1 registers.

How To Enter A New Value For The Translation Control Register

Access the deMMUer configuration display, and enter the following commands:

set TC_register <enter the desired value>

The deMMUer will check to make sure you entered a valid value for the translation control register. If your entry is valid, the deMMUer will accept it. If not valid, the translation control register will not accept it, and the following message will be displayed on the STATUS line:

ERROR: Invalid value for TC_register.

The following information shows you the format for entry of a value in the translation control register:

E	**SRE/FCL**	**PS**	**IS**	**TIA**	**TIB**	**TIC**	**TID**

E can be 8 or 0.
  8 = TC_register enabled.
  0 = TC_register disabled.
  1 through 7 and 9 through 15 are undefined.

SRE/FCL can be 0 through 3.
  0 = SRE disabled, FCL disabled.
  1 = SRE disabled, FCL enabled.
  2 = SRE enabled, FCL disabled.
  3 = SRE enabled, FCL enabled.
  4 through 15 are undefined.

PS can be any number from 8 through 15.
  8 = 8 bits will address within a 256-byte page size.
  9 = 9 bits will address within a 512-byte page size.
  10 = 10 bits will address within a1K-byte page size.
  11 = 11 bits will address within a 2K-byte page size.
  12 = 12 bits will address within a 4K-byte page size.
  13 = 13 bits will address within a 8K-byte page size.

14 = 14 bits will address within 16K-byte page size.
15 = 15 bits will address within 32K-byte page size.
7 through 7 are undefined.

IS can be any number from 0 through 15.
n = the first n high-order bits will be ignored in the logical address when indexing into the translation tables.

TIA is the number of bits used to index in the first table (or second table if FCL is used to index into the first table).

TIB is number of bits used to index into the tables indicated by TIA.

TIC is number of bits used to index into the tables indicated by TIB.

TID is number of bits used to index in the tables indicated by TIC.

In a valid number, the following is true:
PS + IS + TIA + TIB + TIC + TID = 32.
This ensures that every bit in the address either addresses a byte on the page, is part of the index at some level of the address table, or is ignored by "IS".

When TIA, TIB, or TIC is 0, all the subsequent field values are ignored.

The following is an example of a valid number:

82A68800H
8 = enabled
2=SRE enabled, FCL disabled
A=1K page size
6=first 6 high-order bits are ignored for the index.
8=next 8 high-order bits are used to index into the first level of translation tables (TIA).
8=next 8 high-order bits are used to index into the second level of translation tables (TIB).
0=no translation tables this deep. The page address resides in the preceding level of tables. Subsequent digits (for TID in this example) are ignored.
H=the value is expressed as a hexadecimal number.

How To Select A Root Pointer Descriptor_Type

The root pointer contains the address of the top level table of the translation tree. The address can be in one of three formats: short, long, or page. To inform the deMMUer which of the three formats is used by the address in the root pointer, access the deMMUer configuration display and enter the following command:

set descriptor_type <short, page, or **long>**

If you select **short**, the deMMUer assumes the first table in the translation table tree contains short format descriptors. The 68030 multiplies the address bit field being translated at this level by four to access the next level in the translation table. Short format descriptors must be long-word aligned.

If you select **long**, the deMMUer assumes the first table in the translation table tree contains long format descriptors. The 68030 multiplies the address bit field being translated at this level by eight to access the next level in the translation table. Long format descriptors must be quad-word aligned.

If you select **page**, the deMMUer can be turned off. This is because the address translation is no more than an offset added to the logical address to obtain the physical address. No table searches will occur when the DT is **page**. This means that translation tables within the deMMUer will never be updated. The physical addresses will simply be passed through the deMMUer without translation.

In the case where **page** is selected, use the offset feature available in the trace specification and in the trace list of the analyzer to add the required offset to values in your trace specification, and to obtain correct addresses in your trace lists.

In the case where the 68030 MMU has one descriptor type in its CRP root pointer and a different descriptor type in its SRP root pointer, use the following information as a guide when selecting the appropriate descriptor for the deMMUer root pointer descriptor type:

1. If one of the root pointer descriptor types is **page**, select the type specified for the other root pointer.

2. If one of the root pointer descriptor types is **short** and the other is **long**, select one of the following two options:

   a. Pick the root pointer that points to the code you want to analyze (either user or supervisor code), and use the descriptor type that is assigned to that root pointer. Accesses governed by the other root pointer will not provide correct logical address information

   b. Use function-code lookup at the first level of indexing into the translation tables. In this way, the root pointer descriptor type will not affect operation. The root pointer descriptor type is only used when the deMMUer calculates the index into the first level of translation tables.

How To Enter Upper Address Bits In The Virtual Address Start Field

Access the deMMUer configuration display, and enter the following command:

set virtual_address <enter the desired value>

This is your virtual address start value. You can enter it using any of the four number bases (binary, octal, decimal, or hexadecimal). This field defines the values of the logical address bits that are being ignored by the Initial Shift field.

There may be a problem when using Initial Shift to ignore some upper address bits, and then using Virtual Address Start to specify values for the ignored bits. The deMMUer cannot detect changes in the ignored bits during a run of the program. If your program resides in two or more logical address ranges (identified by different bits in the ignored set of bits), the output of the deMMUer will be incorrect when execution is in one of the unspecified ranges. The assumption was made that the ignored bits would remain constant during a run of program.

How Does The DeMMUer Work?

The physical address from the 68030 MMU is supplied as an input to the deMMUer. The deMMUer contains a set of translations like those in the 68030 MMU. The deMMUer translations provide the reverse function of the translations in the MMU (given a physical address, they look up the logical address from which it was derived). The deMMUer outputs the logical address corresponding to the physical address from the 68030 MMU.

Each time the 68030 MMU performs a table search, the deMMUer detects the event and follows MMU activity to build a corresponding reverse-address translation.

If you have the deMMUer enabled from the time you start the 68030 MMU, the deMMUer will have current translations to reverse each of the translations performed by the MMU. This is true whether you select trace lists with physical or logical addresses.

Note

Be sure to flush the address translation cache (ATC) of the MMU before enabling the MMU. Otherwise, out-of-date translations (logical to physical) may reside in the ATC. There is no facility in the 68030 emulator/analyzer to flush the ATC. You can include an option to the command that loads the TC register or loads the root pointer to ensure that the ATC is flushed after reset.

For addresses that the deMMUer has no translation, it supplies the physical address that was output by the 68030 MMU, and tags it as being a physical address (places a "p" in front of it in the Address column of the trace list). The analyzer will show this address in its trace list, but it will not be able to show any symbol associated with this address, nor will it be able to recognize any trace commands occurring on this address if those commands are specified using source-file symbols.

When Do I Need To Use The DeMMUer?

You need to use the deMMUer when the 68030 MMU is active, and you want to use any of the following features during analysis of a program:

1. You want the trace list to show the assembly language form of the activity it captured during a trace. The inverse assembler requires sequential logical addresses in order to look up the next piece of program information. Physical addresses will probably be non-sequential when crossing a page boundary.

2. You want to enter a trace specification that will be met when a certain source-file event appears during a trace. To do this, you enter the name of the source-file symbol that identifies that event. Basic trigger/store/count features are not supported for code in physical addresses. In a dynamic environment, the relationship between an instruction or data location and its physical address may not be constant throughout the running of a program.

3. You want the trace list to show address values in terms of the symbol names assigned in the source files. Symbol and source line referencing operates on the fact that a symbol or source line resides at a particular logical address. That relationship is established with the language tools. The source referencing has no knowledge of physical addresses.

4. You want to perform high-level analysis on the program you are developing by using such tools as the HP Software Performance Measurement Tool (SPMT). High-level analysis tools, such as SPMT, gather data based on logical address information. These tools have no facilities for performing physical-to-logical address translations.

When Do I Need To Turn Off The DeMMUer?

Turn off the deMMUer when you want to trace activity that shows the addresses within the physical memory. This information may be useful when you are analyzing the behavior of your operating system.

Under What Conditions Will The DeMMUer Fail To Work?

There are three conditions under which the deMMUer will fail to perform reverse-address translations correctly:

1. If the root pointers use **page** descriptor DT fields. In this case, no table searches will occur. Physical addresses will equal logical addresses plus the offset specified in the root pointer.

2. If the two root pointer DT fields are different types (for example, one short and the other long), and both root pointers are used, the deMMUer will fail to work because the deMMUer has facilities for only one root-pointer definition. Refer to How To Select A Root Pointer Descriptor Type earlier in this chapter for suggestions concerning how to handle this problem.

3. If Level A is the first table to be accessed (you are not using function-code lookup), then the first "n" low-order bits in the root pointer (CRP and SRP registers) table address must be 0. During the boot-up routine, the root pointer table address is normally loaded just before the TC register is loaded and enabled in the 68030 MMU; make sure the low-order "n" bits of the root pointer table address are 0.

   How to calculate the value of "n" in 3 above:

   a. If the root pointer DT value is SHORT, then
      "n" = 2 + the value of TIA in the TC register.
      Example: If TIA = 8 bits, then "n" = 10.
      This means the bottom 10 bits of the root pointer table address must be 0's.

   b. If the root pointer DT value is LONG, then
      "n" = 3 + the value of TIA in the TC register.
      Example: If TIA = 8 bits, then "n" = 11.
      This means the bottom 11 bits of the root pointer table address must be 0's.

Example Procedure Using DeMMUer In Memory Management

This procedure shows you how to set up and use the 68030 MMU and the deMMUer of the internal analyzer to manage code in a virtual system and supply the corresponding logical addresses to the internal analyzer. The internal analyzer must have logical addresses in order to accept commands specified using source-file symbols and segment names, and provide trace lists that show addresses in terms of the symbols and code segments being traced.

The "os" and "towers" programs which are supplied with your emulation/analysis software are used in this example. If you are using your system to develop a different operating system or application program, change the entries you make accordingly.

The Initial Conditions

To begin, the 68030 MMU must be hardware-enabled so that the operating system can activate it and set it up. The emulator will hardware-enable the 68030 MMU when you follow the procedure described in the paragraph titled "Turn On The 68030 MMU" in this chapter. If you are running this demonstration with your emulator connected to a target system, make sure your target system has not asserted the MMU disable pin ($\overline{\text{MMUDIS}}$) of the 68030.

Access The Emulator

If you're not already in your emulation session, gain access to it, and load the emulation configuration with commands similar to the following:

<meas_sys> em68030 Return

load configuration preconfig Return

This configuration allocates 1 megabyte of physical memory for the emulator.

Turn Off The DeMMUer

The configuration "preconfig" begins with the deMMUer hardware disabled, and the fields of the deMMUer configuration in their boot-up states. To ensure that this initial condition exists, invoke the emulation configuration questions and answer them as follows:

modify configuration Return

Proceed to the memory mapping question and answer it as follows:

Modify memory configuration? yes Return

Go to the memory map and enter:

configure_deMMUer Return

With the deMMUer configuration on screen, enter the following command:

disable_deMMUer Return

This disables the deMMUer hardware. Now exit the deMMUer configuration display, and the memory mapping configuration by using the following commands:

return Return

end Return

Turn On The 68030 MMU

This is done next in the emulation configuration questions. This provides the 68030 MMU hardware enable. It must be done so that the operating system will be able to activate it and set it up. Enter the following commands:

Modify emulator pod configuration? yes Return

Accept the next two questions with their present answers, and make sure the third question is answered as follows:

MMU **enabled during session? yes Return**

Accept the rest of the questions in the configuration set of questions. Assign the file name "preconfig" to this configuration file (if not already assigned) by using the following command:

Configuration file name? /..your directory../preconfig Return

Load The Operating System And Get MMU Information

Now load and run the operating system program "os". The "os" program is a small operating system routine that performs the software-enable and loads appropriate values in the 68030 MMU so it can perform table searches and manage memory. Use the following commands:

reset Return

load memory physical os Return

The "os" program is a short operating-system script that sets up the 68030 MMU to manage memory for this demonstration program.

In "os", physical memory is mapped 1:1 to logical memory. As a result, addresses do not change when the MMU is enabled. Mapping the memory 1:1 is not required for operation of the emulator, but it serves to simplify this example.

Use the following commands to run "os":

trace Return

run from entry Return

When the STATUS line of the display shows Trace complete, "os" will have done its work.

