

μ MPS Principles of Operation

Michael Goldweber Renzo Davoli

August 21, 2005

Contents

I	The Architecture of μMPS	6
1	Introduction	7
2	System Structure and Overview	9
2.0.1	System Status at Boot/Reset Time	13
2.0.2	Guideposts to Understanding μ MPS	14
3	Exception Handling	15
3.1	Exception Types	16
3.1.1	Program Traps (PgmTrap)	16
3.1.2	SYSCALL/Breakpoint (SYS/Bp)	17
3.1.3	TLB Management (TLB)	17
3.1.4	Interrupts (Ints)	18
3.2	Processor Actions on Exception	19
3.2.1	Processor State	21
3.2.2	Old and New Processor State Areas	22
3.3	The Cause CP0 Register	24

3.4	The Truth about ROM	27
4	Memory Management	28
4.1	Physical Memory	28
4.2	Virtual Memory in μ MPS	31
4.3	Virtual Address Translation in μ MPS	33
4.3.1	Segment Tables	33
4.3.2	Page Tables - PgTbl's	35
4.3.3	Translation Lookaside Buffer - TLB	37
4.3.4	How Virtual Address Translation Works in μ MPS	37
4.4	CP0 Registers used in Address Translation	40
4.5	The Truth About ROM	42
5	Device Interfaces	45
5.1	Device Registers	48
5.2	The Bus Device and Device Bit Maps	51
5.2.1	Bus Register Area	51
5.2.2	Installed Devices Bit Map	53
5.2.3	Interrupting Devices Bit Map	54
5.3	Disk Devices	56
5.4	Tape Devices	59
5.5	Network (Ethernet) Devices	62
5.6	Printer Devices	63
5.7	Terminal Devices	64

6	Summary of ROM Services and Library Functions	69
6.1	Bootstrap ROM Functionality	70
6.2	New ROM Services/Instructions	71
6.2.1	ROM Actions Upon Loading a New Processor State . . .	72
6.3	Accessing Machine Registers and Assembler Instructions in C . .	73
6.3.1	Accessing CP0 Instructions in C	74
6.3.2	Accessing CP0 Registers in C	75
6.3.3	Accessing ROM-Implemented Services/Instructions in C .	76
II	Interacting with μMPS	79
7	Programming and Compiling for μMPS	80
7.1	A Word About Endian-ness	81
7.2	C Language Software Development	82
7.3	The Compiling Process	86
7.3.1	The .aout Format	87
7.3.2	The .core Format	91
7.3.3	The .rom Format	92
7.3.4	Using the Compiler, Linker, and Assembler	93
7.3.5	Using The UMPS-ELF2UMPS Object File Conversion Utility	94
7.3.6	Using The UMPS-OBJDUMP Object File Analysis Utility .	95
7.4	Encapsulation Strategy for C Programming	96

7.4.1	Module Encapsulation in C	97
8	The μMPS GUI	100
8.1	The μ MPS Simulator	100
8.2	UMPS Invocation and the .UMPSRC Configuration File	101
8.3	The UMPS Main Window	105
8.3.1	The Status Lines	105
8.3.2	Run-Control Bar	107
8.3.3	“Stop-on” Area	107
8.3.4	VM indicator button	108
8.3.5	Sim Speed Slider	109
8.3.6	The CPU and CP0 & Other Registers Area	110
8.3.7	Main Window Pull-Down Menus	111
8.4	The Setup Window	116
8.5	The Memory Browser Window	118
8.5.1	Breakpoint Ranges	119
8.5.2	Suspect Ranges	121
8.5.3	Traced Ranges	122
8.6	The Symbol Table Window	123
8.7	The TLB Display Window	124
8.8	Using The UMPS-MKDEV Device Creation Utility	125
8.8.1	Creating Disk Devices	126
8.8.2	Creating Tape Cartridges	127

9	Debugging in μMPS	128
9.1	μ MPS Debugging Strategies	129
9.1.1	Using a Character Buffer to Mimic <code>printf</code>	130
9.1.2	Implementing Debugging Functions	130
9.2	Common Pitfalls to Watch Out For	132
9.2.1	Errors in Syntax	133
9.2.2	Errors in Structure Initialization	133
9.2.3	Overlapping Stack Spaces and Other Program Components	134
9.2.4	Compiler Anomalies	134
	References	136

Part I

The Architecture of μ MPS

Chapter 1

Introduction

Computer system architecture is the attributes of a computing system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation.[1]

The architecture of μ MPS is based on the MIPS R2/3000 RISC processor architecture. μ MPS's integer instruction set mirrors that of the MIPS almost perfectly. The memory management and exception handling capabilities of μ MPS are loosely based on that of the MIPS. Finally, a complete set of I/O devices (e.g. disks, printers, terminals) is provided. Since the MIPS architecture does not detail a device interface, the device interface of μ MPS is based on that found in other common architectures.

This manual, along with a MIPS processor handbook to document the integer instruction set of μ MPS, presents a complete description of the μ MPS virtual

machine. Since development for the μ MPS is done in C using a cross compiler to generate μ MPS code, it is unlikely that one will make much (any?) use of a MIPS processor handbook.

Notational conventions:

- Words being defined are *italicized*.
- Register, fields and instructions are **bold**-marked.
- Field **F** of register **R** is denoted **R.F**.
- Bits of storage are numbered right-to-left, starting with 0.
- The i -th bit of a storage unit named **N** is denoted **N**[i].
- Memory addresses and operation codes are given in hexadecimal and displayed in big-endian format.
- All diagrams illustrate memory and going from low addresses to high addresses using a left to right, bottom to top orientation.

Chapter 2

System Structure and Overview

μ MPS contains

- A main CPU.
- A system control coprocessor, **CP0**, incorporated into the main CPU.
- ROM and RAM devices. The ROM contains routines for both the bootstrap process and for exception handling.
- Peripheral devices: up to eight instances for each of four device classes. The four device classes are disks, tape devices, printers, and terminals.
- A system bus connecting all the system components.

μ MPS's main CPU implements an accurate simulation of a MIPS R2/3000 RISC processor. It provides:

- A RISC-type integer instruction set based on the load/store paradigm.

- A 32-bit *word* length for both instructions and registers. All physical addresses are 32 bits wide. The physical address space therefore is $2^{32}=4\text{GB}$; every single 8-bit byte has its own address. *doublewords* are 64 bits and *halfwords* are 16 bits.
- 32 general purpose registers (**GPR**) denoted **\$0** . . **\$31**
 - Register **\$0** is hardwired to zero (0). This register ignores loads and always returns zero on read/store.
 - Registers **\$1** . . **\$31** support both loads and stores. In addition to a numeric designation, each register has a mnemonic connotation as well. Ten of these registers are for general computations while the rest are reserved for various purposes. The most important reserved register is register **\$28**, denoted **\$SP**, is used as the stack pointer. Registers **\$26** and **\$27**, denoted **\$k0** and **\$k1** respectively are reserved solely for kernel use.
- Two special registers, **HI** and **LO**, are for holding the results from multiplication and division operations.
- A program counter, **PC**, for instruction addressing.

The system control coprocessor, **CP0**, which is incorporated into the main CPU provides:

- Support for two processor operation modes; kernel-mode and user-mode.

- Support for exception handling. See Chapter 3.
- The resolution of all virtual addresses (i.e. virtual address translation); see Chapter 4. **CP0** provides support for two memory management processing modes: Virtual memory (VM) off or VM on.

CP0 implements eight control registers. Five (**Index**, **Random**, **EntryHi**, **EntryLo**, and **BadVAddr**) are used to support virtual address translation. Two, **Cause** and **EPC** are used by the exception/interrupt handling mechanism to indicate what type of exception and/or interrupt has occurred.

Finally, **Status** is a read/writable register that controls the usability of the co-processors, the processor mode of operation (kernel vs. user), the address translation mode, and the interrupt masking bits.

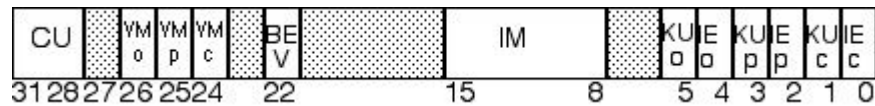


Figure 2.1: Status Register

All bit fields in the **Status** register are read/writable. In particular:

- **IEc**: bit 0 - The “current” global interrupt enable bit. When 0, regardless of the settings in **Status.IM** all external interrupts are disabled. When 1, external interrupt acceptance is controlled by **Status.IM**.
- **KUc**: bit 1 - The “current” kernel-mode user-mode control bit. When **Status.KUc**=0 the processor is in kernel-mode.

- **IEp & KUp**: bits 2-3 - the “previous” settings of the **Status.IEc** and **Status.KUc**.
- **IEo & KUo**: bits 4-5 - the “previous” settings of the **Status.IEp** and **Status.KUp** - denoted the “old” bit settings.

These six bits; **IEc**, **KUc**, **IEp**, **KUp**, **IEo**, and **KUo** act as a 3-slot deep **KU/IE** bit stack. Whenever an exception is raised the stack is pushed and whenever an interrupted execution stream is restarted, the stack is popped. See Section 3.2 for a more detailed explanation.

- **IM**: bits 8-15 - The Interrupt Mask. An 8-bit mask that enables/disables external interrupts. When a device raises an interrupt on the *i*-th line, the processor accepts the interrupt only if the corresponding **Status.IM[i]** bit is on.
- **BEV**: bit 22 - The Bootstrap Exception Vector. This bit determines the starting address for the exception vectors.
- **VMc**: Bit 24 - The “current” VM on/off flag bit. **Status.VMc=0** indicates that virtual memory translation is currently off.
- **VMp**: bit 25 - the “previous” setting of the **Status.VMc** bit.
- **VMo**: bit 26 - the “previous” setting of the **Status.VMp** bit - denoted the “old” bit setting.

These three bits; **VMc**, **VMp**, and **VMo** act as a 3-slot deep **VM** bit stack.

Whenever an exception is raised the stack is pushed and whenever an interrupted execution stream is restarted, the stack is popped. See Section 3.2 for a more detailed explanation.

- **CU**: Bits 28-31 - a 4-bit field that controls coprocessor usability. The bits are numbered 0 to 3; Setting **Status.CU[i]** to 1 allows the use of the i-th co-processor. Since μ MPS only implements **CP0** only **Status.CU[0]** is writable; the other three bits are read-only and permanently set to 0.

Trying to make use of a coprocessor (via an appropriate instruction) without the corresponding coprocessor control bit set to 1 will raise a Coprocessor Unusable exception. In particular untrusted processes can be prevented from **CP0** access by setting **Status.CU[0]=0**. **CP0** is always accessible/usable when in kernel mode (**Status.KUc=0**), regardless of the value of **Status.CU[0]**.

Important Point: Since **CP1** (the floating point co-processor) is not implemented, floating point instruction execution attempts generate a Coprocessor Unusable exception.