Set Up The DeMMUer Configuration

Now gather the information you need to set up the deMMUer configuration. This information is available in the setup of the 68030 MMU. Enter the following commands:

display registers mmu Return

See figure 5-1. Copy the values of the TC register and the CRP root pointer descriptor. You will need to use these same values in the deMMUer configuration:

```
M68030 Registers    :mmu

 NextPC 000F0040       SFC 0 MOT RSVD            DFC 0 MOT RSVD
  D0-D7 00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  A0-A6 00010000 00000000 00000000 00000000 00000000 000F0800 00000000
    USP 000F0F00                 1 t0 s  m  i  x  n  z  v  c     CAAR 00000000
    MSP 000F0F00    STATUS 2000  0  0  1  0  0  0  0  0  0  0        VBR 00000000
   *ISP 000F0EF4                   wa dbe  fd  ed ibe  fi  ei
                      CACR 0011     0   0   0   0   1   0   1

                              L/U LIMIT  DT      TABLE ADDR
    SRP  30C05180FDDFFFFF      U  30C0  INVALID    FDDFFFF
    CRP  80000002000F4000      L  0000  SHORT      000F400
                   l s w i m t n                      TT0  FFFF0777
    MMUSR  0200    0 0 0 0 0 1 0 0                       TT1  FFFF0777
                     sre  fcl   ps   is  tia tib tic tid
    TC 80C0C440   1    0    0    4 Kb  0   C   4   4   0

STATUS:   M68030--Running                 Trace complete________________........
```

Figure 5-1. MMU Registers Display

TC register value = 80C0C440H

CRP = SHORT

Now you have enough information to set up the deMMUer configuration display. Again, invoke the emulation configuration questions, and access the deMMUer configuration display, as follows:

modify configuration Return

Go to the memory mapping question, and answer it as follows:

Modify memory configuration? yes Return

Go to the memory map and enter:

configure_deMMUer Return

With the information you obtained from the MMU register display, enter the following deMMUer configuration commands:

set TC_register 80c0c440h Return

set descriptor_type short Return

The above two values are the ones you copied from the MMU registers display. Now enter the next two commands to complete the deMMUer configuration:

range A 00h thru 0ffffffh Return

enable_deMMUer Return

Your display should look like that shown in figure 5-2. Range A is assigned to have the same range of addresses as those mapped to emulation memory because you want address translations in all areas of program execution. The last command enables the hardware of the deMMUer.

Now you are finished setting up the emulation configuration to support memory management. Use the following commands:

return Return

end Return

```
deMMUer configuration
---------------------------------------------------------------------------
deMMUer hardware enabled

Translation Control - 80C0C440H
        <e      sre   fcl    ps     is    tia    tib    tic    tid>
         1       0     0     4 Kb    0    12      4      4      -

Virtual address start - 00000000H

Root Descriptor Type ->Short Descriptor


Range List -       Start             End
  A             00000000H      000FFFFFH
  B                 undefined range
  C                 undefined range
  D                 undefined range
STATUS:   Configuring DeMMUer__________________________________________...R....
 enable_deMMUer


_range__ _enable_ _disable_ __set___         display_ _return_ ________ ________
```

Figure 5-2. DeMMUer Configuration Setup

Accept the rest of the questions with their present answers. Name this configuration "testconfig" by answering the last configuration question as follows:

Configuration file name? /..your directory../testconfig Return

In the future, when making tests for this operating system and application program, you can simply load configuration testconfig, and your MMU and deMMUer will be properly set up.

Viewing The Execution Of "os"

The next series of trace lists show execution of the operating system script that was used to set up the 68030 MMU. These trace lists also show important considerations you need to know in order to read trace lists made when the 68030 MMU is managing memory and the deMMUer is supplying logical addresses to the internal analyzer.

set analysis mode logical Return

This command turns on the deMMUer (by providing its software enable) so that it will supply logical addresses to the analyzer for storage in its trace memory. The "mode" command token is only available when the deMMUer is hardware enabled, which is why you haven't seen it before.

Notice that you can select the storage of either **logical** or **physical** addresses with this command. You will find **logical** address information most useful when developing application programs, and **physical** address information most useful when developing operating systems. The analyzer memory cannot store both logical and physical addresses for each state in trace memory; that is why you make this selection before you start the trace.

You have to reload the operating system program into the emulation system because the emulation memory configuration has been changed since the program was loaded last. Enter the following commands:

reset Return

load memory physical os Return

Start the analyzer and re-run the operating system program "os". These commands will trace "os" using addresses suppllied by the deMMUer. Enter the following commands:

trace Return

display trace symbols on Return

run from entry Return

The "entry" symbol used in the last command is a symbol in the data base of the "os" program. This starts the "os" program running at the appropriate point. See figure 5-3.

```
Trace List                                         Mode:logical address
Label:       Address                     Opcode or Status                time count
Base:        symbols                    mnemonic w/symbols                relative
-0007
-0006
-0005
-0004
-0003
-0002
-0001
trigger AB|entry.s:reset    $00000E2A     supr prgm long rd log addr ------------
+0001        abs 00000004   $00000A00     supr prgm long rd log addr    0.40us
+0002      ABS|STACKTOP     $2700----     supr data word wr log addr    338.ms
+0003        abs 000F0F02   $----000F     supr data long wr log addr    12.2us
+0004        abs 000F0F04   $0000----     supr data word wr log addr    0.60us
+0005        abs 000F0F06   $----0000     supr data word wr log addr    12.1us
+0006      ABS|STACKTOP     $2700----     supr data word rd log addr    21.4us
+0007        abs 000F0F06   $----0000     supr data word rd log addr    0.40us

STATUS:    M68030--Running                 Trace complete________________...R....
 run from entry


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

Figure 5-3. Trace List Showing Execution Of "os"

Enter the following command:

display trace disassemble_from_line_number 10 Return

See figure 5-4. The command used to obtain this display resynchronized the inverse assembler after a program transfer occurred. The reason this was necessary is that the inverse assembler may lose synchronization when the processor performs a program transfer, such as a jump, call, loop, return, fault, or interrupt.

Notice the instruction on trace memory line 19. The root pointer must always start on a page boundary.

```
Trace List                                       Mode:logical address
Label:       Address                    Opcode or Status               time count
Base:        symbols                   mnemonic w/symbols               relative
+0010   PR|entry.s:entry MOVEA.L      #$000F0F00,A7                     0.60us
+0011                                                                   0.48us
       =e:entry+00000006 MOVEA.L      #$000F0800,A5
+0012   e:entry+00000008   $000F0800    supr prgm long rd log addr      0.48us
+0013   e:entry+0000000C MOVE         SR,D0                             0.52us
       =e:entry+0000000E BFCLR        D0{21:03}
+0014                                                                   0.48us
       =e:entry+00000012 MOVE         D0,SR
+0015   e:entry+00000014 MOVE.W       #$0011,D0                         0.60us
+0016   e:entry+00000014 MOVE.W       #$0011,D0                         0.80us
+0017   e:entry+00000018 MOVEC        D0,CACR                           0.44us
+0018   e:entry+0000001C JSR          PROG|/os.s:_main                  0.52us
+0019                                                                   0.88us
       =PROG|/os.s:_main LEA          DAT|os.s:ROOTPTR,A0
+0020   e:entry+00000020 ORI.B        #$F9,-(A2)                        0.52us

STATUS:   M68030--Running               Trace complete________________...R....
 display trace disassemble_from_line_number 10


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

Figure 5-4. Trace List Resynchronized On Line 10

Enter the following command:

display trace disassemble_from_line_number 20 low_word Return

See figure 5-5. In the trace list shown in figure 5-5, the inverse assembler lost synchronization due to the JSR instruction on line 19. The inverse assembler had to be resynchronized on the low word of the long-word stored in trace memory line number 20. If the "low_word" token had not been used in the above command, inverse assembly would have begun with the high-word (by default), and would have produced an incorrect inverse assembly. (Compare line 20 with trace memory line 20 in figure 5-4 where inverse assembly did begin on the high word.)

```
Trace List                                          Mode:logical address
Label:       Address                      Opcode or Status                 time count
Base:        symbols                     mnemonic w/symbols                 relative
+0020                                                                        0.52us
      =PROG|/os.s:_main LEA            DAT|os.s:ROOTPTR,A0
+0021      abs 000F0EFC    $000F0022     supr data long wr log addr    0.40us
+0022   o:_main+00000002   $000F0800     supr prgm long rd log addr    0.40us
+0023   o:_main+00000006 PMOVE.Q       (A0),CRP                        0.44us
+0024   o:_main+0000000A LEA           DATA|os.s:MMUCTL,A0             0.52us
+0025   DAT|os.s:ROOTPTR   $80000002     supr data long rd log addr    0.60us
+0026   ROOTPTR+00000004   $000F4000     supr data long rd log addr    0.40us
+0027   o:_main+0000000A LEA           DATA|os.s:MMUCTL,A0             0.60us
+0028                                                                  0.44us
      =o:_main+00000010 PMOVE.L       (A0),TC
+0029                                                                  0.48us
      =o:_main+00000014 LEA           $00010000,A0
+0030   o:_main+00000016   $00010000     supr prgm long rd log addr    0.52us
+0031   DATA|os.s:MMUCTL   $80C0C440     supr data long rd log addr    0.40us

STATUS:    M68030--Running               Trace complete________________...R....
 display trace disassemble_from_line_number 20 low_word


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

Figure 5-5. Trace List Resynchronized On Line 20

Enter the following command:

display trace disassemble_from_line_number 36 low_word Return

See figure 5-6. In the trace list shown in figure 5-6, the inverse assembler was resynchronized on the low word of the long-word stored in trace memory line number 36. The event that caused the inverse assembler to lose synchronization was the operating system enabling the 68030 MMU. As in the preceding illustration, execution started with the instruction beginning in the low word following the event.

Figure 5-6 shows the end of the "os" program. When the operating system script "os" supplied in this demonstration has finished setting up the MMU, it executes an infinite loop.

```
Trace List                                          Mode:logical address
Label:       Address                      Opcode or Status                time count
Base:        symbols                     mnemonic w/symbols                relative
+0036                                                                      0.60us
       =o:_main+00000014 LEA            $00010000,A0
+0037   o:_main+00000016   $00010000      supr prgm long rd log addr      0.48us
+0038   o:_main+0000001A PTESTR         #6,(A0),#7                        0.44us
+0039   PROG|g/os.s:SELF JMP            ($FFFE,PC)                        0.48us
+0040          p000F4000   $000F800A      supr data long rd phys add      1.52us
+0041          p000F8004   $00020001      supr data long rd phys add      0.64us
+0042       abs 000F0044 MOVE.B         -(A2),-(A4)                       0.84us
       =    abs 000F0046 incomplete cycle: /0044/????/
+0043   PROG|g/os.s:SELF JMP            ($FFFE,PC)                        0.48us
+0044       abs 000F0044 MOVE.B         -(A2),-(A4)                       0.44us
       =    abs 000F0046 incomplete cycle: /0044/????/
+0045   PROG|g/os.s:SELF JMP            ($FFFE,PC)                        0.44us
+0046       abs 000F0044 MOVE.B         -(A2),-(A4)                       0.44us
       =    abs 000F0046 incomplete cycle: /0044/????/

STATUS:   M68030--Running                Trace complete________________...R....
 display trace disassemble_from_line_number 36 low_word


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

Figure 5-6. Trace List Resynchronized On Line 36

Use the "page-up" key on your keyboard to view the display in figure 5-7. This display shows the table walk that preceded the execution shown in figure 5-6.

Table searches are always performed in physical memory. Note the letter "p" preceding each address shown as part of the table walk (trace memory lines 32 through 35). When the deMMUer is supplying logical addresses and encounters a physical address for which it has no translation, it shows the physical address from the 68030 MMU, and precedes it with the letter "p". The analyzer cannot show any symbols that are associated with physical addresses.

```
Trace List                                       Mode:logical address
Label:       Address                   Opcode or Status                time count
Base:        symbols                  mnemonic w/symbols                 relative
+0023   o:_main+00000006 PMOVE.Q        (A0),CRP                          0.44us
+0024   o:_main+0000000A LEA           DATA|os.s:MMUCTL,A0                0.48us
+0025   DAT|os.s:ROOTPTR    $80000002     supr data long rd log addr      0.60us
+0026   ROOTPTR+00000004    $000F4000     supr data long rd log addr      0.40us
+0027   o:_main+0000000A LEA           DATA|os.s:MMUCTL,A0                0.60us
+0028                                                                     0.48us
       =o:_main+00000010 PMOVE.L       (A0),TC
+0029                                                                     0.48us
       =o:_main+00000014 LEA           $00010000,A0
+0030   o:_main+00000016    $00010000     supr prgm long rd log addr      0.52us
+0031   DATA|os.s:MMUCTL    $80C0C440     supr data long rd log addr      0.40us
+0032          p000F4000    $000F8002     table walk  long rd (strm)      2.60us
+0033          p000F4002    $----800A     table walk  word wr (strm)      0.60us
+0034          p000F803C    $000F0001     table walk  long rd (strm)      0.40us
+0035          p000F803E    $----0009     table walk  word wr (strm)      0.88us

STATUS:   M68030--Running               Trace complete________________........
 display trace disassemble_from_line_number 36 low_word

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

Figure 5-7. Trace List Showing First Table Walk

Load The Application Program

Now that the MMU and deMMUer are both set up to perform memory management and address translations, it is time to load your application program. Enter the following command:

break Return

load memory towers Return

The above command causes the towers program to be loaded logically through the monitor. (This requires several minutes.) The reason "reset" was not used here is that it would shut down the MMU. The MMU must manage the loading, as well as the running, of the application program.

trace TRIGGER_ON a= long_aligned main Return

run from transfer_address Return

The program will run about 15 seconds before the analyzer finds its trigger point and captures a trace.

display trace disassemble_from_line_number 0 Return

display trace source on inverse_video on symbols on Return

See figure 5-8.