2.0.1 System Status at Boot/Reset Time

When μ MPS is first turned on, or reset (see Chapter 8), Coprocessor 0 is enabled (**Status.CU[0]=1**), VM is off (**Status.VMp=0**), interrupts are disabled (**Status.IEc=0**), the Bootstrap Exception Vector is on (**Status.BEV=1**), and user-mode (**Status.KUc=0**) is off; i.e. **Status=0x1040.0000**. The **PC** is set to the Bootstrap

ROM code (0x1FC0.0000 - see Section 4.1), and the **\$SP** is set to 0x0000.0000. See Section 6.1 for a description of the actions of the Bootstrap ROM code.

2.0.2 Guideposts to Understanding μ MPS

The details of the μ MPS architecture are divided into four areas. The first is exception processing which is described in Chapter 3. The second is memory management/virtual address translation which is explained in Chapter 4. The details for interacting with all the devices supported by μ MPS is found in Chapter 5 and the new CPU instructions for processor management are described in Chapter 6.

Chapter 3

Exception Handling

An *exception* is defined as a relatively infrequent event that interrupts the current execution stream. There are four categories of exceptions:

- *Program Traps* (PgmTrap): These include the Address Error, Bus Error, Reserved Instruction, Coprocessor Unusable, and Arithmetic Overflow exceptions.
- *SYSCALL/Breakpoint* (SYS/Bp): These include the System call and Breakpoint exceptions.
- *TLB Management* (TLB): These include the TLB-Modification, TLB-Invalid, PTE-MISS, and Bad-PgTbl exceptions.
- *Interrupts* (Ints): This is for external device and Software Interrupt exceptions.

3.1 Exception Types

3.1.1 Program Traps (PgmTrap)

- Address Error (*AdEL & AdES*): This exception is raised whenever
 - A load/store/instruction fetch of a word is not aligned on a word boundary.
 - A load/store of a halfword is not aligned on a halfword boundary.
 - A user-mode access is made to an address below 0x2000.0000 when **Status.VMc=0**.
 - A user-mode access is made to an address below 0x8000.0000 (kseg0) when **Status.VMc=1**.
- Bus Error (*IBE & DBE*): This exception is raised whenever an access is attempted on a non-existent physical memory location or when an attempt is made to write onto ROM storage.
- Reserved Instruction (*RI*): This exception is raised whenever an instruction is ill-formed, not recognizable, or is privileged and is executed while in user-mode.
- Coprocessor Unusable (*CpU*): This exception is raised whenever an instruction requiring the use of or access to an uninstalled or currently unavailable coprocessor is executed. Since all μ MPS control registers are implemented as part of **CP0**, access to these registers when **Status.KUc=1** and

Status.CU[0]=0 will raise this exception. **CP0** is always available when in kernel-mode (**Status.KUc=0**).

- Arithmetic Overflow (*Ov*): This exception whenever an **ADD** or **SUB** instruction execution results in a 2's-compliment overflow.

3.1.2 SYSCALL/Breakpoint (SYS/Bp)

These exceptions, denoted *Sys* and *Bp* respectively, are raised whenever a **BREAK** or **SYSCALL** instruction is executed. These instructions are used by processes to request operating system services.

3.1.3 TLB Management (TLB)

For all of these exceptions more details on the circumstances of when they are raised can be found in Chapter 4.

- TLB-Modification (*Mod*): This exception is raised when on a write request a “matching” entry is found, the entry is marked valid, but not dirty/writable.
- TLB-Invalid (*TLBL* & *TLBS*): This exception is raised whenever a “matching” entry is found but the entry is marked invalid.
- Bad-PgTbl (*BdPT*): This exception is raised during a TLB-Refill event and the ROM-TLB-Refill handler determines that the
 - address of the PTE is less than 0x2000.0000.

- address of the PTE is not word-aligned.
- PTE’s magic number is not 0x2A.
- address of the PTE + 4 + (8 * the PTE’s entry count) is greater than RAMTOP.

See Section 4.3.4 for a more complete description of TLB-Refill events.

- PTE-MISS (*PTMs*): This exception is raised during a TLB-Refill event and the ROM-TLB-Refill handler does not find a desired “matching” entry while linearly searching the PgTbl. See Section 4.3.4 for a more complete description of TLB-Refill events.

3.1.4 Interrupts (Ints)

An *interrupt*, denoted *Int*, is an exception (usually) raised by a device external to the processor. μ MPS allows for 8 *interrupt lines* to be monitored, with each line supporting a number of devices connected to it. Interrupt lines are numbered 0–7. A lower interrupt line indicates a higher servicing precedence for the devices connected to that line. Only 6 interrupt lines are available for external devices.

The two highest precedence interrupt lines, 0 & 1, are reserved for software interrupts. A software interrupt, unlike an external device interrupt which is triggered by an external event (e.g. a device completing a requested operation) is triggered by a software event; the writing of a 1 into **Cause.IP[0]** or **Cause.IP[1]**.

3.2 Processor Actions on Exception

Whenever an exception occurs there are a number of actions that the μ MPS processor always takes. Furthermore, these actions are performed atomically. They are:

1. **CP0** loads the *Exception PC (EPC)* **CP0** register with the current **PC** value.
2. The exception cause code is set in **Cause.ExcCode**.
3. The **VM** and **KU/IE** stacks in the **Status CP0** register are set/*pushed* in the following manner:

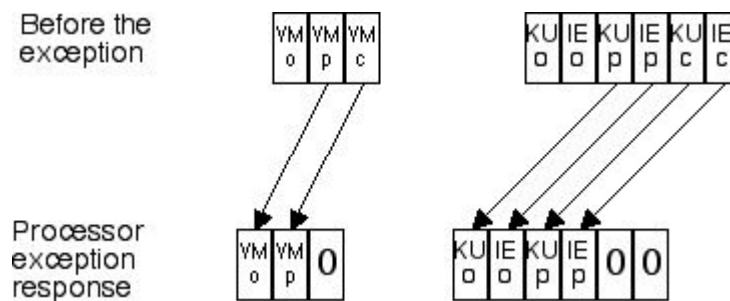


Figure 3.1: **VM** and **KU/IE** Stack Push

Hence the processor always enters an exception handler in kernel-mode with virtual memory turned off and with all interrupts disabled.

Additionally, the processor will also on:

- Address Error exceptions:
 - Load the **BadVAddr CP0** register with the offending address; this register is used even if **Status.VMc=0**.

- Interrupt exceptions:
 - Update the **Cause.IP** field bits to show on which lines interrupts are pending. In the case of a software interrupt, this was already done since it is this action that triggers a software exception.
- Coprocessor Unusable exceptions:
 - Place the appropriate coprocessor number in the **Cause.CE** field.
- TLB-Modification, TLB-Invalid, PTE-MISS, and Bad-PgTbl exceptions (these exceptions can only occur when **Status.VMc=1**):
 - Load the **BadVAddr CP0** register with the virtual address value that failed translation.
 - Load **EntryHi.SEGNO** and **EntryHi.VPN** with the **SEGNO** and **VPN** from the virtual address that failed translation.

Finally, the **PC** is loaded with the address of one of two ROM-based exception handlers. One, located in the Bootstrap ROM code (0x1FC0.0180) is used whenever **Status.BEV=1**. The other, located in the execution ROM code (0x0000.0080) is used whenever **Status.BEV=0**. This allows for different exception handlers to be used during the OS bootstrapping process, **Status.BEV=1**, and for regular processor execution, **Status.BEV=0**.

In summary, when an exception is raised, the processor performs a number of steps atomically. These include a push operation on the **KU/IE** and **VM** stacks,

saving off the current **PC**, setting the exception code in **Cause**, possibly setting some other **CP0** registers (e.g. **BadVAddr**), and finally loading the **PC** with one of two addresses depending on the setting of **Status.BEV**. What happens next is up to the ROM exception handler whose address is placed in the **PC**.

The job of the execution (or non-Bootstrap) ROM exception handler (ROM-Excpt handler) is to facilitate the “passing” of the handling of the exception to the OS. Towards this end, the ROM-Excpt handler will atomically save off the current processor state –store the contents of the processor’s registers into a given memory location– and then load a new processor state –load the processor’s registers from values stored at a given memory location.

3.2.1 Processor State

A processor state is defined as a 35 word block that contains the following registers:

- 1 word for the **EntryHi CP0** register. This register contains the current ASID (**EntryHi.ASID**).
- 1 word for the **Cause CP0** register.
- 1 word for the **Status CP0** register.
- 1 word for the **PC** (New Area) or **EPC** (Old Area) - the **PC/EPC** slot.
- 29 words for the **GPR** registers. **GPR** registers \$0, \$k0, and \$k1 are excluded.

- 2 words for the **HI** and **LO** registers.

Since there is no single non-interruptible CPU instruction that loads a processor state or stores a processor state, the ROM-Excpt handler stores and loads processor states atomically by turning off interrupts and then individually, register-by-register, first storing off the current processor state –the 35 above defined registers– and then loading these same 35 registers with new values.

Important Point: The current processor state (i.e. the current contents of the above defined 35 processor registers) of the ROM-Excpt handler is the same state the processor was in at the time the exception was raised except that that the **KU/IE** and **VM** stacks have been pushed, the **PC** at the time of the exception has been stored in the **EPC**, and **Cause.ExcCode** has been appropriately updated. When the ROM-Excpt handler stores the processor state, the **EPC** is what is stored in the **PC/EPC** slot. When the ROM-Excpt handler loads a new processor state the contents of the **PC/EPC** slot is loaded into the **PC**.

3.2.2 Old and New Processor State Areas

The ROM-Excpt handler needs a location in memory to store the current processor state in addition to an address in memory from which to load a new processor state from. The first frame of physical RAM (located at 0x2000.0000), which is called the *ROM Reserved Frame* is reserved for this purpose. The ROM Reserved Frame is also used to store process segment tables and provide stack space for the execution ROM exception handler (ROM-Excpt handler) and the execution ROM

TLB-Refill event handler (ROM-TLB-Refill handler). See Section 4.3 for more details about the segment tables stored in the ROM Reserved Frame.

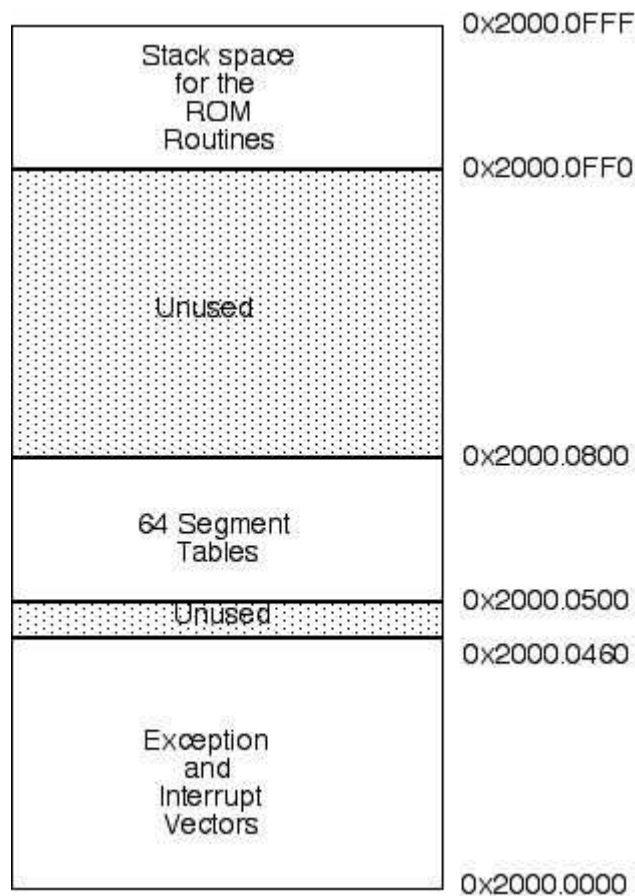


Figure 3.2: ROM Reserved Frame

Where the ROM-Excpt handler stores the current processor state in the ROM Reserved Frame is dependent on the type of exception that occurred. The current processor state is stored in either the Ints, TLB, SYS/Bp, or PgmTrap Old Area. The processor registers are then loaded from the corresponding New Area by the ROM-Excpt handler.