```
Trace List                                      Mode:logical address
Label:       Address                  Opcode or Status              time count
Base:        symbols                 mnemonic w/symbols              relative
trigger  000010E8 NOP                                                  0.40us
       #########towers.c - line     1 thru   128 ##############################
       static void towers();
       static int  ask_for_number();

       main()
       {
       = 000010EA LINK.W       A6,#$0000
+0001    7FFFFFC8    $00000A5C     supr data long wr log addr (strm)     0.40us
+0002                                                                  0.40us
       = 000010EE MOVE.L       A3,-(A7)
+0003    000010F0 MOVE.L       A2,-(A7)                                0.48us
       = 000010F2 LEA          ($800C,A5),A0
+0004    7FFFFFC4    $7FFFFFF0     supr data long wr log addr (strm)     0.40us
+0005                                                                  0.40us

STATUS:   M68030--Running                 Trace complete________________........
 display trace source on inverse_video on


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

Figure 5-8. Trace List Showing Entry To Towers

In this display, you can see lines of source code (shown in inverse video on your screen), followed by the states in the trace memory that were emitted by those source lines and executed in your emulation system.

The trigger line in figure 5-8 shows the beginning of execution in the towers program. This program will run just like it did before you set out to use memory management. The deMMUer will provide all of the translations required to allow symbols to be used in your commands, and to allow symbolic addresses to be shown in analyzer trace lists.

6

Dequeued Trace Lists

This chapter shows you how to obtain and read the contents of dequeued trace lists provided by the 68030 internal analyzer. Normal trace list dequeuing is discussed, and examples are shown. Finally, this chapter shows you how to recognize occasions when the dequeuer makes an incorrect assumption, and how you can use dequeuer command tokens to correct the trace list in these events.

The information available from the processor is not sufficient to determine the logical behavior of the processor at all times. If a portion of your dequeued trace list makes no sense, you may have to read the "not-dequeued" trace to determine exactly what the processor did. The dequeuer offers control selections (discussed in this chapter) that you can use to correct your trace list if the dequeuer makes an incorrect assumption.

How To Turn On And Turn Off The Dequeuer

Turn on the dequeuer by using the "dequeued" softkey in a command such as:

display trace disassemble_from_line_number 44 **dequeued**

With the dequeuer on, the trace list shows a logical display of the activity that was executed. The operands are aligned with the instructions that caused them to occur. An operand is shown as a single data value, regardless of how many bus cycles were used to transfer that data value. All of the unused prefetches are eliminated from the trace list.

Turn off the dequeuer by using the "not_dequeued" softkey in a command such as:

display trace disassemble_from_line_number 81 **not_dequeued**

With the dequeuer off, the trace list shows the content of every bus cycle, regardless of whether or not that bus cycle was part of a program execution. The bus cycles are shown in the order they occurred. That means instructions may appear in the list several bus cycles before the data transactions they cause.

When Do I Want A Dequeued Trace List?

You'll want to dequeue your trace list when you want to see the order of program execution. The dequeued trace list presents a logical display of activity. It is is easier to read and understand. You'll want to see a "not_dequeued" trace list if you need to see each bus cycle performed by the processor.

What Does A Dequeued Trace List Show?

A dequeued trace list shows a high-level view of the activity performed by the processor. It does this by performing the following tasks on the content of the trace memory:

- Align operands with the instructions that caused the operand cycles (i.e. group data transfers with the instructions that caused them to occur).
- Display operands in their logical forms (one data operand, regardless of how many bus cycles were used to transfer that operand).
- Determine whether or not conditional branches were taken, and place TAKEN or NOT TAKEN in the trace list beside each branch.
- Prevent display of unexecuted program fetch cycles.
- Display exception stack frames in logical format (condensing the information that is transferred by bus cycles when an exception is taken).

Dequeued Trace List Examples

Figures 6-1 through 6-8 show trace lists that have not been dequeued followed by trace lists of the same area of memory after they have been dequeued. These examples show how the dequeuer selects information and presents it in a dequeued trace list.

Table 6-1 shows notations used in dequeued trace lists to describe transactions.

Table 6-1. Shorthand Notation In Dequeued Trace Lists

Shorthand Notation	Definition
sdata	supervisor data
udata	user data
sprog	supervisor program
uprog	user program
dest	destination
src	source
stck	stack push or pop
addr	indirect address fetch for pre-indexed or post-indexed addressing mode

Short Jump/Branch Example

Notice in figure 6-1 that the bus cycles at address 1018H (lines 7 and 8) are shown twice following the branch on line 5. This is because the destination address of the branch had already been prefetched into the queue before the branch instruction was recognized. The processor flushes its queue when it takes a jump or branch. The unused prefetch and the fetch of the branch destination both appear in the "not_dequeued" trace list.

Figure 6-2 shows the same area of trace as figure 6-1, but shows only one occurrence of the destination address of the branch. The dequeuer recognized the unused prefetch and suppressed it so that only the activity that was actually part of the execution is shown in the trace list.

Notice the "TAKEN" notation on line 5 of the dequeued trace list. It informs you of the disposition of the BNE.B instruction.

```
Trace List                                        Mode:logical address
Label:   Address              Opcode or Status                     time count
Base:      hex                    mnemonic                          relative
trigger  00001000 LEA         $0000E000,A7                           0.52us
+0001  = 00001006 MOVEQ       #$0000000E,D1                          0.40us
+0002    00001008 MOVE.L      #$00000200,D0                          0.48us
+0003  = 0000100E CMP.L       $0000C000,D0                           0.44us
+0004    00001010   $0000C000   supr prgm long rd log addr (ds32)    0.44us
+0005    00001014 BNE.B       $00001018                              0.44us
       = 00001016 ADDQ.L      #4,D1
+0006    0000C000   $00001234   supr data long rd log addr (ds32)    0.36us
+0007    00001018 MOVE.L      D1,$0000C004                           0.36us
+0008    00001018 MOVE.L      D1,$0000C004                           0.40us
+0009  = 0000101E CMP.L       $0000C008,D1                           0.40us
+0010    0000C004   $0000000E   supr data long wr log addr (ds32)    0.40us
+0011    00001020   $0000C008   supr prgm long rd log addr (ds32)    0.32us
+0012    00001024 BNE.B       $0000102A                              0.48us
       = 00001026 MOVE.W      #$0019,D1

STATUS:   M68030--Running                  Trace complete________________........
display trace disassemble_from_line_number 0


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-1. Short Jump Example, Not_Dequeued

```
Trace List                                        Mode:logical address
Label:   Address              Opcode or Status                     time count
Base:      hex                    mnemonic                          relative
trigger  00001000 LEA         $0000E000,A7                           0.52us
+0001  = 00001006 MOVEQ       #$0000000E,D1                          0.40us
+0002    00001008 MOVE.L      #$00000200,D0                          0.48us
+0003  = 0000100E CMP.L       $0000C000,D0                           0.44us
       = 0000C000  src  sdata rd:$00001234
+0005    00001014 BNE.B       $00001018 TAKEN                        0.88us
+0008    00001018 MOVE.L      D1,$0000C004                           1.12us
       = 0000C004  dest sdata wr:$0000000E
+0009  = 0000101E CMP.L       $0000C008,D1                           0.40us
       = 0000C008  src  sdata rd:$00004000
+0012    00001024 BNE.B       $0000102A TAKEN                        1.20us
+0015  = 0000102A MOVE.L      D1,$0000C004                           1.08us
       = 0000C004  dest sdata wr:$0000000E
+0017    00001030 MOVE.B      $0000C00C,D4                           0.80us
       = 0000C00C  src  sdata rd:$04

STATUS:   M68030--Running                  Trace complete________________........
display trace disassemble_from_line_number 0 dequeued


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-2. Short Jump Example, Dequeued

Stack Push And Pop Example

A stack push and pop are shown in figures 6-3 and 6-4. The dequeued trace list shows the stack push as an operand of the JSR instruction (line 10 + 2, figure 6-4). The stack pop is shown as an operand of the RTS instruction (line 17 + 1, figure 6-4). This is the logical representation of what the processor did during execution, and it eliminates a lot of bus cycles that were used to accomplish the push and pop operations (line 10 + 1 through line 20, figure 6-3).

```
Trace List                                              Mode:logical address
Label:   Address                  Opcode or Status                          time count
Base:      hex                        mnemonic                               relative
+0010    00001020 MOVE.L          D0,-(A7)                                    0.48us
       = 00001022 JSR             $00001052
+0011    0000C000   $FFFFFFFB       supr data long rd log addr (ds32)         0.32us
+0012    00001024   $00001052       supr prgm long rd log addr (ds32)         0.36us
+0013    0000DFFC   $FFFFFFFB       supr data long wr log addr (ds32)         0.44us
+0014    00001050 MOVE.B          D6,D0                                       0.36us
       = 00001052 MOVE.L          ($0004,A7),D7
+0015    0000DFF8   $00001028       supr data long wr log addr (ds32)         0.36us
+0016  = 00001056 BPL.B           $0000105A                                   0.36us
+0017    00001058 NEG.L           D7                                          0.40us
       = 0000105A RTS
+0018    0000DFFC   $FFFFFFFB       supr data long rd log addr (ds32)         0.32us
+0019    0000105C BHI.B           $000010C0                                   0.56us
       = 0000105E BHI.B           $000010C2
+0020    0000DFF8   $00001028       supr data long rd log addr (ds32)         0.52us

STATUS:   M68030--Running                  Trace complete________________........
display trace disassemble_from_line_number 10


__run___ _trace__ __set___ __step__        display_ _modify_ __end___ ---ETC--
```

Figure 6-3. Stack Push And Pop Example, Not_Dequeued

```
Trace List                                              Mode:logical address
Label:   Address                  Opcode or Status                          time count
Base:      hex                        mnemonic                               relative
+0010    00001020 MOVE.L          D0,-(A7)                                    0.48us
       = 0000DFFC  dest sdata wr:$FFFFFFFB
       = 00001022 JSR             $00001052
       = 0000DFF8  stck sdata wr:$00001028
+0011    0000C000   $FFFFFFFB       supr data long rd log addr (ds32)         0.32us
+0014  = 00001052 MOVE.L          ($0004,A7),D7                               1.16us
       = 0000DFFC  src  sdata rd:$FFFFFFFB
+0016  = 00001056 BPL.B           $0000105A NOT TAKEN                         0.72us
+0017    00001058 NEG.L           D7                                          0.40us
       = 0000105A RTS
       = 0000DFF8  stck sdata rd:$00001028
+0021    00001028 ADDA.L          #$00000004,A7                               1.84us
+0022  = 0000102E MOVE.L          D7,$0000C000                                0.40us
       = 0000C000  dest sdata wr:$00000005
+0024    00001034 MOVE.L          $0000C004,-(A7)                             0.88us

STATUS:   M68030--Running                  Trace complete________________........
display trace disassemble_from_line_number 10


__run___ _trace__ __set___ __step__        display_ _modify_ __end___ ---ETC--
```

Figure 6-4. Stack Push And Pop Example, Dequeued

Operands From 8-Bit Memory Example

The "not_dequeued" trace shows that several bus cycles were used to form an instruction because it was contained in 8-bit memory (lines 6 through 11, figure 6-5). The operand cycles caused by execution of that instruction are shown on lines 16 and 17 of figure 6-5. The "dequeued" trace list (figure 6-6) shows the instruction, and on the next line, its operand (lines 6 and 6 + 1). A dequeued trace list shows the logical activity that was executed by the processor, regardless of the number of bus cycles required to transact that activity.

```
Trace List                                              Mode:logical address
Label:    Address                Opcode or Status                         time count
Base:       hex                      mnemonic                              relative
+0006     00004006 MOVE.W       $00004230,D0                                 0.36us
+0007     00004007   $39          supr prgm byte rd log addr (ds8)           0.36us
+0008     00004008   $00          supr prgm long rd log addr (ds8)           0.44us
+0009     00004009   $00          supr prgm 3byte rd log addr (ds8)          0.36us
+0010     0000400A   $42          supr prgm word rd log addr (ds8)           0.36us
+0011     0000400B   $30          supr prgm byte rd log addr (ds8)           0.32us
+0012     0000400C ADDI.W       #$0040,D0                                    0.48us
+0013     0000400D   $40          supr prgm 3byte rd log addr (ds8)          0.32us
+0014     0000400E   $00          supr prgm word rd log addr (ds8)           0.36us
+0015     0000400F   $40          supr prgm byte rd log addr (ds8)           0.36us
+0016     00004230   $00          supr data word rd log addr (ds8)           0.36us
+0017     00004231   $00          supr data byte rd log addr (ds8)           0.36us
+0018     00004010 CMP.W        $00004238,D0                                 0.32us
+0019     00004011   $79          supr prgm 3byte rd log addr (ds8)          0.36us
+0020     00004012   $00          supr prgm word rd log addr (ds8)           0.36us

STATUS:   M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 6


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-5. Operands From 8-Bit Memory, Not_Dequeued

```
Trace List                                              Mode:logical address
Label:    Address                Opcode or Status                         time count
Base:       hex                      mnemonic                              relative
+0006     00004006 MOVE.W       $00004230,D0                                 0.36us
       = 00004230  src  sdata rd:$0000
+0012     0000400C ADDI.W       #$0040,D0                                    2.32us
+0018     00004010 CMP.W        $00004238,D0                                 2.08us
       = 00004238  src  sdata rd:$0000
+0024     00004016 BLT.W        $00004020 ?TAKEN?                            2.20us
+0030     0000401A ADD.W        $0000423C,D0                                 2.16us
       = 0000423C  src  sdata rd:$0405
+0036     00004020 MOVE.W       D0,$00004230                                 2.40us
       = 00004230  dest sdata wr:$0445
+0044     00004026 BRA.W        $00004006                                    2.80us
+0054     00004006 MOVE.W       $00004230,D0                                 3.76us
       = 00004230  src  sdata rd:$0445
+0060     0000400C ADDI.W       #$0040,D0                                    2.20us
+0066     00004010 CMP.W        $00004238,D0                                 2.08us

STATUS:   M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 6 dequeued


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-6. Operands From 8-Bit Memory, Dequeued

Exceptions And Interrupts Example

Notice the trace lists showing exceptions and interrupts (figures 6-7 and 6-8). Exceptions and interrupts require many bus cycles to process them correctly. The dequeuer recognizes exceptions and interrupts and suppresses all of the extraneous bus-cycle information, showing just the logical flow resulting from the exception or interrupt. Compare lines 6 through 24 in figures 6-7 and 6-8.