SYSCALL/BREAK New Area	0x2000.03D4
SYSCALL/BREAK Old Area	0x2000.0348
Program Trap New Area	0x2000.02BC
Program Trap Old Area	0x2000.0230
TLB Management New Area	0x2000.01A4
TLB Management Old Area	0x2000.0118
Interrupt New Area	0x2000.008C
Interrupt Old Area	0x2000.0000

Figure 3.3: Old and New State Areas

3.3 The Cause CP0 Register

Cause is a **CP0** register containing information about the current exception and/or pending device interrupts. As described above it is set by the hardware at the time the exception is raised and is stored as part of the current processor state in the appropriate Old Area in the ROM Reserved Frame.

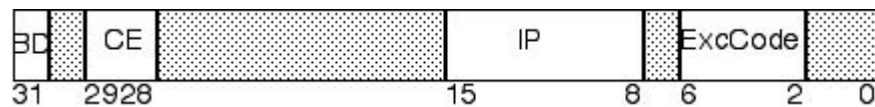


Figure 3.4: The **Cause** CP0 Register

The **Cause** fields are all read-only, except **Cause.IP[0]** and **Cause.IP[1]**, and are defined as follows:

- **ExcCode** (bits 2-6): a 5-bit field that provides a code as to which exception

was raised.

- **IP** (bits 8-15): an 8-bit field indicating on which interrupt lines interrupts are currently pending. If an interrupt is pending on interrupt line i , then **Cause.IP**[i] is set to 1.

Important Point: Many interrupt lines may be active at the same time. Furthermore, many devices on the same interrupt line may be requesting service. **Cause.IP** is always up to date, immediately responding to external device events.

The first two bits of **Cause.IP** are the only read/writable bits in **Cause**. A software interrupt is raised by writing a 1 into one of these two bits. See Section 6.3.2 for a description on how to write into the **Cause** register and Section 5.2.3 for the procedure for acknowledging software interrupts.

- **CE** (bits 28-29): A 2 bit field which indicates which coprocessor was illegally accessed when a Coprocessor Unusable exception is raised.
- **BD** (bit 31): This single bit indicates the last exception raised occurred in a *Branch Delay slot*. μ MPS faithfully simulates an R2/3000 as much as possible. As described in Chapter 7.3 μ MPS code is compiled using a standard MIPS R2/3000 cross compiler. This compiler organizes the resultant machine code for a real MIPS R2/3000 processor which includes an awareness of *delayed loads* and branch delay slots. Delayed loads and branch delay slots are conventions/techniques used by fast RISC processors to prevent pipeline slowdowns or stalls. (See [2] for more information.) Since

μ MPS is a simulated processor, there are no pipeline stages nor overlapped instruction execution. Though delayed loads and branch delay slots are present –from the compiler– they are correctly handled. Hence, the **BD** bit can safely be ignored.

The 15 codes used in **Cause.ExcCode** are:

Number	Code	Description
0	<i>Int</i>	External Device Interrupt
1	<i>Mod</i>	TLB-Modification Exception
2	<i>TLBL</i>	TLB Invalid Exception: on a Load instr. or instruction fetch
3	<i>TLBS</i>	TLB Invalid Exception: on a Store instr.
4	<i>AdEL</i>	Address Error Exception: on a Load or instruction fetch
5	<i>AdES</i>	Address Error Exception: on a Store instr.
6	<i>IBE</i>	Bus Error Exception: on an instruction fetch
7	<i>DBE</i>	Bus Error Exception: on a Load/Store data access
8	<i>Sys</i>	Syscall Exception
9	<i>Bp</i>	Breakpoint Exception
10	<i>RI</i>	Reserved Instruction Exception
11	<i>CpU</i>	Coprocessor Unusable Exception
12	<i>OV</i>	Arithmetic Overflow Exception
13	<i>BdPT</i>	Bad Page Table
14	<i>PTMs</i>	Page Table Miss

3.4 The Truth about ROM

As more fully described in Section 4.5, virtual address translation in general, and TLB-Refill event handling in particular is dealt with cooperatively between the physical hardware and the ROM-TLB-Refill handler. The physical hardware only understands **Cause.ExcCode** values [0..12]. As described in Section 4.5, it is the ROM-TLB-Refill handler that alters the value in the **Cause.ExcCode**, in the TLB Old Area from either *TLBL* or *TLBS*, the code set by the physical hardware, to either *BdPT* or *PTMs* as indicated.

Chapter 4

Memory Management

Since μ MPS supports virtual memory, there are two views of the memory subsystem: physical and virtual. We start with the physical.

4.1 Physical Memory

The physical address space is divided into equal sized frames of 4KB each. Hence a physical address has two components; a 20-bit *Physical Frame Number* or **PFN**, and a 12-bit **Offset** into the frame. Physical addresses have the following format:

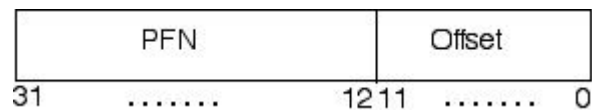


Figure 4.1: Physical Address Format

This means that μ MPS can have up to 2^{20} (or about a 1M) frames of memory.

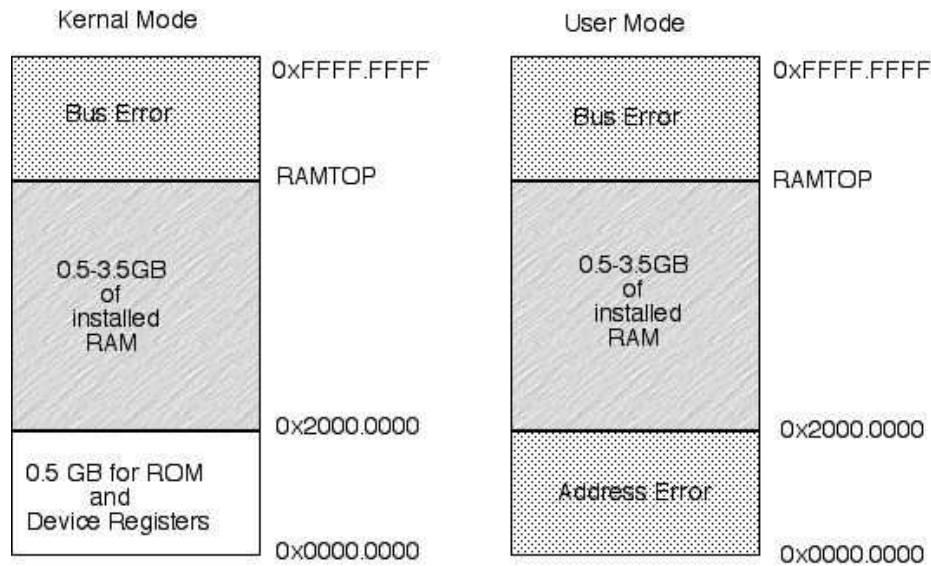


Figure 4.2: The Physical Address Space

The physical address space is the same under kernel-mode processing as under user-mode processing except for one difference. As Figure 4.2 indicates, the first 2^{17} (or about 128K) frames, which amounts to the first 0.5GB of memory can only be accessed in kernel-mode. The CPU must be in kernel-mode when reading/writing any address in this range. An Address Error exception is raised when attempting to access an address in this range while the CPU is in user-mode.

The installed physical RAM starts at 0x2000.0000 and continues up to RAM-TOP. This area will hold

- The operating system code (*text*), global variables/structures (*data*), and stack(s).
- The user processes' *text*, *data* and stacks.

- The ROM Reserved Frame. As detailed in Section 3.2.2, the ROM code needs some writable storage. The first 4KB (i.e. the first frame) of physical RAM is reserved for this purpose.

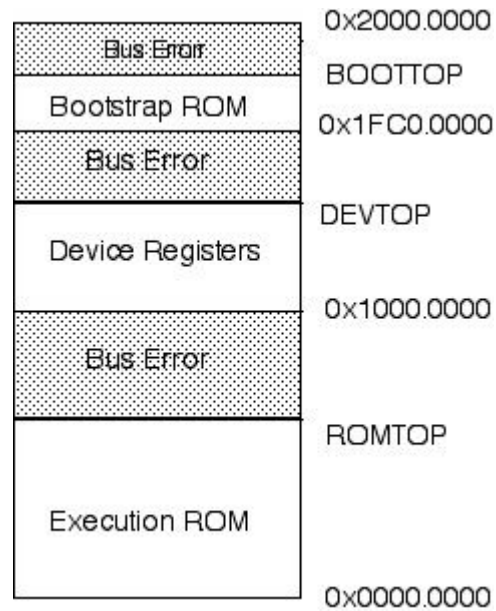


Figure 4.3: ROM Areas and Device Drivers

The first 0.5GB of the physical address space, as illustrated by Figure 4.3, is reserved for

- The Execution ROM code. This read-only code segment starts at 0x0000.0000 and goes until ROMTOP.
- The device Registers. This read/writable area begins at 0x1000.0000 and extends to DEVTOP.

- The Bootstrap ROM code. This read-only code segment starts at 0x1FC0.0000 and goes until BOOTTOP.

Any attempt to access an undefined memory area (ROMTOP – 0x1000.0000, DEVTOP – 0x1FC0.0000, BOOTTOP – 0x2000.0000, and RAMTOP – 0xFFFF.FFFF) will generate a Bus Error exception.

4.2 Virtual Memory in μ MPS

μ MPS implements a segmented-paged virtual memory (VM) scheme. Whether VM is on or not is controlled by the **Status.VMc** bit.

Important Point: Addresses between 0x0000.0000 and 0x2000.0000 are always considered physical addresses, even when virtual memory is turned on. Virtual memory address translation is disabled for addresses in this range.