Line 6 is the TRAP instruction. The next three lines are unexecuted prefetches. The trap handler is on line 13. The RTE is on line 16. Line 24 returns to address 1016H (1014H was fetched because it is the high word address where 1016H resides, but it is not shown in the dequeued trace list at line 24 because it is not executed at this point).

Notice the additional information supplied to clarify processor activity in the dequeued trace. An exception line such as

```
Trace List                                         Mode:logical address
Label:   Address                      Opcode or Status                          time count
Base:      hex                           mnemonic                                relative
+0006    00001014 TRAP           #0                                              0.44us
       = 00001016 ADDI.L         #$00000032,D0
+0007    0000D006   $----2222      supr data word rd log addr (ds32)             0.36us
+0008    00001018   $00000032      supr prgm long rd log addr (ds32)             0.36us
+0009    0000CFF8   $2700----      supr data word wr log addr (ds32)             0.52us
+0010    00000080   $00001060      supr data long rd log addr (ds32)             0.36us
+0011    0000CFFA   $----0000      supr data long wr log addr (ds32)             0.36us
+0012    0000CFFC   $1016----      supr data word wr log addr (ds32)             0.32us
+0013    00001060 MOVEQ          #$00000000,D1                                   0.36us
       = 00001062 MOVE.L         ($00,A7,D1.L),D0
+0014    0000CFFE   $----0080      supr data word wr log addr (ds32)             0.44us
+0015  = 00001066 MOVE.L         ($0004,A7),D0                                   0.36us
+0016  = 0000106A RTE                                                            0.56us
+0017    0000CFF8   $27000000      supr data long rd log addr (ds32)             0.36us
+0018    0000106C MOVE.L         (A7),D0                                         0.44us
       = 0000106E incomplete instr.: /202F/????/
+0019    0000CFFC   $10160080      supr data long rd log addr (ds32)             0.36us
+0020    0000CFF8   $2700----      supr data word rd log addr (ds32)             0.48us
+0021    0000CFFE   $----0080      supr data word rd log addr (ds32)             0.36us
+0022    0000CFFA   $----0000      supr data long rd log addr (ds32)             0.36us
+0023    0000CFFC   $1016----      supr data word rd log addr (ds32)             0.36us
+0024    00001014 TRAP           #0                                              0.52us

STATUS:   M68030--Running                  Trace complete________________........
display trace disassemble_from_line_number 0


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-7. Exceptions And Interrupts, Not_Dequeued

"**EXCEPTION** TRAP #0" will be included in a dequeued trace list to describe the type of exception detected. Additional information relevant to the exception will be shown following the "**EXCEPTION** line. The notations in these information lines are defined in table 6-2.

Table 6-2. Notations Following Exceptions Defined

Notation	Definitions
PSW	processor status word
PC	program counter
FMT	format word (format of transfer status register)
VEC	exception vector fetch

```
Trace List                                          Mode:logical address
Label:    Address                   Opcode or Status                          time count
Base:       hex                         mnemonic                               relative
trigger   00001000 LEA           $0000D000,A7                                   0.56us
+0001   = 00001006 NOP                                                          0.40us
+0002     00001008 MOVE.W        $0000D004,D0                                   0.44us
        = 0000D004  src  sdata rd:$1111
+0003   = 0000100E MOVE.W        $0000D006,D1                                   0.44us
        = 0000D006  src  sdata rd:$2222
+0006     00001014 TRAP          #0                                             1.20us
+0009              **EXCEPTION** TRAP #0                                        1.24us
          0000CFF8  PSW=$2700 PC=$00001016 FMT=$0080
        = 00000080  VEC=$00001060
+0013     00001060 MOVEQ         #$00000000,D1                                  1.40us
        = 00001062 MOVE.L        ($00,A7,D1.L),D0
        = 0000CFF8  src  sdata rd:$27000000
+0015   = 00001066 MOVE.L        ($0004,A7),D0                                  0.80us
        = 0000CFFC  src  sdata rd:$10160080
+0016   = 0000106A RTE                                                          0.56us
+0020     0000CFF8  PSW=$2700 PC=$00001016 FMT=$0080                            1.64us
+0024   = 00001016 ADDI.L        #$00000032,D0                                  1.60us
+0026     0000101C ADDI.L        #$00000032,D1                                  0.80us

STATUS:    M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 0 dequeued


__run___ _trace__ __set___ __step__        display_ _modify_ __end___ ---ETC--
```

Figure 6-8. Exceptions And Interrupts, Dequeued

Problems You May See When Using The Dequeuer

The following paragraphs discuss problems you may see in dequeued trace lists. Each paragraph shows you how to recognize the problem, and how to correct it in your trace list.

Bus Cycles In The Trace List

When you see a line with bus-cycle width identifiers (ds8), (ds16), or (ds32), you are seeing bus cycles instead of logical, dequeued information. When bus cycles appear in the dequeued trace list, it usually means the dequeuer is not aligning operands and instructions correctly. To obtain correct alignment of operands and instructions, issue a new "disassemble_from" command and use the "align_data_from_line" token to re-synchronize inverse assembly, as described in the following command:

display trace disassemble_from_line_number <instructionline> **dequeued align_data_from_line** <dataline>

Where:
<instructionline> = the number of the line containing the instruction with the incorrect operand.
<dataline> = the number of the line containing the correct operand.

Note

You can use as many new "disassemble_from" and "align_data" commands as you like. The inverse assembler will remember all of the specified points and correct the trace list accordingly.

Perhaps you are looking at bus cycles that resulted from instructions that were executed before the location you specified in your earliest "disassemble_from" command. Dequeuing begins with the line number you specified and proceeds forward from there, but never backward.

Bus cycles may appear in the first few lines after the beginning of the inverse assembled portion of your trace list. These are usually

```
Trace List                                              Mode:logical address
Label:   Address                       Opcode or Status                          time count
Base:      hex                             mnemonic                               relative
+0009      0000101C MOVE.B       ($0010,A0),($FFFE,A1)                             0.32us
       = 00002000  src  sdata rd:$00
       = 00002002  dest sdata wr:$00
+0012      00002000   $00------      supr data byte rd log addr (ds32)              1.08us
+0013      00002002   $----00--      supr data byte wr log addr (ds32)              0.32us
+0014  = 00001022 MOVE.B       ($01,A0,D3.W),($FF,A1,D3.L)                          0.36us
       = 00002002  src  sdata rd:$00
       = 00002002  dest sdata wr:$00
+0015      00002010   $00------      supr data byte rd log addr (ds32)              0.44us
+0017      00002000   $00------      supr data byte wr log addr (ds32)              0.72us
+0019      00001028 MOVE.B       ($01,A0,A3.W*4),($FF,A1,A3.L*2)                    0.68us
       = 00002005  src  sdata rd:$02
       = 00002003  dest sdata wr:$02
+0021  = 0000102E MOVE.B       $00002008,$00002018                                  0.80us
       = 00002008  src  sdata rd:$11

STATUS:    M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 9 dequeued


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-9. Problem Of Bus Cycles In Trace List

```
Trace List                                              Mode:logical address
Label:   Address                       Opcode or Status                          time count
Base:      hex                             mnemonic                               relative
+0009      0000101C MOVE.B       ($0010,A0),($FFFE,A1)                             0.32us
       = 00002010  src  sdata rd:$00
       = 00002000  dest sdata wr:$00
+0014  = 00001022 MOVE.B       ($01,A0,D3.W),($FF,A1,D3.L)                          1.76us
       = 00002002  src  sdata rd:$00
       = 00002002  dest sdata wr:$00
+0019      00001028 MOVE.B       ($01,A0,A3.W*4),($FF,A1,A3.L*2)                    1.84us
       = 00002005  src  sdata rd:$02
       = 00002003  dest sdata wr:$02
+0021  = 0000102E MOVE.B       $00002008,$00002018                                  0.80us
       = 00002008  src  sdata rd:$11
       = 00002018  dest sdata wr:$11
+0026      00001034 MOVE.B       $00002008,$00002018                                1.88us
       = 00002008  src  sdata rd:$11
       = 00002018  dest sdata wr:$11

STATUS:    M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 9 dequeued align_data_from_line 15


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-10. Example Of Bus Cycles Fixed In Trace List

operand cycles from instructions that occurred before the line specified in your "disassemble_from" command.

Figure 6-9 is a dequeued trace list that shows some bus cycles, indicating a dequeuer error. The designer was able to recognize that the operand on line 15 belongs with the instruction on line 9. Lines 9+1 through 13 are operand cycles caused by instructions that occurred before line 9. By using the command "display trace disassemble_from_line_number 9 dequeued align_data_from_line 15", dequeuing is corrected in the inverse assembled trace list (figure 6-10).

Jumps (JMP and JSR)

The trace list may show disassembly beginning one word before the actual destination of a jump, or you may see meaningless disassembly following a jump. You may also see unaligned cycles in the trace list following a jump.

In cases where the destination address of a jump is stated clearly in the code, disassembly will resume at the correct address after the jump. If the destination address of a jump is not stated clearly in the code (such as in register-indirect addressing mode), disassembly will begin on the even address in the long word at the destination of the jump. If the jump destination is an even word, disassembly will be correct. If the jump destination is an odd word, disassembly will not be correct.

Try disassembling the trace from a new point. Use the "low_word" token if the activity is obtained from 32-bit memory. Use a new trace list line number if the activity is from 16-bit or 8-bit memory.

Figure 6-11 shows a jump instruction on line 1. The destination of the jump is on line 3 + 1. The state on line 3 is the high word at the 32-bit address where the jump destination resides. By using the "low_word" token, this error is corrected in figure 6-12. The "low_word" token could be used in a new command to correct the error on line 16. The inverse assembler would remember both corrected locations and produce a correct trace list.

Notice that the dequeued trace list shows the values of D0, D1, and D2 on one line following the MOVEM.L instruction.