The first two bits of a virtual address are the *Segment Number* (**SEGNO**). Virtual pages are the same size as physical frames, so the final 12-bits indicate an offset. The remaining 18-bits indicate the *Virtual Page Number* or **VPN**. Virtual addresses have the following format:

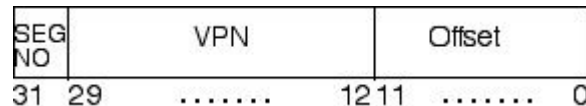


Figure 4.4: Virtual Address Format

While 2-bits are used to designate the segment, μ MPS only implements three segments:

- Segment ksegOS (aka Segment 0). ksegOS is designated by **SEGNO**'s 00 and 01. This 2GB segment is for the OS *text*, *data*, stacks, as well as the ROM code and device registers that sit at the beginning of this segment. When **Status.VMc**=1 (since talking about segments when virtual memory is turned off doesn't make any sense) any access to this segment in user-mode will generate an Address Error exception.

Important Point: When VM is off (**Status.VMc**=0) only the first 0.5GB of the address space is protected automatically by the hardware from user-mode access. When VM is on (**Status.VMc**=1), this protection extends through all of ksegOS.

- Segment kUseg2 (aka Segment 2). kUseg2 is designated by **SEGNO**=10. This 1GB virtual address space is for the use of user-mode processes.
- Segment kUseg3 (aka Segment 3). kUseg3 is designated by **SEGNO**=11. This 1GB virtual address space is for the use of user-mode processes.

As part of its VM implementation, μ MPS assigns to each process a 6-bit identifier; hence μ MPS only allows up to $2^6 = 64$ concurrent processes. To reflect the fact that each of these processes will run in its own virtual address space this identifier is called the *Address Space Identifier* (ASID). The “current” ASID is part of the processor state and is stored in **EntryHi.ASID**. See Section 6.3.1 for a description of the special kernel-mode instructions that support the reading and writing of the **EntryHi CP0** register.

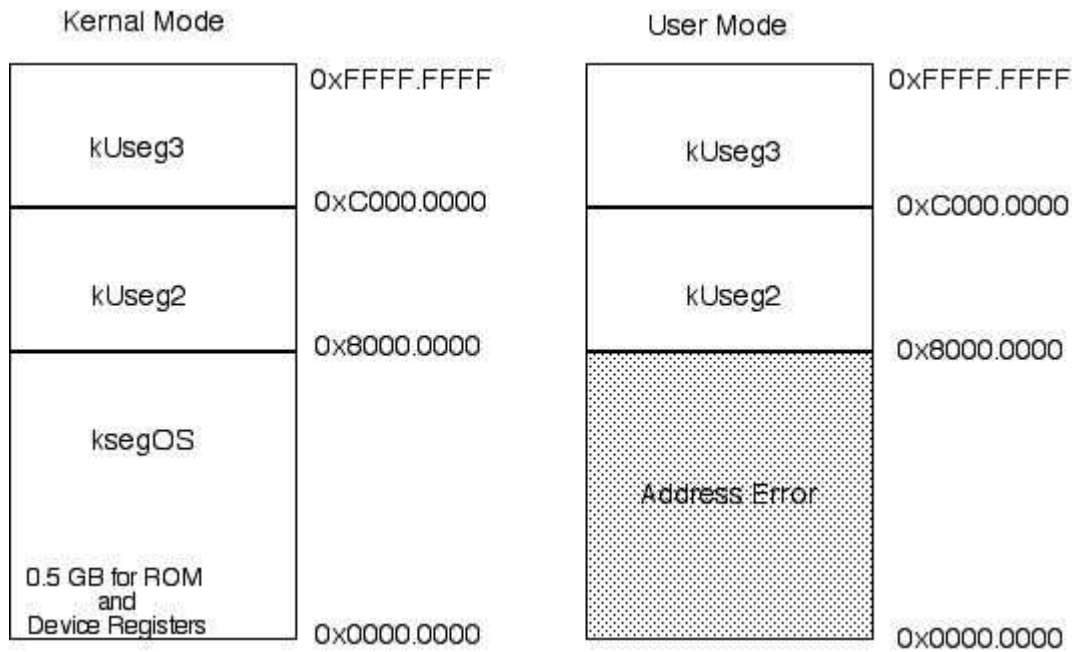


Figure 4.5: The Virtual Address Space

4.3 Virtual Address Translation in μ MPS

There are three primary components involved in virtual address translation in μ MPS; segment tables, page tables (PgTbl's), and a translation lookaside buffer (TLB).

4.3.1 Segment Tables

As discussed in Section 3.2.2, the ROM Reserved Frame contains some data structures the ROM routines access/manipulate. One of these data structures is the segment tables for all 64 ASID's.

As shown in Figure 4.6 this frame contains a 0x300 byte area which μ MPS

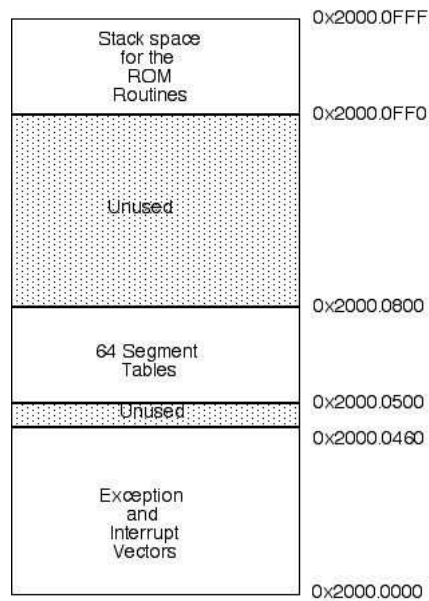


Figure 4.6: ROM Reserved Frame

expects to contain a 64×3 array of page table (PgTbl) pointers. There are 3 PgTbl pointers for each ASID; one each for ksegOS, kUseg2, and kUseg3. Since μ MPS only supports three segments, a μ MPS segment table need only contain three entries. Each entry consists solely of the address of the PgTbl associated with that segment.

When translating a virtual address, μ MPS uses **EntryHi.ASID** and the **SEGNO** of the virtual address to be translated as the indices into this table to determine the address of the indicated PgTbl. (Though kUseg2 is called Segment 2, its page table addresses are in column 1, while Segment 3's page table addresses are found in column 2.)

Important Point: The PgTbl addresses used during address translation are physical addresses.

to bound linear searches of the PgTbl; it indicates the number of entries in the PgTbl.

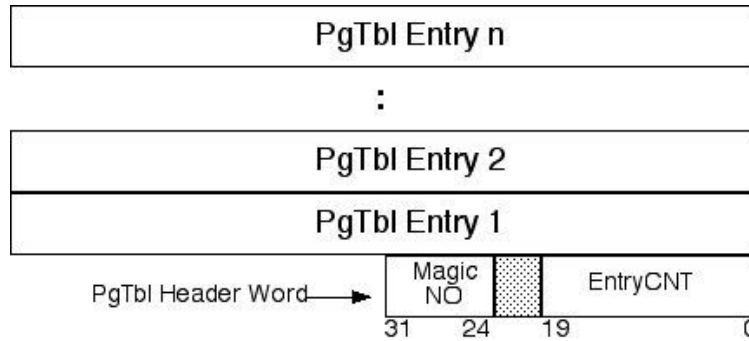


Figure 4.8: Page Table (PgTbl) Format

Each *page table entry* (PTE) consists of a double word. The first word has the same format as the **EntryHi CP0** register and the second word has the same format as the **EntryLo CP0** register.

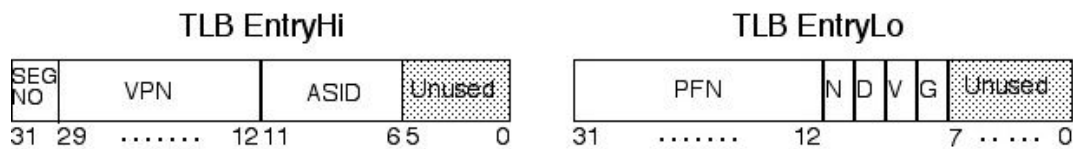


Figure 4.9: **EntryHi** and **EntryLo CP0** registers

All of these fields have been defined except the **N**, **D**, **V**, and **G** access control bits.

- **N** - the non-cacheable bit: Not used in μ MPS.
- **D** - Dirty bit: This bit is used to implement memory protection mechanisms.

When **EntryLo.D=0**, a write access to a location in the physical frame will

cause a TLB-Modification exception to be raised. This bit therefore acts as a “write protection” bit, allowing for the realization of memory protection schemes.

- **V** - Valid bit: If **EntryLo.V**=1, this PTE entry is considered valid, otherwise a TLB-Invalid exception is raised. This bit allows for the construction of memory paging schemes.
- **G** - Global bit: If **EntryLo.G**=1, the PTE entry will match any ASID with the corresponding **VPN**. This bit allows for memory sharing schemes.

4.3.3 Translation Lookaside Buffer - TLB

The TLB is an associative memory or cache, that can hold between 4–64 PTE’s. The μ MPS interface allows one to define the size of the TLB at boot/reset time; see Chapter 8 for a description on how to set the TLB size. The current size of the TLB is denoted as **TLBSIZE**.

By utilizing a cache of recently used PTE’s, μ MPS’s virtual address translation mechanism can avoid making multiple memory accesses for each translation operation; obtaining the PgTbl address and linearly searching the indicated PgTbl for a matching entry.

4.3.4 How Virtual Address Translation Works in μ MPS

Virtual address to physical address translation proceeds as follows:

1. The ASID for this translation is **EntryHi.ASID**.
2. If the virtual address to be translated is in kseg0S and **Status.KUc**=1 (i.e. User-mode) an Address Error exception is raised. Note: Address translation is disabled for all addresses below 0x2000.0000.
3. All TLB entries are simultaneously searched for a matching PTE. A match is defined as an entry in the TLB whose **SEGNO** and **VPN** are the same as those from the virtual address to be translated and either the global bit is on (**G**=1) or the ASID of the entry matches **EntryHi.ASID**. If more than one TLB entry matches, the highest numbered matching TLB entry is used.
4. If no matching entry is found, a *TLB-Refill event* occurs. A TLB-Refill event triggers a search for a matching PTE in the indicated PgTbl.
 - (a) **EntryHi.ASID** and the virtual address to be translated's **SEGNO** are used as indices into the segment table to acquire the address of the desired PgTbl.
 - (b) The PgTbl is validated as being well-formed and well-located. If any of the following conditions are true then a Bad-PgTbl exception is raised.
 - If the address of the PgTbl is less than 0x2000.0000.
 - If the address of the PgTbl is not word-aligned.
 - If the PgTbl's magic number is not 0x2A.

- If the address of the PgTbl + 4 + (8 * the PgTbl's entry count) is greater than RAMTOP.
- (c) A linear search of the PgTbl is made until either a matching entry is found or the end of the PgTbl is detected. A match is defined as the first entry found in the PgTbl whose **VPN** is the same as the **VPN** in the virtual address to be translated and either the entry's global bit is on (**G=1**) or the entry's ASID matches **EntryHi.ASID**.
 - (d) If no matching entry is found a PTE-MISS exception is raised.
 - (e) If a match is found the matching PTE entry is written into one of **TLBSIZE-1** TLB slots selected at random. The first TLB slot, slot 0, is never used during a TLB-Refill event.
5. At this point either there is a matching PTE (either found during the initial associative search, or found during the linear search and then written into the TLB) or an exception has been raised (either an Address Error, Bad-PgTbl, or PTE-MISS exception). If there is a matching TLB entry then the **V** and **D** control bits of the matching PTE are checked respectively. If no TLB-Invalid or TLB-Modification exception is raised, the physical address is constructed by concatenating the **Offset** from the virtual address to be translated to the **PFN** from the matching PTE.

4.4 CP0 Registers used in Address Translation

CP0 implements five registers used to support virtual address translation.

The contents of the TLB can be modified by writing values into the **CP0 EntryHi** and **EntryLo** registers and issuing either the *TLB-Write-Index* (**TLBWI**) or *TLB-Write-Random* (**TLBWR**) CP0 instruction. Which slot in the TLB the entry is written into is determined by which instruction is used and the contents of either the **Random** or **Index** CP0 register.

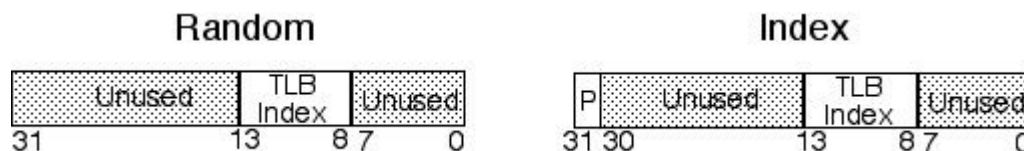


Figure 4.10: **Random** and **Index** CP0 control registers

Both the **Random** and the **Index** CP0 registers have a 6-bit **TLB-Index** field which addresses one of the **TLBSIZE** slots in the TLB. The **Index** register is a read/writable register. When a **TLBWI** instruction is executed, the contents of the **EntryHi** and **EntryLo** CP0 registers are written into the slot indicated by **Index.TLB-Index**.

The **Random** register is a read-only register used to index the TLB randomly; allowing for more effective TLB-refiling schemes. **Random.TLB-Index** is initialized to **TLBSIZE-1** and is automatically decremented by one every processor cycle until it reaches 1 at which point it starts back again at **TLBSIZE-1**. This leaves one TLB “safe” entry (entry 0) which cannot be indexed by **Random**. When a **TLBWR** instruction is executed, the contents of the **EntryHi** and **En-**

tryLo CP0 registers are written into the slot indicated by **Random.TLB-Index**.

(μ MPS's TLB-Refill algorithm uses **TLBWR** to populate the TLB.)

Three other useful **CP0** instructions associated with the TLB are the *TLB-Read* (**TLBR**), *TLB-Probe* (**TLBP**), and the *TLB-Clear* (**TLBCLR**) commands.

- The **TLBR** command places the TLB entry indexed by **Index.TLB-Index** into the **EntryHi** and **EntryLo CP0** registers. Note, that this instruction has the potentially dangerous affect of altering the value of **EntryHi.ASID**.
- The **TLBP** command initiates a TLB search for a matching entry in the TLB that matches the current values in the **EntryHi CP0** register. If a matching entry is found in the TLB the corresponding index value is loaded into **Index.TLB-Index** and the Probe bit (**Index.P**) is set to 0. If no match is found, **Index.P** is set to 1.
- The **TLBCLR** command zero's out the "unsafe" TLB entries; entries 1 through **TLBSIZE-1** This command effectively invalidates the current contents of the TLB cache.

See Sections 6.3.1 and 6.3.2 for more details on the **TLBWI**, **TLBWR**, **TLBR**, **TLBP**, **TLBCLR CP0** instructions and how to access the **EntryHi**, **EntryLo**, and **Index CP0** registers.

4.5 The Truth About ROM

As with exception processing (see Chapter 3) virtual address translation in general and TLB-Refill events in particular are dealt with in a cooperative manner. The **CP0** coprocessor performs some of the tasks while the ROM code performs some others.

Virtual address translation begins with **CP0** checking the virtual address to be translated for illegal access to kseg0; i.e. when **Status.KUc**=0. If the access is illegal, **CP0** raises an Address Error exception.

If no Address Error exception was raised, **CP0** performs an associative search of the TLB. If a match is found, as described above, the matching entry's control bits (**V** and **D**) are checked. At this point either an exception (TLB-Invalid or TLB-Modification) is raised due to the setting of the matching entry's control bits or a physical address is constructed.

If no match is found in the TLB, a TLB-Refill event occurs. A TLB-Refill event is handled in a manner similar to a TLB-Invalid exception. As described in Chapter 3, the **EPC** is loaded with the current **PC**, the **KU/IE** & **VM** stacks are pushed, **Cause.ExeCode** is loaded with either *TLBL* or *TLBS*, **BadVAddr** is loaded with the virtual address that failed translation, and **EntryHi.SEGNO** and **EntryHi.VPN** are loaded with the **SEGNO** and **VPN** from the virtual address that failed translation.

Instead of loading the **PC** with the address of one of the two ROM exception handlers, on a TLB-Refill event, the **PC** is loaded with the address of one of

two *different* event handlers, the *ROM TLB-Refill Event* handlers. One, located in the bootstrap ROM code (0x1FC0.0100) is used whenever **Status.BEV**=1. The other, located in the execution ROM code (0x0000.0000) is used whenever **Status.BEV**=0. This allows for different TLB-Refill event handlers to be used during the OS bootstrapping process, **Status.BEV**=1, and for regular processor execution, **Status.BEV**=0.

The execution (or non-Bootstrap) ROM TLB-Refill event handler (ROM-TLB-Refill handler) begins by looking up the address of the indicated PgTbl in the segment table. The PgTbl is then validated as being well-formed and well-located. Finally, the ROM-TLB-Refill handler searches the PgTbl for a matching PTE, and if found copies the PTE into a randomly selected slot in the TLB. The matching entry is first copied into the **EntryHi** and **EntryLo CP0** registers then a random TLB entry is filled as a result of issuing a **TLBWR** command. To preserve the current value of **EntryHi.ASID**, this field is saved off before the TLB is updated and restored immediately afterwards.

If the ROM-TLB-Refill handler found a match and wrote it into the TLB, the event handler will conclude by returning processor control to the interrupted execution stream. In one atomic step, the **KU/IE** & **VM** stacks are popped and the address in the **EPC** is placed in the **PC**. (The ROM-TLB-Refill handler essentially executes the *Return From Exception (RFE)* μ MPS assembler instruction. See Chapter 6 for more about popping the **KU/IE** & **VM** stacks and the **RFE** instruction.) Execution now continues with a repeated attempt to translate the virtual address that generated the TLB-Refill event in the first place. This time

CP0 will find a matching PTE in the TLB and will either correctly construct the translated physical address or it'll generate a TLB-Invalid or TLB-Modification exception - which will then “invoke” the ROM-Excpt handler to pass up the exception handling to the OS.

The other case is that the ROM-TLB-Refill handler discovered that either a Bad-PgTbl or PTE-MISS exception needs to occur. In both of these cases the ROM-TLB-Refill handler performs the same “passing up” actions as the ROM-Excpt handler; save the current state in the TLB Old Area and load the processor with the state in the TLB New Area. After the current state has been stored, and immediately prior to the loading of the new state, the ROM-TLB-Refill handler alters the **Cause.ExcCode** in the TLB Old Area from *TLBL* or *TLBS* (set by **CP0** when the TLB-Refill event was raised) to either Bad-PgTbl or PTE-MISS appropriately.

Chapter 5

Device Interfaces

μ MPS supports four different classes of external devices: disk, tape, printer and terminal. Furthermore, μ MPS can support up to eight instances of each device type. Each single device is operated by a *controller*. Controllers exchange information with the processor via *device registers*; special memory locations.

A device register is a consecutive 4-word block of memory. By writing and reading specific fields in a given device register, the processor may both issue commands and test device status and responses.

μ MPS implements the *full-handshake interrupt-driven* protocol. Specifically:

1. Communication with device i is initiated by the writing of a command code into device i 's device register.
2. Device i 's controller responds by both starting the indicated operation and setting a status field in i 's device register.

3. When the indicated operation completes, device *i*'s controller will again set some fields in *i*'s device register; including the status field. Furthermore, device *i*'s controller will generate an interrupt exception by asserting the appropriate interrupt line. The generated interrupt exception informs the processor that the requested operation has concluded and that the device requires its attention.
4. The interrupt is acknowledged by writing the acknowledge command code in device *i*'s device register.
5. Device *i*'s controller will de-assert the interrupt line and the protocol can restart. For performance purposes, writing a new command after the interrupt is generated will both acknowledge the interrupt and start a new operation immediately.

The device registers are located in low-memory starting at 0x1000.0000. As explained in Chapter 4, regardless of **Status.VMc**, all addresses between 0x1000.0000 and DEVTOP are interpreted as physical addresses. Furthermore, the device registers can only be accessed when **Status.KUc=0**.

The following table details the correspondence between device class/type and interrupt line.

Interrupt Line #	Device Class
2	Bus (Interval Timer)
3	Disk Devices
4	Tape Devices
5	Network (Ethernet) Devices
6	Printer Devices
7	Terminal Devices

Some important issues relating to device management:

- Since there are multiple interrupt lines, and multiple devices attached to the same interrupt line, at any point in time there may be multiple interrupts pending simultaneously; both across interrupt lines and on the same interrupt line.
- The lower the interrupt line number, the higher the priority of the interrupt. Note how fast/critical devices (e.g. disk devices) are attached to a high priority interrupt line while slow devices are attached to the low priority interrupt lines.
- Interrupt lines 0 & 1 are reserved for use by the processor for software interrupts.
- Disk and tape devices support *Direct Memory Access* (DMA); that is through cooperation with the bus, these devices are able to transfer whole blocks of data to/from memory from/to the device. Data blocks must be both word-

aligned and of multiple-word in size. μ MPS supports any number of concurrent DMA operations; each on a different device. Care must be taken to prevent simultaneous DMA operations on the same chunk of memory.

- After an operation has begun on a device, its device register “freezes” – becomes read-only – and will not accept any other commands until the operation completes.
- Any device register for an uninstalled device is “frozen” – set to zero – and subsequent writes to the device register have no effect.
- Device registers use only physical addresses; this includes addresses used in DMA operations.
- Each device in μ MPS except the bus device, is identified by the interrupt line it is attached to and its device number; an integer in [0..7]. μ MPS limits the number of devices per interrupt line to eight.
- For performance reasons, devices in the same class are, by default, attached to the same interrupt line.

5.1 Device Registers

All devices, except the bus, share the same device register structure.

Field #	Address	Field Name
0	(base) + 0x0	STATUS
1	(base) + 0x4	COMMAND
2	(base) + 0x8	DATA0
3	(base) + 0xc	DATA1

While each device class has a specific use and format for these fields, all device classes, except terminal devices, use:

- **COMMAND** to allow commands to be issued to the device controller.
- **STATUS** for the device controller to communicate the device status to the processor.
- **DATA0** & **DATA1** to pass additional parameters to the device controller or the passing of data from the device controller.