```
Trace List                                                Mode:logical address
Label:   Address                       Opcode or Status                        time count
Base:      hex                             mnemonic                              relative
trigger  00001000 LEA            $0000101A,A0                                    0.52us
+0001  = 00001006 JMP            (A0)                                            0.40us
+0003    00001018 Illegal Coprocessor Instruction: $FFFF                         1.00us
       = 0000101A ADDI.L         #$00000032,D0
+0005    00001020 ADDI.L         #$00000032,D0                                   0.80us
+0006  = 00001026 MOVEM.L        D0-D2,$0000C000                                 0.48us
       = 0000C000  D0=$00000064 D1=$00000000 D2=$00000000
+0008  = 0000102E JMP            $00001000                                       0.88us
+0013    00001000 LEA            $0000101A,A0                                    1.84us
+0014  = 00001006 JMP            (A0)                                            0.40us
+0016    00001018 Illegal Coprocessor Instruction: $FFFF                         1.00us
       = 0000101A ADDI.L         #$00000032,D0
+0018    00001020 ADDI.L         #$00000032,D0                                   0.80us
+0019  = 00001026 MOVEM.L        D0-D2,$0000C000                                 0.48us
       = 0000C000  D0=$000000C8 D1=$00000000 D2=$00000000

STATUS:   M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 0 dequeued


__run___ _trace__ __set___ __step__        display_ _modify_ __end___ ---ETC--
```

Figure 6-11. Example Of Uncertain Jumps In Trace List

```
Trace List                                                Mode:logical address
Label:   Address                       Opcode or Status                        time count
Base:      hex                             mnemonic                              relative
trigger  00001000 LEA            $0000101A,A0                                    0.52us
+0001  = 00001006 JMP            (A0)                                            0.40us
+0003  = 0000101A ADDI.L         #$00000032,D0                                   0.52us
+0005    00001020 ADDI.L         #$00000032,D0                                   0.80us
+0006  = 00001026 MOVEM.L        D0-D2,$0000C000                                 0.48us
       = 0000C000  D0=$00000064 D1=$00000000 D2=$00000000
+0008  = 0000102E JMP            $00001000                                       0.88us
+0013    00001000 LEA            $0000101A,A0                                    1.84us
+0014  = 00001006 JMP            (A0)                                            0.40us
+0016    00001018 Illegal Coprocessor Instruction: $FFFF                         1.00us
       = 0000101A ADDI.L         #$00000032,D0
+0018    00001020 ADDI.L         #$00000032,D0                                   0.80us
+0019  = 00001026 MOVEM.L        D0-D2,$0000C000                                 0.48us
       = 0000C000  D0=$000000C8 D1=$00000000 D2=$00000000
+0021  = 0000102E JMP            $00001000                                       0.88us

STATUS:   M68030--Running                 Trace complete________________........
display trace disassemble_from_line_number 3 low_word dequeued


__run___ _trace__ __set___ __step__        display_ _modify_ __end___ ---ETC--
```

Figure 6-12. Uncertain Jumps Fixed In Trace List

Branch - Was It Taken Or Not?

The trace list in figure 6-13 shows a branch and ?TAKEN? on line 5. The notation "?TAKEN?" indicates the dequeuer couldn't determine whether or not the branch had been taken.

Usually the dequeuer can tell whether or not a branch was taken by looking for a change in the sequence of addresses in the execution that follows the branch. In these cases, the dequeuer will place "TAKEN" or "NOT TAKEN" in the trace list beside the branch instruction, and it will dequeue the trace list that follows the branch.

In figure 6-13, the dequeuer could not determine whether the branch on line 5 was taken or not because the same address sequence would be fetched whether or not the branch was taken.

When a short, forward branch is taken (branch to an address that is only a few addresses higher than the address of the branch instruction), the dequeuer may not be able to tell if the branch was taken. When the dequeuer is unsure whether or not a branch was taken, it assumes that the branch was not taken. It also assumes the instructions immediately following the branch were executed after falling through the branch. This produces a correct trace list if the branch was not taken.

If the branch was taken, the dequeuer may produce an incorrect trace list. After making the wrong assumption about the branch, the dequeuer will try to find memory operands for instructions that were never executed. This may produce a trace list in which instructions are aligned with the wrong operands. Bus cycles may also appear in the list. Sometimes the bus cycles will not appear for several instructions after the branch marked ?TAKEN?.

To determine whether or not a branch marked ?TAKEN? was actually taken, see if there are memory operands in the trace list that indicate the adjacent fetches were executed. Look for bus cycles in the trace list. You may need to look at the associated "not_dequeued" trace list (figure 6-14) to determine the values of the condition codes used by the branch instruction.

Correcting Branches ?TAKEN? In Trace List

If you find that a branch marked ?TAKEN? was actually taken, you can correct your trace list as follows:

1. Find the trace list line number containing the address of the branch destination. For 32-bit wide memory, determine if the high word or low word contains the branch destination.

2. Issue a "display trace disassemble_from..." command to restart inverse assembly at the address of the branch destination. If necessary, include an "align_data_from_line" token to resynchronize the instructions and data in your trace list.

```
Trace List                                          Mode:logical address
Label:   Address                   Opcode or Status                      time count
Base:      hex                         mnemonic                           relative
trigger  00001000 LEA           $0000E000,A7                               0.52us
+0001  = 00001006 MOVEQ         #$0000000E,D1                              0.40us
+0002    00001008 MOVE.L        #$00000200,D0                              0.48us
+0003  = 0000100E CMP.L         $0000C000,D0                               0.44us
       = 0000C000  src  sdata rd:$00001234
+0005    00001014 BNE.B         $0000101E ?TAKEN?                          0.88us
       = 00001016 MOVE.L        #$00005678,D2
+0008    0000101C ADDQ.L        #4,D1                                      1.12us
       = 0000101E MOVE.L        D1,$0000C004
       = 0000C004  dest sdata wr:$0000000E
+0010    00001024 CMP.L         $0000C008,D1                               0.80us
       = 0000C008  src  sdata rd:$00004000
+0012  = 0000102A BNE.B         $00001030 ?TAKEN?                          0.68us
+0014    0000102C MOVE.W        #$0019,D1                                  0.76us
+0015    00001030 MOVE.L        D1,$0000C004                               0.56us

STATUS:   M68030--Running                Trace complete________________........
display trace disassemble_from_line_number 0 dequeued


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-13. Example Branches ?Taken? In Trace List

```
Trace List                                      Mode:logical address
Label:   Address                Opcode or Status                    time count
Base:     hex                      mnemonic                          relative
trigger  00001000 LEA         $0000E000,A7                            0.52us
+0001  = 00001006 MOVEQ       #$0000000E,D1                           0.40us
+0002    00001008 MOVE.L      #$00000200,D0                           0.48us
+0003  = 0000100E CMP.L       $0000C000,D0                            0.44us
+0004    00001010   $0000C000   supr prgm long rd log addr (ds32)     0.44us
+0005    00001014 BNE.B       $0000101E                               0.44us
       = 00001016 MOVE.L      #$00005678,D2
+0006    0000C000   $00001234   supr data long rd log addr (ds32)     0.36us
+0007    00001018   $00005678   supr prgm long rd log addr (ds32)     0.36us
+0008    0000101C ADDQ.L      #4,D1                                   0.40us
       = 0000101E MOVE.L      D1,$0000C004
+0009    00001020   $0000C004   supr prgm long rd log addr (ds32)     0.40us
+0010    00001024 CMP.L       $0000C008,D1                            0.40us
+0011    0000C004   $0000000E   supr data long wr log addr (ds32)     0.32us
+0012  = 0000102A BNE.B       $00001030                               0.36us

STATUS:   M68030--Running               Trace complete________________........
display trace disassemble_from_line_number 0 not_dequeued


__run___ _trace__ __set___ __step__       display_ _modify_ __end___ ---ETC--
```

Figure 6-14. Not-Dequeued Trace To Solve Branch ?Taken?

Analyzer Commands

This chapter contains pages that supplement the syntax pages in Chapter 15 of the 16- And 32-Bit Internal Analysis Reference Manual. Some of the pages in this chapter are new pages, and some are replacement pages for corresponding syntax pages. You can leave the syntax pages of this chapter in this manual so that you can compare the differences between this analyzer and the analyzer described in the reference manual, or you can take the pages out of this chapter and use them to replace the corresponding pages in the reference manual.

New Pages

The copy MMU and display MMU pages in this chapter are new. They show the new command structures required to support this analyzer's ability to display MMU address mappings.

Replacement Pages

The **display trace** pages in this chapter are different from the corresponding syntax pages in the reference manual. The only differences in the **display trace** command syntax are caused by the addition of command tokens that are needed for trace list dequeuing.

Notes

copy MMU

Syntax

The syntax of the **copy mmu_tables** and **copy mmu_mappings** commands are shown in two diagrams in this section.

Function

The **copy mmu_mappings** and **copy mmu_tables** commands allow you to obtain copies of the present MMU mappings under a selected root pointer, and copies of the path that maps any selected logical address to its physical address. The **copy mmu_mappings** and **copy mmu_tables** commands can send electronic copies to be produced on a printer, stored in an HP-UX file, or processed by an HP-UX command of your choice.

Note

The **mmu_mappings** and **mmu_tables** features only work when you are using the background monitor of your emulator. The memory accesses that are required to support these features are not implemented in the foreground monitor.