All 40 device registers in μ MPS are located in low memory starting at 0x1000.0000.

This area also includes three other data structures:

- *Bus Register Area*: for system status information and the “bus device register.”
- *Installed Devices Bit Map*: which indicates which devices are actually installed and where.
- *Interrupting Devices Bit Map*: which indicates which devices have an interrupt pending.

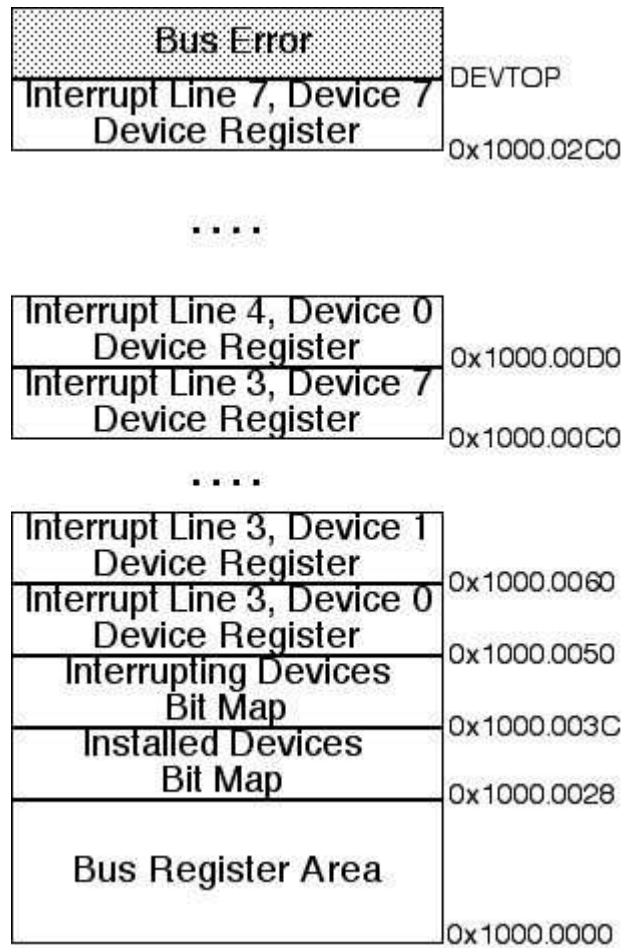


Figure 5.1: Device Registers Area

Given an interrupt line (IntLineNo) and a device number (DevNo) one can compute the starting address of the device's device register:

$$\text{devAddrBase} = 0x1000.0050 + ((\text{IntlineNo} - 3) * 0x80) + (\text{DevNo} * 0x10)$$

5.2 The Bus Device and Device Bit Maps

The bus acts as the interface between the processor and the RAM, ROM, and all the external devices. In particular the bus performs the following tasks:

1. Management of the time of Day (TOD) clock and Interval Timer.
2. Arbitration among the interrupt lines, the devices attached to each interrupt line and the device registers.
3. Repository of basic system information.

5.2.1 Bus Register Area

The bus register area is a 10 word area containing

Physical Address	Field Name
0x1000.0000	RAM Base Physical Address
0x1000.0004	Installed RAM Size
0x1000.0008	Exec. ROM Base Physical Address
0x1000.000c	Installed Exec. ROM Size
0x1000.0010	Bootstrap ROM Base Physical Address
0x1000.0014	Installed Bootstrap ROM Size
0x1000.0018	Time of Day Clock - High
0x1000.001c	Time of Day Clock - Low
0x1000.0020	Interval Timer
0x1000.0024	Time Scale

The first 6 words/fields are read-only and are set at system boot/reset time. RAMTOP is calculated by adding the RAM base physical address to the installed RAM size. ROMTOP and BOOTTOP are calculated in similar fashion.

The other four words are:

1. Time Scale: A read-only field, set at system boot/reset time which indicates the number of clock ticks that will occur in a microsecond. As described in Chapter 8 one may adjust the processor clock speed. When the processor speed is set to 1MHz, the Time Scale is set to 1. This field is used to help make accurate timing computations.
2. Time of Day Clock (TOD): This read-only doubleword register (split into its high and low word parts) is set by μ MPS circuitry to zero at system

boot/reset time. It is incremented by one after every processor cycle; i.e. a clock tick. Each μ MPS machine instruction is designed to take one processor cycle to execute.

3. Interval Timer: A read/writable unsigned word that is decremented by one every processor cycle and is set by μ MPS circuitry to 0xFFFF.FFFF at system boot/reset time. The Interval Timer will generate an interrupt on interrupt line 2 whenever it makes the 0x0000.0000 \Rightarrow 0xFFFF.FFFF transition.

This is the only device attached to interrupt line 2, hence any interrupt on this line may be assumed to be associated with the Interval Timer. Since any value may be loaded into the Interval Timer and interrupts will only be generated at the 0x0000.0000 \Rightarrow 0xFFFF.FFFF transition care must be taken to avoid inadvertently loading the Interval Timer with a very large value; i.e. a small negative integer.

Interval Timer interrupts are acknowledged by writing a new value into the Interval Timer register.

5.2.2 Installed Devices Bit Map

This is a read-only five word area that indicates which devices are attached to which interrupt line. One word each is reserved to describe the devices attached to interrupt lines 3–7.

Word #	Physical Address	Field Name
0	0x1000.0028	Interrupt Line 3 Installed Devices Bit Map
1	0x1000.002C	Interrupt Line 4 Installed Devices Bit Map
2	0x1000.0030	Interrupt Line 5 Installed Devices Bit Map
3	0x1000.0034	Interrupt Line 6 Installed Devices Bit Map
4	0x1000.0038	Interrupt Line 7 Installed Devices Bit Map

Each Installed Devices Bit Map word has the same format:

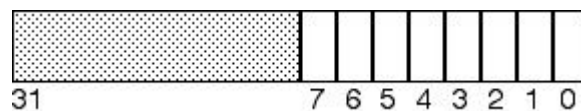


Figure 5.2: Installed Devices Bit Map

When bit i in word j is set to one then there is a device, with device number i that is attached to interrupt line $j + 3$. These words are set by μ MPS at system boot/reset time and never change.

5.2.3 Interrupting Devices Bit Map

This is a read-only five word area that indicates which devices have an interrupt pending. One word each is reserved to indicate which devices have interrupts pending on interrupt lines 3–7.

Word #	Physical Address	Field Name
0	0x1000.003C	Interrupt Line 3 Interrupting Devices Bit Map
1	0x1000.0040	Interrupt Line 4 Interrupting Devices Bit Map
2	0x1000.0044	Interrupt Line 5 Interrupting Devices Bit Map
3	0x1000.0048	Interrupt Line 6 Interrupting Devices Bit Map
4	0x1000.004C	Interrupt Line 7 Interrupting Devices Bit Map

Interrupting Devices Bit Map words have the same format as Installed Device Bit Map words; Figure 5.2. When bit i in word j is set to one then device i attached to interrupt line $j + 3$ has a pending interrupt.

An interrupt pending bit is turned on automatically by the hardware whenever a device's controller asserts the interrupt line to which it is attached. The interrupt will remain pending –the pending interrupt bit will remain on– until the interrupt is acknowledged. Interrupts for external devices are acknowledged by writing the acknowledge command code in the appropriate device's device register.

Whenever any of the devices on interrupt line i has an interrupt pending, in addition to the interrupt pending bit(s) in the $i - 3$ rd word of the Interrupting Devices Bit Map being on, **Cause.IP**[i] will also be on. **Cause.IP**[i] will only be off when none of the devices attached to line i have a pending interrupt.

Interrupt pending bits, both in **Cause.IP** and in the Interrupting Devices Bit Map get automatically turned on in response to device controllers asserting interrupt lines. The interrupt masking flags, **Status.IEc** and **Status.IM**, are used to determine if a pending interrupt actually generates an interrupt exception or not.

A pending interrupt on interrupt line i will generate an interrupt exception if both **Status.IEc** and **Status.IM**[i] are set to 1.

There are no Interrupting Devices Bit Maps for interrupt lines 0–2. **Cause.IP**[2]=1 should be interpreted as signalling a pending interrupt from the Interval Timer, while **Cause.IP**[0]=1 or **Cause.IP**[1]=1 indicate a pending software interrupt. As discussed above, an Interval Timer interrupt is acknowledged by writing a new value into the Interval Timer register. A software interrupt is acknowledged by directly writing the **Cause** register directly to set **Cause.IP**[0]=0 or **Cause.IP**[1]=0.

Important Point: Many interrupt lines may be active at the same time. Furthermore, many devices on the same interrupt line may be requesting service. **Cause.IP** and the Interrupting Devices Bit Map are always up to date, immediately responding to external device events.

5.3 Disk Devices

μ MPS supports up to eight DMA supporting read/writable hard disk drive devices. All μ MPS disk drives have a blocksize equal to the μ MPS framesize of 4KB. Each installed disk drive's device register **DATA1** field is read-only and describes the physical characteristics of the device's geometry.

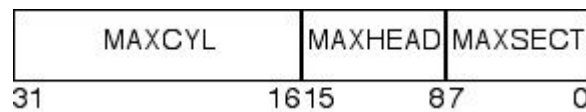


Figure 5.3: Disk Device **DATA1** Field

μ MPS disk drives can have up to 65536 cylinders/track, addressed [0..**MAXCYL**-1]; 256 heads (or track surfaces), addressed [0..**MAXHEAD**-1]; and 256 sectors/track, addressed [0..**MAXSECT**-1]. Each 4KB physical disk block (or sector) can be addressed by specifying its coordinates: (cyl, head, sect).

A disk drive's device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Seek Error	Illegal parameter/hardware failure
5	Read Error	Illegal parameter/hardware failure
6	Write Error	Illegal parameter/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Status codes 1, 2, and 4–7 are completion codes. An illegal parameter may be an out of bounds value (e.g. a cylinder number outside of [0..**(MAXCYL-1)**]), or a non-existent physical address for DMA transfers.

A disk drive's device register **DATA0** field is read/writable and is used to specify the starting physical address for a read or write DMA operation. Since memory is addressed from low addresses to high, this address is the lowest word-aligned physical address of the 4KB block about to be transferred.

A disk drive's device register **COMMAND** field is read/writable and is used to issue commands to the disk drive.

Code	Command	Operation
0	RESET	Reset the device and move the boom to cylinder 0
1	ACK	Acknowledge a pending interrupt
2	SEEKCYL	Seek to the specified CYLNUM
3	READBLK	Read the block located at (HEADNUM , SECTNUM) in the current cylinder and copy it into RAM starting at the address in DATA0
4	WRITEBLK	Copy the 4KB of RAM starting at the address in DATA0 into the block located at (HEADNUM , SECTNUM) in the current cylinder

The format of the **COMMAND** field, as illustrated in Figure 5.4, differs depending on which command is to be issued:

A disk operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set; "Device Ready" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

Disk device performance, because both read and write operations are DMA-

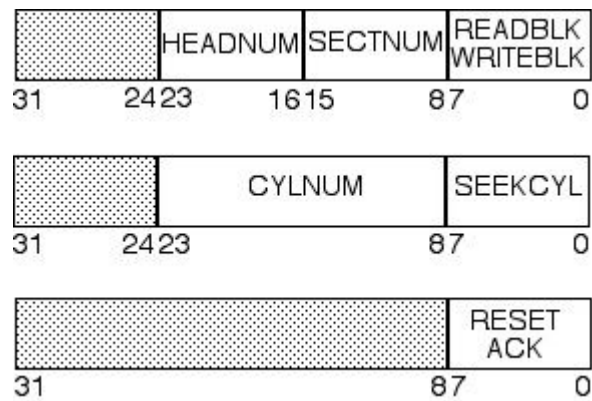


Figure 5.4: Disk Device **COMMAND** Field

based, strongly depends on the system clock speed. While read/write throughput may reach MB's/sec in magnitude, the disk hardware operations remain in the millisecond range.

5.4 Tape Devices

μ MPS supports up to eight tape-removable, DMA supporting, read-only tape devices. All μ MPS tape devices support a blocksize of 4KB. Each installed tape device's register **DATA1** field is read-only and describes the current marker under the tape head when the device is idle.

Code	Marker	Meaning
0	EOT	End of Tape
1	EOF	End of File
2	EOB	End of Block
3	TS	Tape Start

A tape starts with a **TS** marker and ends with an **EOT** marker. It may be viewed as a collection of blocks, delimited by **EOB** markers, which are divided into files, delimited by **EOF** markers. An **EOF** marker acts as the **EOB** marker for the last block of the file and the **EOT** marker act as the **EOF** (and therefore also an **EOB**) marker for the last file on the tape.

When there is no tape cartridge loaded into the tape device, the **DATA1** field will contain the **EOT** marker, and the **STATUS** field will contain the Device Ready code. Since there is no tape cartridge present, the **COMMAND** field, though, will not accept any commands. Only when a tape is loaded does the device “wake up” and begin accepting commands. When a tape cartridge is loaded, the tape device rewinds the cartridge back to the **TS** marker.

A tape drive’s device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Skip Error	Illegal command/hardware failure
5	Read Error	Illegal command/hardware failure
6	Back 1 Block Error	Illegal command/hardware failure
7	DMA Transfer Error	Illegal physical address/hardware failure

Status codes 1, 2, and 4–7 are completion codes. An illegal parameter may be an attempt to read beyond the **EOT** marker or a non-existent physical address for DMA transfers.

A tape drive's device register **DATA0** field is read/writable and is used to specify the starting physical address for a DMA read operation. Since memory is addressed from low addresses to high, this address is the lowest word-aligned physical address of the 4 KB block about to be transferred.

A tape drive's device register **COMMAND** field is read/writable and is used to issue commands to the tape drive.

Code	Command	Operation
0	RESET	Reset the device and rewind the tape to TS marker
1	ACK	Acknowledge a pending interrupt
2	SKIPBLK	Forward the tape up to the next EOB/EOT
3	READBLK	Read the current block up to the next EOB/EOT marker and copy it into RAM starting at the address in DATA0
4	BACKBLK	Rewind the tape to the previous EOB/EOT marker

A tape operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set; "Device Ready" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

Tape device performance, because read operations are DMA-based, strongly depends on the system clock speed. Tape read throughput can range from 2 MB/sec when the processor clock is set at 1 MHz, to over 4 MB/sec when the processor clock is bumped up to 99 MHz.

5.5 Network (Ethernet) Devices

Document the network interface...

5.6 Printer Devices

μ MPS supports up to eight parallel printer interfaces, each one with a single 8-bit character transmission capability.

The **DATA0** field for printer devices is read/writable and is used to set the character to be transmitted to the printer. The character is placed in the low-order byte of the **DATA0** field. The **DATA1** field, for printer devices is not used.

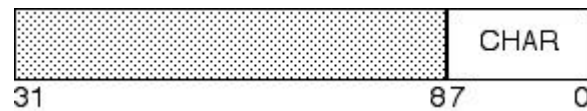


Figure 5.5: Printer Device **DATA0** Field

A printer's device register **STATUS** field is read-only and will contain one of the following status codes:

Code	Status	Possible Reason for Code
0	Device Not Installed	Device not installed
1	Device Ready	Device waiting for a command
2	Illegal Operation Code Error	Device presented unknown command
3	Device Busy	Device executing a command
4	Print Error	Error during character transmission

Status codes 1, 2, and 4 are completion codes.

A printer's device register **COMMAND** field is read/writable and is used to issue commands to the printer interface.

Code	Command	Operation
0	RESET	Reset the device interface
1	ACK	Acknowledge a pending interrupt
2	PRINTCHR	Transmit the character in DATA0 over the line

A printer operation is started by loading the appropriate value into the **COMMAND** field. For the duration of the operation the device's status is "Device Busy." Upon completion of the operation an interrupt is raised and an appropriate status code is set; "Device Ready" for successful completion or one of the error codes. The interrupt is then acknowledged by issuing an ACK or RESET command.

The printer interface's maximum throughput is 125 KB/sec.

5.7 Terminal Devices

μ MPS supports up to eight serial terminal device interfaces, each one with a single 8-bit character transmission and receipt capability.

Each terminal interface contains two sub-devices; a *transmitter* and a *receiver*. These two sub-devices operate independently and concurrently. To support the two-subdevices a terminal interface's device register is redefined as follows:

Field #	Address	Field Name
0	(base) + 0x0	RECV_STATUS
1	(base) + 0x4	RECV_COMMAND
2	(base) + 0x8	TRANSM_STATUS
3	(base) + 0xc	TRANSM_COMMAND

The **TRANSM_STATUS** and **RECV_STATUS** fields (device register fields 0 & 2) are read-only and have the following format.

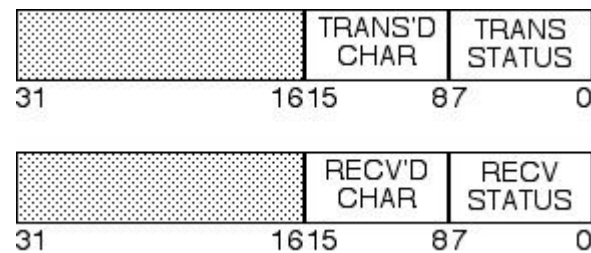


Figure 5.6: Terminal Device **TRANSM_STATUS** and **RECV_STATUS** Fields

The **status** byte has the following meaning:

Code	RECV_STATUS	TRANSM_STATUS
0	Device Not Installed	Device not installed
1	Device Ready	Device Ready
2	Illegal Operation Code Error	Illegal Operation Code Error
3	Device Busy	Device Bust
4	Receive Error	Transmission Error
5	Character Received	Character Transmitted

The meaning of status codes 0–4 are the same as with other device types.

Furthermore:

- The Character Received code (5) is set when a character is correctly received from the terminal and is placed in **RECV_STATUS.RECV'D-CHAR**.
- The Character Transmitted code (5) is set when a character is correctly transmitted to the terminal and is placed in **TRANSM_STATUS.TRANS'D-CHAR**.
- The Device Ready code (1) is set as a response to an ACK or RESET command.

A terminal's **TRANSM_COMMAND** and **RECV_COMMAND** fields are read/writable and are used to issue commands to the terminal's interface.

Code	TRANSM COMMAND	RECV COMMAND	Operation
0	RESET	RESET	Reset the transmitter or receiver interface
1	ACK	ACK	Ack a pending interrupt
2	TRANSMITCHAR	RECEIVECHAR	Transmit or Receive the character over the line

The **TRANSM_COMMAND** and **RECV_COMMAND** fields have the following format:

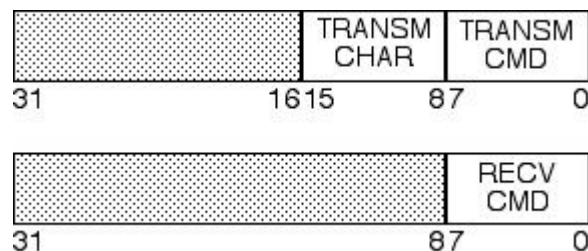


Figure 5.7: Terminal Device **TRANSM_COMMAND** and **RECV_COMMAND** Fields

RECV_COMMAND.RECV-CMD is simply the command. The **TRANSM_COMMAND** field has two parts; the command itself (**TRANSM_COMMAND.TRANSM-CMD**) and the character to be transmitted (**TRANSM_COMMAND.TRANSM-CHAR**).

A character is received, and placed in **RECV_STATUS.RECV'D-CHAR** only after a **RECEIVECHAR** command has been issued to the receiver.

The operation of a terminal device is more complicated than other devices because it is two sub-devices sharing the same device register interface. When a terminal device generates an interrupt, the (operating system's) terminal device interrupt handler, after determining which terminal generated the interrupt, must furthermore determine if the interrupt is for receiving a character, for transmitting a character, or both; i.e. two interrupts pending simultaneously.

If there are two interrupts pending simultaneously, both must be acknowledged in order to have the appropriate interrupt pending bit in the Interrupt Line 7 Interrupting Devices Bit Map turned off.

To make it possible to determine which sub-device has a pending interrupt there are two sub-device “ready” conditions; Device Ready and Character Received/Transmitted. While other device types can use a Device Ready code to signal a successful completion, this is insufficient for terminal devices. For terminal devices it is necessary to distinguish between a state of successful completion though the interrupt is not yet acknowledged, Character Received/Transmitted, and a command whose completion has been acknowledged, Device Ready.

A terminal operation is started by loading the appropriate value(s) into the **TRANSM_COMMAND** or **RECV_COMMAND** field. For the duration of the operation the sub-device’s status is “Device Busy.” Upon completion of the operation an interrupt is raised and an appropriate status code is set in **TRANSM_STATUS** or **RECV_STATUS** respectively; “Character Transmitted/Received” for successful completion or one of the error codes. The interrupt is acknowledged by issuing an ACK or RESET command to which the sub-device responds by setting the Device Ready code in the respective status field.

The terminal interface’s maximum throughput is 12.5 KB/sec for both character transmission and receipt.

Chapter 6

Summary of ROM Services and Library Functions

As described in Chapters 3 & 4, the ROM code provides vital system-level services. In particular:

- Additional system initialization at system boot or reset time.
- The ROM-Excpt handler “passes up” exception handling to the OS by first saving off the state of the processor at the time of the exception and then loading a new state to perform the actual handling of the exception. The code for the ROM-Excpt handler can be found in the file EXEC.S
- The ROM-TLB-Refill handler finds the address of the needed PgTbl. It then validates the indicated PgTbl and if valid linearly searches it for a matching PTE. If the search ends with a match, the match is then copied into the

TLB and control is returned to the interrupted execution stream, otherwise, like the ROM-Excpt handler, the ROM-TLB-Refill handler “passes up” the handling of the Bad-PgTbl or PTE-MISS exception. The code for the ROM-TLB-Refill handler can also be found in the file EXEC.S

The file EXEC.S along with all the other files mentioned in this chapter (COREBOOT.S, TAPEBOOT.S, DISKBOOT.S, CRT0.S, and LIBUMPS.S) are part of the μ MPS distribution. The recommended installation directory for these files is /USR/LOCAL/UMPS/SUPPORT/

6.1 Bootstrap ROM Functionality

Whenever μ MPS is booted or reset (See Chapter 8), **Status** is set such that **CP0** is enabled (**Status.CU[0]**=1), VM is off (**Status.VMc**=0), interrupts are disabled (**Status.IEc**=0), the Bootstrap Exception Vector is on (**Status.BEV**=1), and user-mode (**Status.KUc**=0) is off; i.e. **Status**=0x1040.0000. The **PC** is set to the bootstrap ROM code (0x1FC0.0000 - see Section 4.1), and the **\$SP** is set to 0x0000.0000.