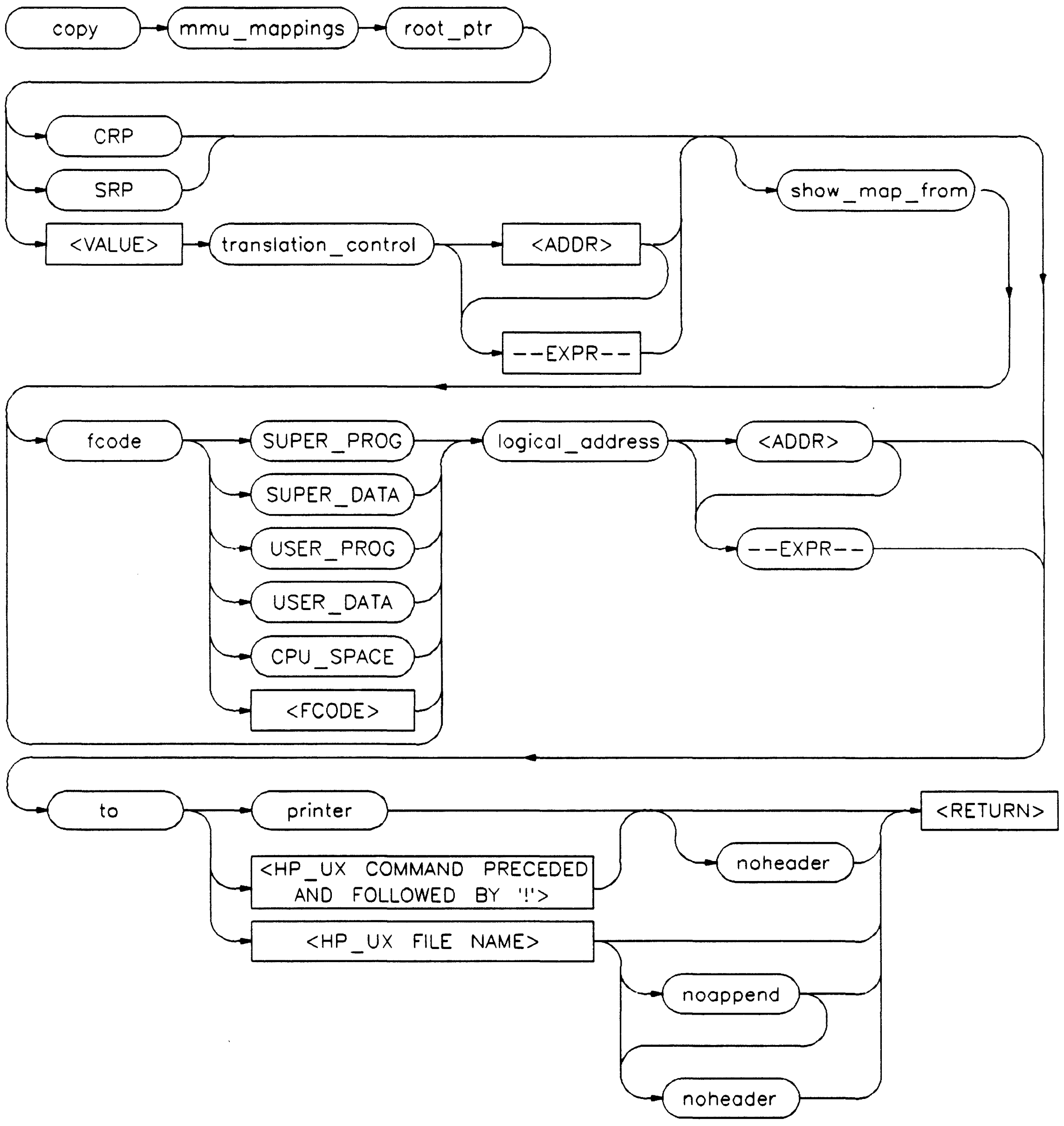
Examples

copy mmu_tables root_ptr CRP logical address 0 **to** mytables

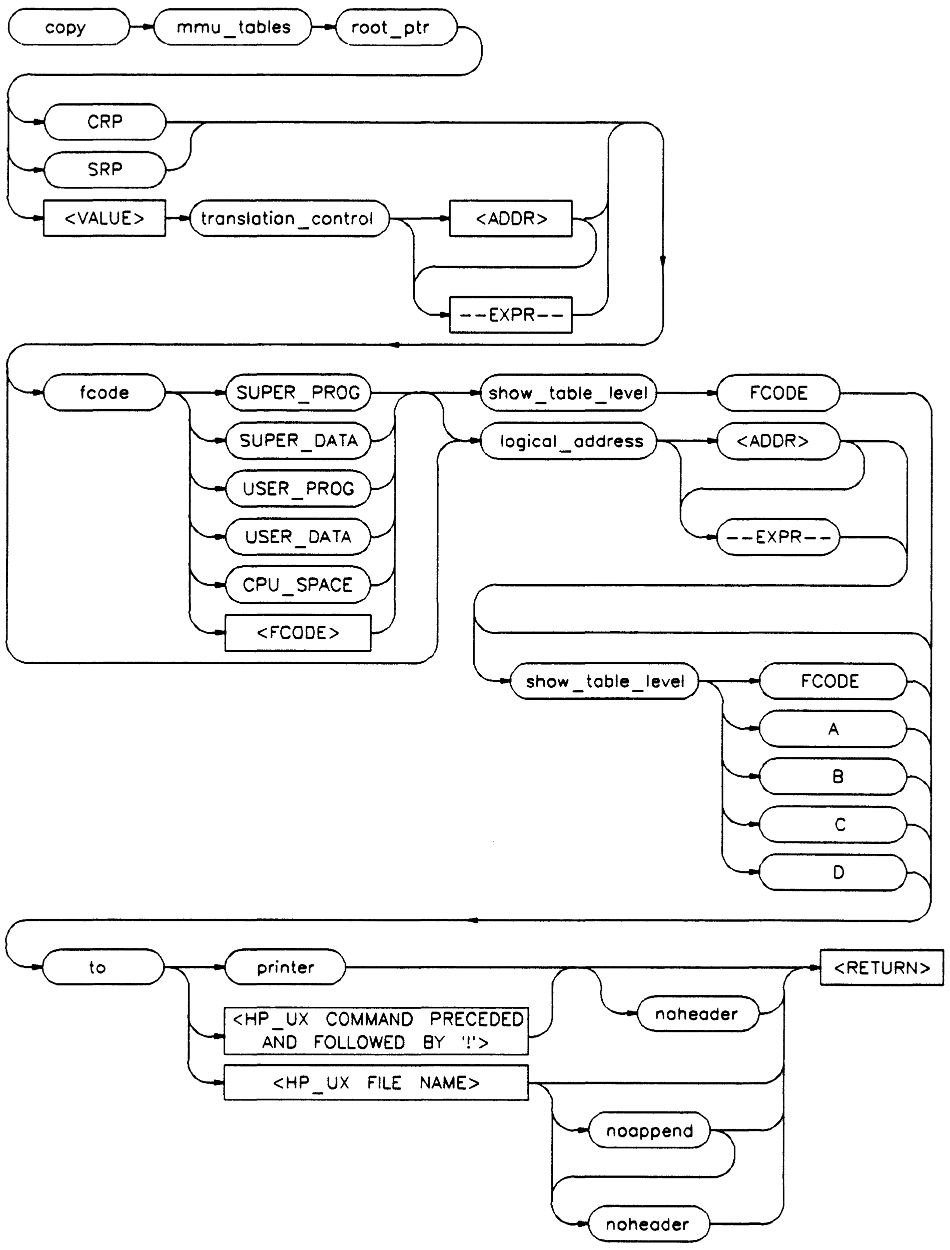
copy mmu_tables root_ptr CRP fcode USER_DATA logical_address 02000000H **show_table_level FCODE to printer noheader**

copy mmu_tables root_ptr 011B **translation_control** 82CF5000H **logical_address** 02000000H **show_table_level B**

copy mmu_mappings



copy mmu_tables



Parameters

<ADDR>	This prompts you to enter an address expression. Refer to the a/d/s EXPRESSION syntax diagram in the 16- and 32-Bit Internal Analysis manual for HP 64400-Series analyzers for details.
CRP	CPU Root Pointer.
fcode	Use this to specify which function code you want to begin your mmu_mappings list, or mmu_tables display.
logical_address	The address in logical (virtual) memory space.
mmu_mappings	The list of mappings that show the logical address ranges and their corresponding physical address ranges.
mmu_tables	The path taken through the tables to show how a selected logical address is mapped to its corresponding physical address, or by adding "show_table_level", the details of a table's content within a narrow address range.
noappend	Use this to overwrite an existing file. If you copy to a file, the default routine will append your information to the existing file instead of overwriting the file.
noheader	This allows you to turn off storage of the header information in your copy to save space.
printer	Use this to direct your copy to be made on the system printer.
root_ptr	Use this to introduce the source of the root pointer descriptor.

show_map_from	Use this to specify the logical address (with or without function code) where you want your mapping list or tables display to begin.
show_table_ level	Use this to specify the table level that you want to examine in detail in your mmu_tables copy.
SRP	Supervisor Root Pointer
to	This allows you to specify whether your copy is to be sent to the system printer, to an HP-UX file, or be included as part of an HP-UX command.
translation_ control	Use this to specify a value for the translation control register. This is required when you specify your own value for the root pointer.
<VALUE>	Root pointer value to be used instead of the CRP or SRP. You must also specify the value of the translation control register when you specify your own root pointer value.

Notes

display MMU

Syntax

The syntax of the **display mmu_mappings** and **display mmu_tables** commands are shown on two diagrams in this section.

Function

The **display mmu_mappings** command calls up a display that shows all of the ranges of logical addresses that are presently mapped to physical addresses by the MMU. If the first level of your adddress mappings uses function codes, the **mmu_mappings** display will be separated into blocks of addresses under function-code headings.

The **display mmu_tables** command calls up a display that shows the path used to map one logical address to its corresponding location in physical memory. You specify the logical address and your display will show you the corresponding mapping tables. The **mmu_tables** display can be used to troubleshoot mapping problems that appear in **display mmu_mappings** lists (identified by "INVALID" notations in the lists).

Note

The **mmu_mappings** and **mmu_tables** features only work when you are using the background monitor of your emulator. The memory accesses that are required to support these features are not implemented in the foreground monitor.

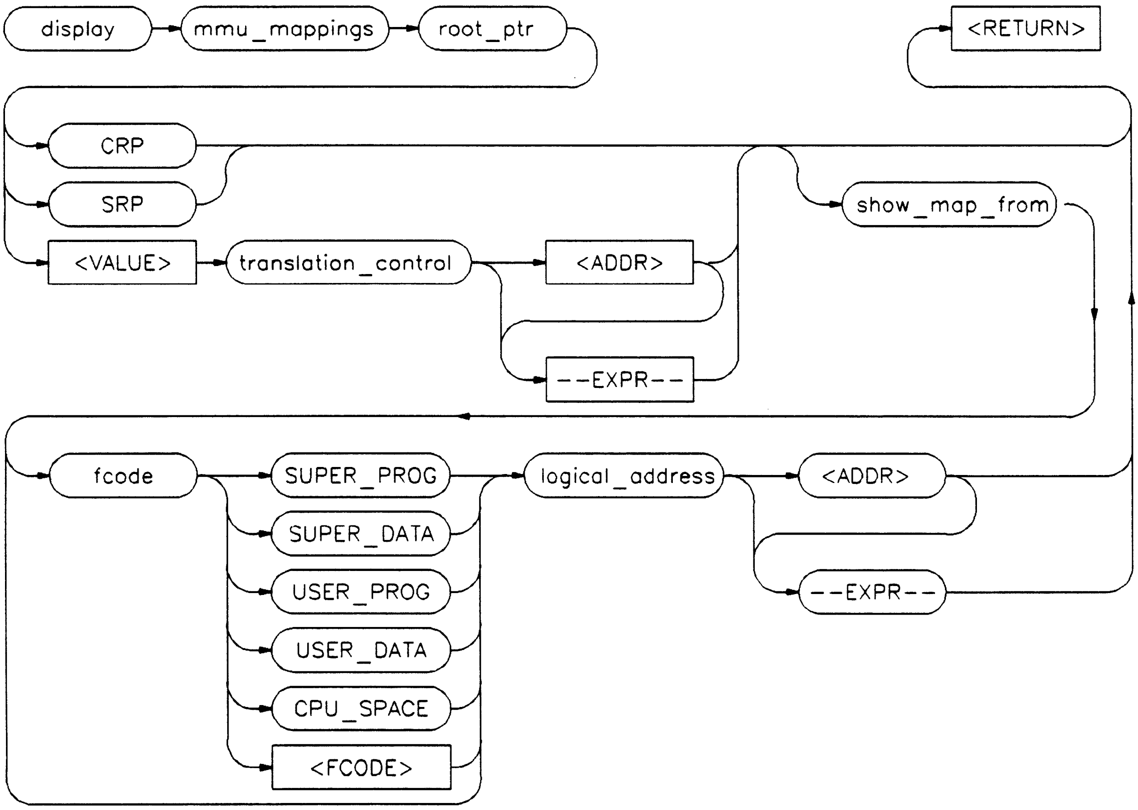
Examples

display mmu_tables root_ptr CRP logical_address 0

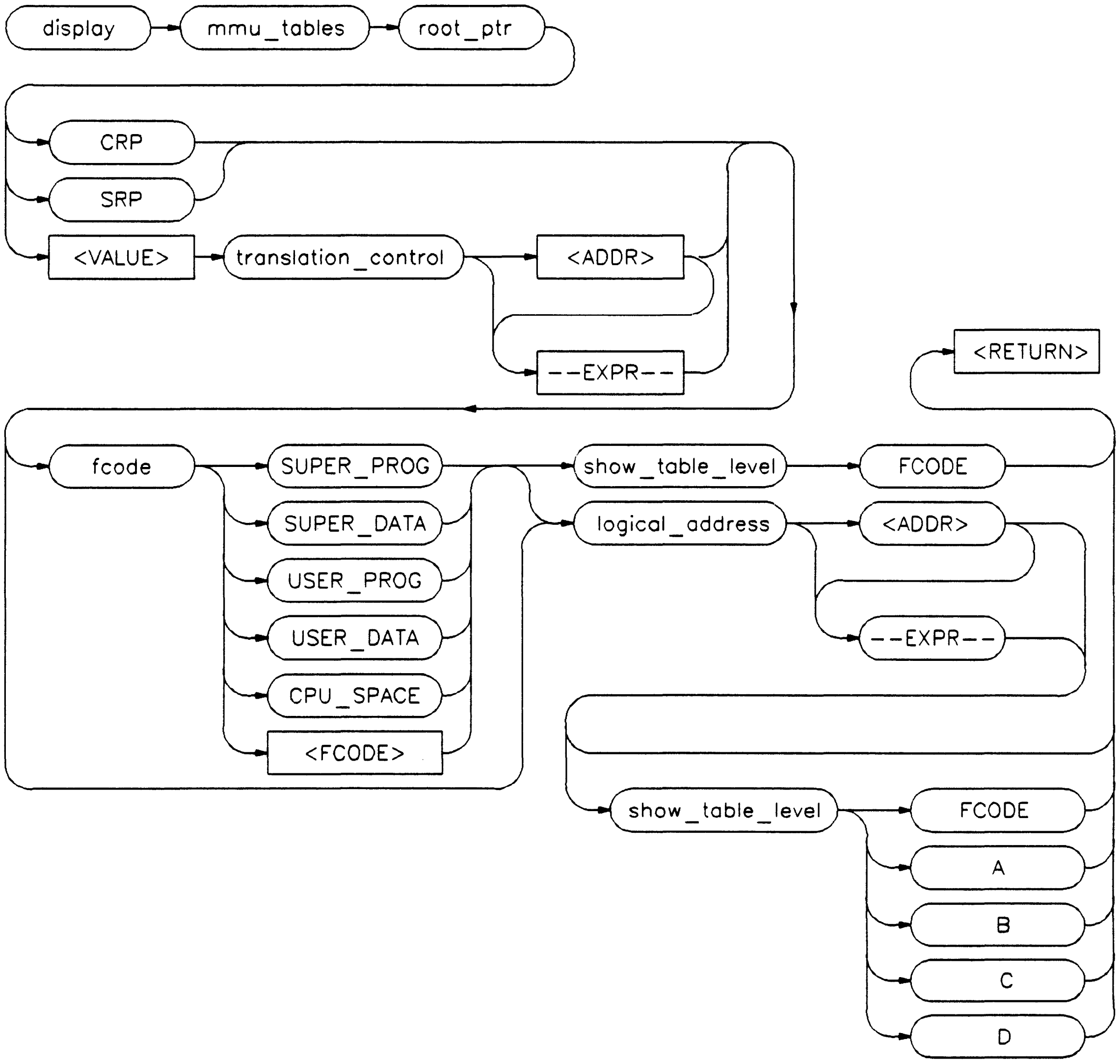
display mmu_tables SRP root_ptr fcode SUPER PROG logical_address 0A1000H

display mmu_mappings root_ptr CRP show_map_from fcode USER_DATA logical address 2000H

display mmu_mappings



display mmu_tables



Parameters

<ADDR>	This prompts you to enter an address expression. Refer to the a/d/s Expression syntax diagram in the 16- and 32-Bit Internal Analysis manual for HP 64400-Series analyzers for details.
CPU_SPACE	Identifies function code value 7H.
CRP	CPU Root Pointer.
fcode	Use this to specify which function code you want to begin your mmu_mappings list, or mmu_tables display.
<FCODE>	This prompts you to press any softkey that has a function-code identifier, or enter a Motorola-reserved function code value, if desired.
logical_address	The address in logical (virtual) memory space.
mmu_mappings	The list of mappings that show the logical address ranges and their corresponding physical address ranges.
mmu_tables	The path taken through the tables to show how a selected logical address is mapped to its corresponding physical address, or by adding "show_table_level", the details of a table's content within a narrow address range.
root_ptr	Use this to introduce the source of the root pointer descriptor.
show_map_from	Use this to specify the logical address (with or without function code) where you want your mapping list or tables display to begin.
show_table_ level	Use this to specify the table level that you want to examine in detail in your mmu_tables display.

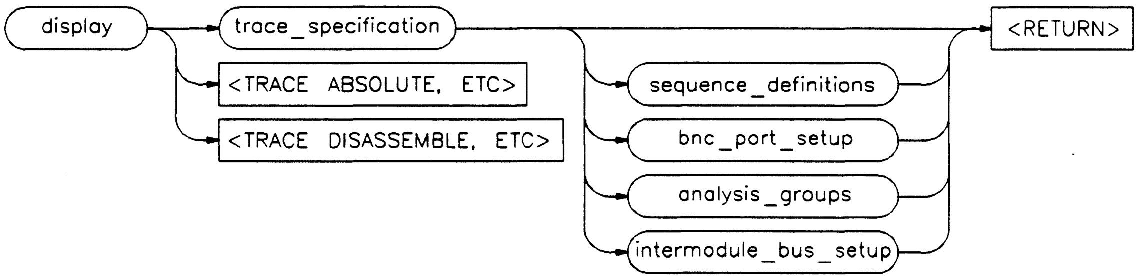
SRP	Supervisor Root Pointer.
SUPER_DATA	Identifies function code value 5H.
SUPER_PROG	Identifies function code value 6H.
translation_ control	Use this to specify the value for the translation control register. This is required when you specify your own value for the root pointer.
USER_DATA	Identifies function code value 1H.
USER_PROG	Identifies function code value 2H.

Notes

display trace

Syntax The syntax of the **display trace** command is shown in three diagrams in this section. The first diagram shows the overall syntax. The next two diagrams show details of the <TRACE ABSOLUTE> and <TRACE DISASSEMBLE> breakdowns.

display trace <TRACE_SPECIFICATION>

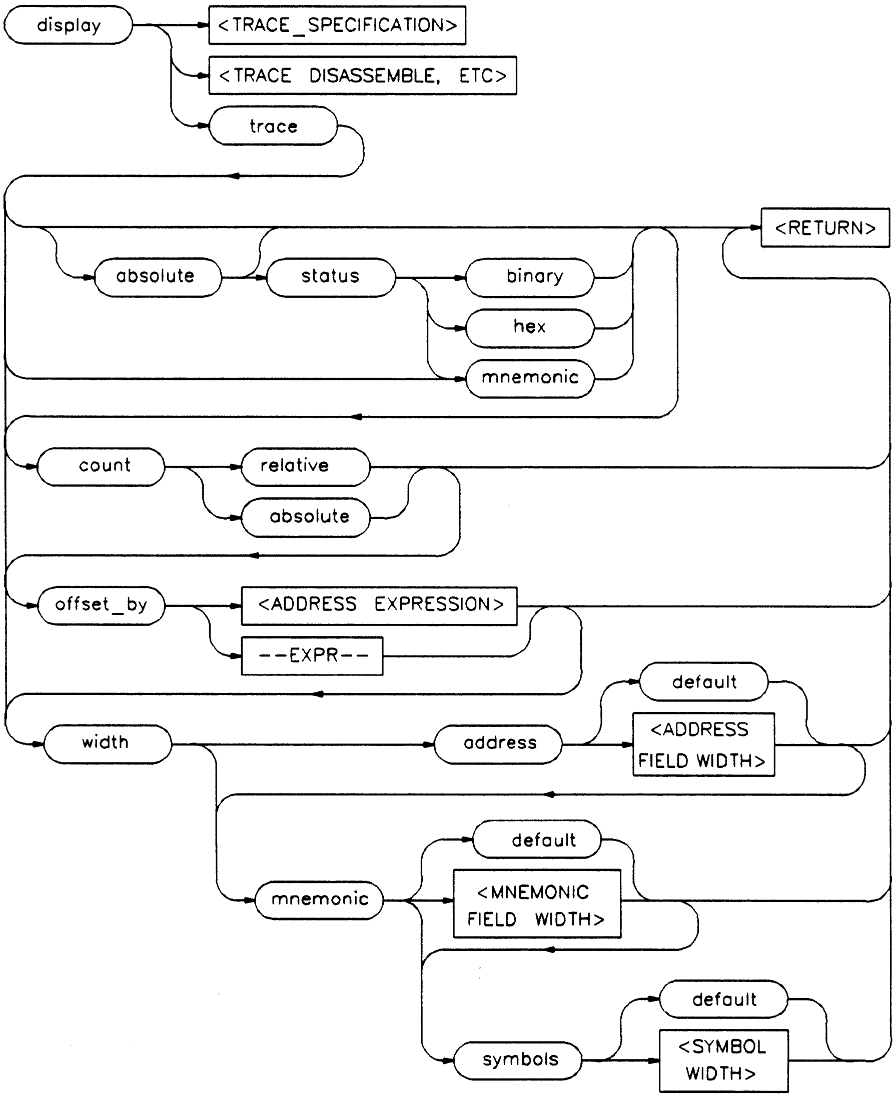


Function The **display trace** command is used to obtain a trace list showing information in the form desired (assembly language, selected column widths, etc). The **display trace_specification** command shows the present setup of the analyzer and associated functions on screen.

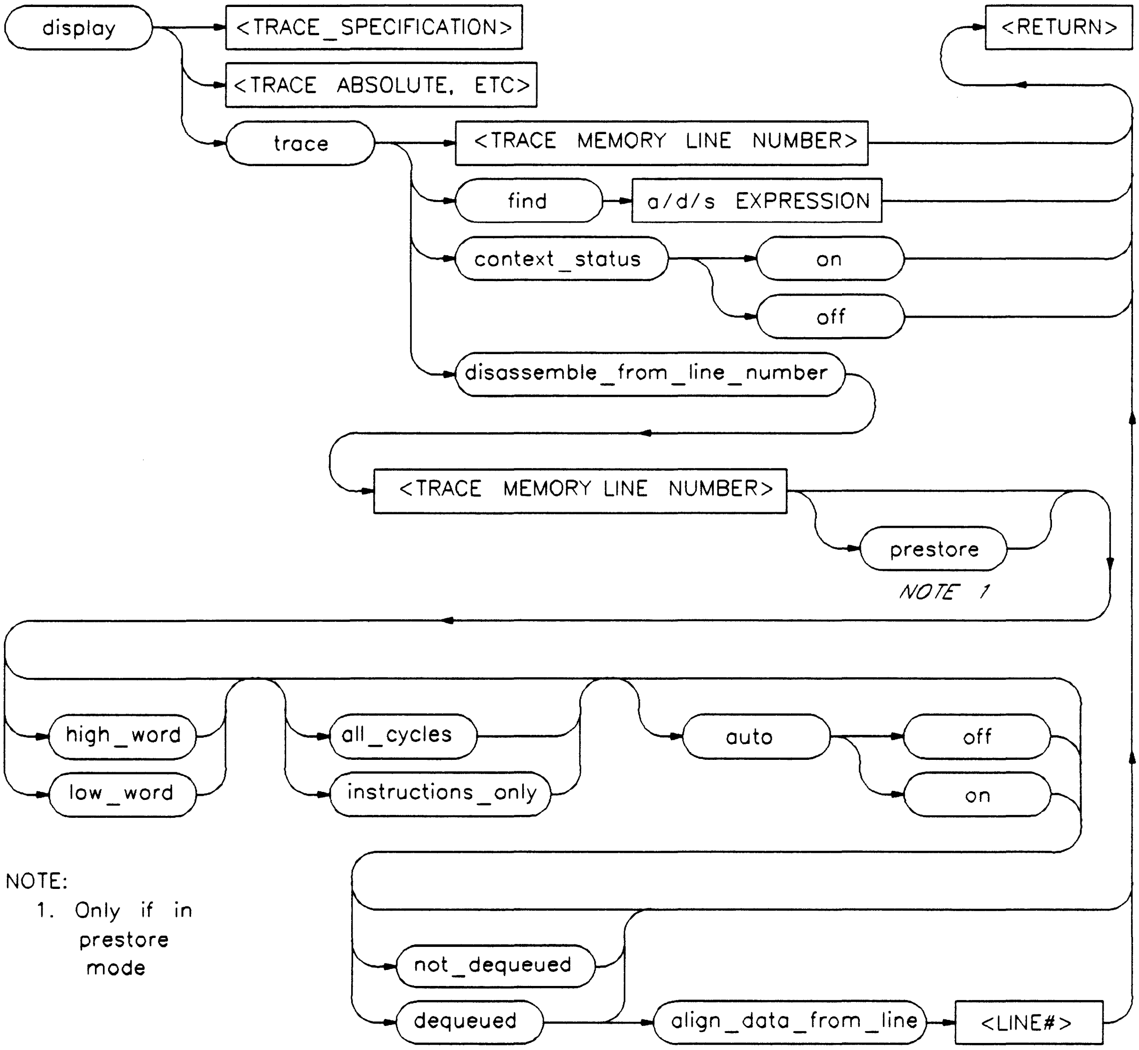
Examples **display trace disassemble_from_line_number 1**

display trace symbols on width address 12

<TRACE ABSOLUTE>


display
<TRACE_SPECIFICATION>
<TRACE DISASSEMBLE, ETC>
trace
<RETURN>
absolute
status
binary
hex
mnemonic
count
relative
absolute
offset_by
<ADDRESS EXPRESSION>
--EXPR--
default
width
address
<ADDRESS FIELD WIDTH>
default
mnemonic
<MNEMONIC FIELD WIDTH>
default
symbols
<SYMBOL WIDTH>

<TRACE DISASSEMBLE>



NOTE:
1. Only if in prestore mode

Parameters

absolute	This allows you to obtain a display of the present trace memory content in absolute numbers.
address	Used to set the width of the address column displayed in the trace list.
align_data_from_ line	Use this to correct data-alignment problems if you see any in a dequeued trace list. If you see that the dequeuer has aligned data with the wrong instructions, use this token to select the correct data alignment by specifying the line that should begin a data realignment (align_data_from_line 36). Refer to Chapter 6 for further information.
all_cycles	Used to specify that all cycles should be included in the inverse-assembled information shown in the trace list.
analysis_ groups	This allows you to move the display window to the first part of the trace_specification where the trigger, store, count, context, and trigger_position specifications are shown.
auto	This allows you to obtain automatic inverse assembly from the same point in your trace list after each new trace execution. Inverse assembly always begins with the line at the top of the screen. If you made a trace and used "disassemble_from_line_number -0006", line -0006 would be at the top of your screen. With "auto on", each time you execute a new trace, the data will be inverse assembled, beginning with line -0006 at the top of the screen. There is a risk associated with this feature. If you leave "auto on", and roll the display to place a different line at the top of the screen (line +0002, for example),

and then trace again, disassembly will begin with line +0002, instead of -0006 at the top of the screen. The inverse assembler assumes that the first word it encounters is an opcode, and it begins inverse assembly on that basis. If line +0002 does not begin with an opcode, you'll get invalid inverse assembly. If you are making a series of traces and rolling the display from one point to another between the traces, you may want to select "auto off".

binary	Selects display of the states of the individual status bits.
bnc_port_ setup	This allows you to move the display window to the portion of the trace_specification that shows the present setup of the BNC ports.
context_status	Only available after you have made a trace in either the "one_analysis_group" or "three_analysis_groups" softkey interface. Used to turn on or turn off the display of context_status lines in your copy of the trace list.
count	This allows you to obtain a count in the "time count" column that shows time measurements either "relative" to the time since the most recent time stamp, or "absolute" time since the capture of the trigger state.
default	Used to set the display width of the associated parameter to its default width.

dequeued	Use this to obtain a trace list showing the activity that the 68030 processed during the trace. Unused prefetches are eliminated from this display, and data transactions are aligned with the instructions that caused them to occur.
disassemble_ from_line_ number	Use this to set the inverse-assembler to begin operation on the opcode located at the specified line number. You can select high_word or low_word to begin inverse-assembly at the desired address within a long word. You can have inverse-assembly include information from all_cycles, or limit information to a list of the opcodes by selecting instructions_only.
--EXPR--	This parameter is shown under the a/d/s EXPRESSION syntax diagram.
find	This feature is used to find occurrences of any event you specify. For example, you can find all occurrences of address 1000h in the trace list, if desired.
hex	Use this to obtain a copy of status information expressed in hexadecimal values of the status bits.
high_word	Use this to have inverse-assembly begin with the opcode in the high word of the long word located in the specified trace-memory line number.

intermodule_ bus_setup	This allows you to obtain a display showing the present setup of the signals of the intermodule bus. This is only available when using the analyzer in systems composed of two or more modules.
instructions_ only	This causes the trace list to contain only those lines that show an instruction opcode.
<LINE_#>	This prompts you to enter a trace memory line number where you want your trace list display to begin or where you want your dequeuer to begin alignment of data (data alignment can never begin on a line earlier than the line specified to begin inverse assembly).
low_word	This causes inverse-assembly to begin with the opcode stored in the low word of the long word at the specified trace memory line number.
mnemonic	This allows you to obtain a display of the trace memory content with the data information inverse-assembled.
not_dequeued	Use this entry to select an inverse-assembled trace listing in the order that information appeared on the external processor buses during the trace. This listing includes all transactions (unused prefetches, etc).
off	Used to turn off display of the associated parameter.

offset_by	This allows you to offset all address expressions by any --EXPR-- selection, or by the address where a source-file symbol was stored. For example, you can offset the addresses by the starting address of a selected function.
on	Used to turn on display of the associated parameter.
only	Used to obtain a trace-list display of a "source only" trace list.
prestore	Used to begin inverse assembly with the prestored state associated with a particular trace-memory line number.
relative	This is the default selection. In time count, it shows elapsed time since capture of the preceding state in the trace memory. In state count, it shows number of qualified states counted since the last count value was listed.
sequence_ definitions	This allows you to position the display window in the trace_specification to begin with the location that shows the present setup of the sequence.
status	This allows you to have your trace display show status information in binary values or hexadecimal values of the status bits, or in mnemonic descriptions of the meanings of the bits.
symbols	This allows you to specify a selected portion of display width for showing the names of the symbols in the trace list.

tabs_are	This allows you to select any desired number of spaces to be represented by each tab when showing lines from a source file. Use this selection to aid readability of the source file information.
trace	This allows you to obtain a display of the present content of the trace memory. With no parameters selected, you will get a display composed according to your most recent **display trace** command. You can add parameters to obtain any desired format in your display.
trace_ specification	This allows you to obtain a display of the present trace specification setup. You can position the display window to show the analysis_groups, sequence_definitions, or the bnc_port_setup, or the intermodule_bus_setup specification, as desired.
width	This allows you to control the amount of display space allocated to the "address column", the "mnemonic column", and to the symbols shown in either column in the trace list. Use this key to get more columns on display, or to allow more information to be shown in a particular column in your trace list. The default softkey allows you to return the associated column to the normal width allocation.

Notes

Index

= sign in trace list, **1-5**

A

access, **3-4**
access to variables in prestore traces, **2-2**
activity dequeued in trace list, **6-1**
addr (in dequeued trace lists), **6-4**
addr_mode selections in status specification, **3-4**
address equivalent in trace list, **1-5**
address symbols, how the MMU affects these, **4-10**
address value preceded by letter "p", **4-22**
address with prepended "p" in trace list, **5-12**
address, how logical is mapped to physical, **4-4**
"align_data_from_line" token, how to use, **6-12**
analyzer commands, **7-1**
application program won't load in managed memory, **5-13**
assembly language trace list, how to obtain, **4-23**
avoiding disable on prefetch of exit, **1-7**
avoiding enable/trigger on unused prefetch, **1-7**
avoiding unused prefetches in prestore trace, **2-4**

B

branch dequeuing incorrect in trace list, **6-16**
?BRANCH? shows in trace list, **6-16**
branching effects, **1-3**
bus cycles in dequeued trace lists, **6-12**
bus_control, **3-7**

C

call to modules causing prestore problems, **2-2**
calling module in a prestore trace, **2-2**
command syntax, **7-1**
conditions when starting to configure deMMUer, **5-1**
configuration created to setup deMMUer, **5-6**
configuration of deMMUer, discussion of the, **4-13**
configuration of deMMUer, where is the information, **4-15**
copy mmu_tables/mmu_mappings command syntax, **5-1**
CRP root pointer descriptor, where to find it, **5-4**
cycle_type, **3-5**

D data and program fetches in 68030, **1-6**
definition of prestore measurements, **2-1**
definitions of status bits, **3-16**
deMMUer configuration display, details of the, **4-13**
deMMUer configuration display, how to access it, **4-16**
deMMUer configuration, how to gain access, **5-5**
deMMUer configuration, how to set it up, **5-5**
deMMUer configuration, where is information for setup, **4-15**
deMMUer control from emulation configuration, **4-11**
deMMUer discussion, **4-1**
deMMUer example usage, **5-1**
deMMUer hardware disable, **5-2**
deMMUer initial conditions, **5-1**
deMMUer on, but I get no logical addresses, **4-11**
deMMUer ranges overridden by TT0 and TT1, **4-17**
deMMUer root pointer descriptor type, **4-20**
deMMUer setup information, where to get it, **5-4**
deMMUer setup made from emulation configuration, **5-6**
deMMUer TC register, how to enter a value, **4-18**
deMMUer turned on by setting analysis mode, **4-12**
deMMUer virtual address start field, **4-21**
deMMUer, conditions under which it will fail, **4-24**
deMMUer, how it works, **4-22**
deMMUer, how to provide the hardware enable, **5-5**
deMMUer, how to provide the software enable, **5-7**
deMMUer, how to specify ranges to follow, **4-17**
deMMUer, how to turn it on and off, **4-11**
deMMUer, providing the software enable, **5-7**
deMMUer, when to start it, **4-10**
deMMUer, when to use it, **4-23**
deMMUer, when you need to turn it off, **4-23**
dequeued display of operands from 8-bit memory, **6-8**
dequeued trace list showing exceptions and interrupts, **6-10**
dequeued trace list, what it shows, **6-3**
dequeued trace lists, **6-1**
dequeuer problems, **6-12**
dequeuer, how to turn on and off, **6-1**
descriptor type for deMMUer root pointer, **4-20**
dest (in dequeued trace lists), **6-4**
disable on prefetch of exit, **1-7**
"disassemble_from_line_number" command, how to use it, **6-12**

disabling the deMMUer hardware, **5-2**
disassembly wrong in trace list, how to fix it, **5-9**
disassembly, where it begins after a jump, **6-14**
display mmu_tables/mmu_mappings command syntax, **7-1**
display registers for deMMUer setup information, **4-15**
display trace command syntax, **7-1**

E

enabling the 68030 MMU hardware, **5-3**
enabling the software of the deMMUer, **5-7**
equals sign "=" in trace list, **1-5**
equivalent addresses in trace list, **1-5**
error messages unique to use of MMU, **4-6**
error messages used by mmu_tables display, **4-6**
example of exceptions and interrupts in dequeued trace list, **6-10**
example of operands from 8-bit memory in dequeued trace list, **6-8**
example of short jump in dequeued trace list, **6-4**
example of stack push and pop in dequeued trace list, **6-6**
example setup and use of deMMUer, **5-1**
example triggering on interrupt status, **3-17**
example unused prefetch, **1-4**
exceptions in dequeued trace list, **6-10**
exit prefetched, causing disable, **1-7**

F

fcode, **3-2**
fetch of two opcodes, **1-5**
finding the desired prestore state, **2-5**
flushing the ATC before enabling the MMU, **4-22**
function code mapping in MMU, **4-6**

H

hardware disable of deMMUer, **5-2**
hardware enable of 68030 MMU, **5-3**
hardware enabling the deMMUer, **5-5**
how "long_aligned" modifies address symbols, **1-6**
how the 68030 fetches program and data, **1-6**
how to avoid enable/trigger on unused prefetch, **1-7**
how to turn on the dequeuer, **6-1**

I

information available for setting up deMMUer, **5-4**
information dequeued in trace lists, **6-1**
"in_procedure" used in place of "long_aligned", 1-7
instructions that access variables, **2-8**
interrupt status used by trigger, **3-17**
interrupts in dequeued trace list, **6-10**

INVALID in mmu_mappings, how to troubleshoot, **4-8**
Invalid translation control value error, **4-6**
inverse assembly is wrong, what may fix it?, **5-9**

J
JMP and JSR problems in dequeued trace lists, **6-14**
jump shown in dequeued trace list, **6-4**

L
left and right shift of trace list, **2-8**
loading your program in managed memory, **5-13**
logical address, table details for mapping, **4-4**
logical-physical analysis comparison, **4-1**
logical-to-physical mappings, displaying, **4-2**
logical/physical displays, how to select, **5-7**
logical/physical status specifications, **3-4**
long root pointer descriptor type, **4-20**
long_aligned module entry+6, **2-4**
"long_aligned" replaced by "in_procedure", **1-7**
"long_aligned", when to use it, **1-6**
"low_word" token, when to use it, **6-14**

M
memory management discussion, **4-1**
missing the calling module in a prestore trace, **2-4**
MMU ATC flushing before startup, **4-22**
MMU hardware enable, **5-3**
MMU map displays, **4-1**
MMU tables for a selected logical address, **4-4**
MMU tables, definitions of the display elements, **4-4**
MMU tables, problem of seeing mappings, **4-2**
MMU tables/mappings, how function codes are shown, **4-6**
MMU, how it affects use of symbols, **4-10**
MMU, seeing the overall list of mappings, **4-2**
MMU, special error messages, **4-6**
MMU, when to start up the deMMUer, **4-10**
mmu_mappings, **4-2**
mmu_mappings display, how to position it, **4-3**
mmu_tables, **4-4**
mnemonic status values in trace lists, **3-8**
module-call problems in prestore traces, **2-2**
MOT RSV, defined in table 4-1, **4-4**
moving trace list left and right, **2-8**
multiple "disassemble_from" commands, **6-12**

N

no-op used to avoid prefetch of exit, **1-8**
NOT TAKEN shown in trace list, **6-3**
not_range used in prestore specification, **2-6**

O

only called module in prestore trace, **2-4**
operands from 8-bit memory, dequeued example, **6-8**
operating system "os" and what it does, **5-3**
operating system example, trace lists of, **5-7**

P

"p" ahead of address in Address column or trace list, **4-22, 5-12**
page root pointer descriptor type, **4-20**
"phys" shown in trace list, **4-22**
physical address, how it is noted in a trace, **5-12**
physical-logical analysis comparison, **4-1**
physical/logical displays, how to select, **5-7**
physical/logical status specifications, **3-4**
pipeline discussion, **1-1**
pipeline flush, **1-3**
prefetch and pipeline discussion, **1-1**
prefetch discussion, **1-1**
prefetches in prestore trace lists, **2-4**
prestore specification with "not_range", **2-6**
prestore trace problems, **2-2**
prestore traces described, **2-1**
prestoring instructions that access variables, **2-8**
problem dequeuing branches in trace lists, **6-16**
problems in dequeued trace lists, **6-12**
program and data fetches in 68030, **1-6**
program fetch with two opcodes, **1-5**
program won't run correctly in managed memory, **5-13**
push and pop example in dequeued trace list, **6-6**

R

reading trace lists with prefetch and pipeline, **1-2**
reads/writes to variables in prestore traces, **2-2**
resynchronizing the inverse assembler, **5-9**
right and left shift of trace list, **2-8**
root pointer descriptor type in deMMUer, **4-20**

S

"s=" specifications described, **3-1**
"sdata" (in dequeued trace lists), **6-4**
short root pointer descriptor type, **4-20**
size, **3-6**
SPMT analysis, requirement for obtaining, **4-23**

"sprog" (in dequeued trace lists), **6-4**
"src" (in dequeued trace lists), **6-4**
stack push and pop in dequeued trace list, **6-6**
starting the deMMUer, **4-10**
status (absolute) in trace lists, **3-16**
status (mnemonic) in trace lists, **3-8**
status interrupt used by trigger, **3-17**
status specification, **3-1**
status-bit definitions, **3-16**
"stck" (in dequeued trace lists), **6-4**
symbols in trace specifications, requirements to obtain, **4-23**
symbols, how to use them with the MMU, **4-10**
syntax of analyzer commands, **7-1**

T

table details for mapping a logical address, **4-4**
TAKEN shown in trace list, **6-3**
TC register of deMMUer, how to enter a value, **4-18**
TC register value, where to find it, **5-4**
trace commands using status, **3-1**
trace list showing status, **3-8**
trace list with wrong inverse assembly, why?, **5-9**
trace list, advantages of dequeuing, **6-3**
trace list, how it shows physical addresses, **5-12**
trace list, left and right shift, **2-8**
trace lists made of "os" execution, **5-7**
trace lists with prefetch and pipeline, **1-2**
trace lists, correcting the dequeued form, **6-12**
trace lists, dequeued, **6-1**
trace specifications using s=, **3-1**
traces using prestore, **2-1**
tracing for deMMUer configuration information, **4-15**
trigger on interrupt status example, **3-17**
troubleshooting INVALID addresses in mmu_mappings, **4-8**
turning on the dequeuer, **6-1**
two different root pointers, how to select the right one, **4-20**
two opcodes in program fetch, **1-5**

U

"udata" (in dequeued trace lists), **6-4**
unused prefetches, **1-3**
unused prefetches in prestore trace, **2-4**
"uprog" (in dequeued trace lists), **6-4**
use of "long_aligned", **1-6**

using "in_procedure" in place of "long_aligned", **1-7**
using "not range" in prestore specification, **2-6**

V value of TC register defined, **4-18**
variable access problems in prestore traces, **2-2**
variable accesses found by prestoring, **2-8**
virtual address start field in deMMUer, **4-21**

W when to use "long_aligned", **1-6**
writes/reads to variables in prestore traces, **2-2**

Notes

HEWLETT
PACKARD

Hewlett-Packard
Printed in the USA