The Bootstrap ROM code, whose code can be found in the file COREBOOT.S or TAPEBOOT.S first turns off the Bootstrap Exception Vector bit (**Status.BEV**=0) and then loads the OS. If the OS is to be presented to μ MPS on tape (i.e. TAPEBOOT.S) the OS is read in from the TAPE0 device and loaded into RAM. If the OS is to be presented to μ MPS in core (i.e. COREBOOT.S) there is nothing to read in or load since the OS will already have been placed in RAM prior to the execution of the Bootstrap ROM code. This second option, while not overly re-

alistic is nonetheless very handy for the development of student OS's. Finally, the Bootstrap ROM code sets the **PC** to the address stored at 0x2000.1004; the address of `_start()`.¹

The function `_start()`, whose code can be found in the file `CRTSO.S`, sets the **\$SP** to `RAMTOP` (stacks in μ MPS grow “downward” from high memory to low memory), and calls `main()`. If `main` ever returns, `_start()` concludes/terminates by calling **HALT**.

6.2 New ROM Services/Instructions

Additionally, the ROM code “extends” the MIPS R2/3000 integer instruction set with the following services/instructions:

- **LDST**: Atomically load the processor state (see Section 3.2.1) with the state located at the supplied *physical* memory location. This service/instruction requires the processor to be in kernel-mode, otherwise a Breakpoint exception is raised.
- **FORK**: Load the processor state with the state located at the supplied physical memory location. This instruction is NOT fully atomic; only the loading of **EntryHi**, **Cause**, **Status**, and **PC** are performed atomically. Further-

¹There is a third bootstrap option; load the OS from disk using the bootstrap ROM code in the file `DISKBOOT.S`. This file, unlike `COREBOOT.S` and `TAPEBOOT.S`, is just a skeleton. An OS developer wishing to use this option would first need to specify how the OS is arranged on `DISK0` and then complete `DISKBOOT.S` with this understanding. Alternatively, one might develop a kernel to be booted from tape (or core) whose sole purpose is to read in from `DISK0` the “real” OS and load it into RAM.

more, the complete processor state is not loaded from the supplied physical memory location, instead **EntryHi**, **Status**, and the **PC** registers are loaded from additional supplied parameters. Additionally, registers **a0**, **a1**, and **a2** are not loaded from memory but passed as they currently are to the new processor state. Finally, **v0** is also not loaded from memory and its value in the new processor state is undefined.

This service/instruction requires the processor to be in kernel-mode, otherwise a Breakpoint exception is raised.

- **PANIC**: Displays the text “kernel panic” on terminal 0 and puts the processor into an infinite loop. This service/instruction requires the processor to be in kernel-mode, otherwise a Breakpoint exception is raised.
- **HALT**: Displays the text “System halted” on terminal 0 and puts the processor into an infinite loop. This service/instruction requires the processor to be in kernel-mode, otherwise a Breakpoint exception is raised.

6.2.1 ROM Actions Upon Loading a New Processor State

It is the job of the ROM-Excpt handler to load new processor states; either as part of “passing up” exception handling (the loading of the processor state from the appropriate New Area) or for **LDST** processing. Whenever the ROM-Excpt handler loads a processor state a pop operation, as illustrated in Figure 6.1 is performed on the **KU/IE** and **VM** stacks. These two pop operations act as the compliment to the push operation that is performed when an exception is raised.

Note how the “old” values in the two stacks remain unchanged. (See Section 3.2.)

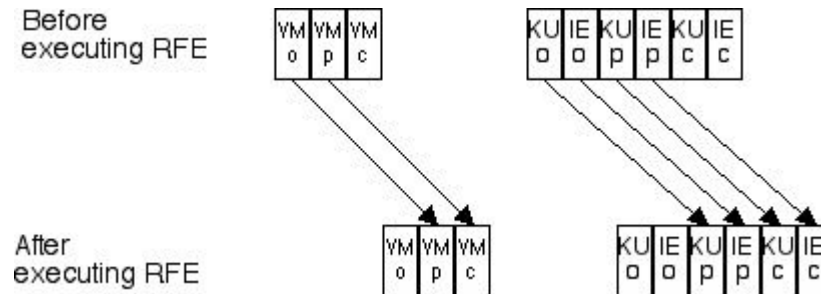


Figure 6.1: **VM** and **KU/IE** Stack Pop

As when the ROM-Excpt handler saves a processor state, the loading of a processor state is performed atomically. Since there is no single μ MPS assembly instruction to support the atomic loading of a processor state, the ROM-Excpt handler loads the new processor state register by register with interrupts disabled. The final step, the loading of the **PC** is performed using the μ MPS assembly instruction *Return From Exception* (**RFE**) which in addition to loading a new **PC** value performs the pop operations on the **KU/IE** and **VM** stacks.

6.3 Accessing Machine Registers and Assembler Instructions in C

In the process of writing an operating system one will need to access various **CP0** registers (e.g. **Status**) and issue special **CP0** assembler instructions (e.g. **TLB-CLR**). To avoid the need to program in μ MPS assembler, a C library, *libumps*, has been supplied to provide access to **CP0** instructions, the **CP0** registers, and

the extended ROM-based services/instructions. This library is implemented in LIBUMPS.S and is described by the interface file LIBUMPS.E

The libumps library also defines a routine, **STST**, which instead of providing the contents of a single register, stores the current processor state (see Section 3.2.1) at the supplied *physical* memory location. **STST**, which is NOT atomic, does not save off the current contents of the **PC**. 0 is written into the saved state instead of the **PC**.

6.3.1 Accessing CP0 Instructions in C

All five of the **CP0** instructions can be “invoked” via the libumps library as parameter-less void C functions. The semantics of these calls are described in Section 4.4. The write commands (**TLBWI**, **TLBWR** & **TLBCLR**) modify the TLB, while the Read and Probe commands modify the **EntryHi**, **EntryLo**, and **Index CP0** registers.

C usage	CP0 Instruction
void TLBWR()	TLB-Write-Random
void TLBWI()	TLB-Write-Index
void TLBR()	TLB-Read
void TLBP()	TLB-Probe
void TLBCLR()	TLB-Clear

Note, that **TLBR** has the potentially dangerous affect of altering the value of

EntryHi.ASID.

All five of these instructions require either the processor to be in kernel-mode or if in user-mode to have **Status.CU[0]=1** otherwise a Coprocessor Unusable exception is raised.

6.3.2 Accessing CP0 Registers in C

CP0 implements eight control registers. Five of these registers are read/writable, while the other three are read-only.

All eight of these registers can be read via the libumps library as parameter-less unsigned int functions. In each case the contents of the specified **CP0** register is returned to the caller. The **STST** function is different in that it is a void function whose sole parameter is a pointer to a processor state.

C usage	CP0 Register
<code>unsigned int getINDEX()</code>	Index
<code>unsigned int getENTRYHI()</code>	EntryHi
<code>unsigned int getENTRYLO()</code>	EntryLo
<code>unsigned int getSTATUS()</code>	Status
<code>unsigned int getCAUSE()</code>	Cause
<code>unsigned int getRANDOM()</code>	Random
<code>unsigned int getEPC()</code>	EPC
<code>unsigned int getBADVADDR()</code>	BadVAddr
<code>void STST(state_t *statep)</code>	STST

The five writable registers can be written via the libumps library as single parameter unsigned int functions. The single parameter is the value to be loaded into the register and the return value is the value in the register after the load operation.

C usage	CP0 Register
<code>unsigned int setINDEX(unsigned int)</code>	Index
<code>unsigned int setENTRYHI(unsigned int)</code>	EntryHi
<code>unsigned int setENTRYLO(unsigned int)</code>	EntryLo
<code>unsigned int setSTATUS(unsigned int)</code>	Status
<code>unsigned int setCAUSE(unsigned int)</code>	Cause

Note, that **setENTRYHI** has the potentially dangerous affect of altering the value of **EntryHi.ASID**.

All fourteen of these instructions require either the processor to be in kernel-mode or if in user-mode, to have **Status.CU[0]=1** otherwise a Coprocessor Unusable exception is raised.

6.3.3 Accessing ROM-Implemented Services/Instructions in C

All of the ROM services/instructions can be “invoked” via the libumps library. The semantics of these calls are described above.

C usage	ROM Service/Instr.
<code>void LDST(state_t *statep)</code>	LDST
<code>void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)</code>	FORK
<code>void PANIC()</code>	PANIC
<code>void HALT()</code>	HALT

All of these commands require that the processor be in kernel-mode otherwise a Breakpoint exception is raised.

Breakpoint Exception on Illegal ROM Service/Instruction

The four ROM services/instructions are implemented using a Breakpoint exception; the assembly code in libumps contains the **BREAK** assembly instruction forcing the exception handling mechanism to be activated. (The **EPC** register is assigned the current **PC** value, **Cause.ExcCode** is assigned the code indicating a Breakpoint exception (9), the **KU/IE** and **VM** stacks are pushed, and the ROM-Excpt handler is invoked.) The ROM-Excpt handler, if **Status.KUc**=0, performs the indicated operation; determined via a code set in **a0** by libumps prior to the **BREAK** instruction. If the ROM-Excpt handler does not recognize the code in **a0** or if **Status.KUc**=1, the handling of the Breakpoint exception is “passed up” in the usual fashion.

Hence an attempt to perform a **LDST** in user-mode does not cause the more in-

tuitive Reserved Instruction exception (**LDST** is NOT a μ MPS assembler instruction). Instead it is seen as a request for an unrecognized ROM service/instruction and is “passed up” accordingly.

Part II

Interacting with μ MPS

Chapter 7

Programming and Compiling for μ MPS

Programming for μ MPS is facilitated by a complete software development kit (SDK). The SDK contains:

- MIPSEL-LINUX-GCC; a C compiler; the gcc MIPS R2/3000 cross-compiler.
- MIPSEL-LINUX-AS; an assembler; the gcc MIPS R2/3000 cross-assembler.
- MIPSEL-LINUX-LD; a linker; the gcc MIPS R2/3000 cross-linker.
- UMPS-MKDEV; a device creation utility. This utility is used to create μ MPS disk devices and to create and load files onto μ MPS tape cartridges. See Section 8.8 for a description of this utility.
- UMPS-ELF2UMPS; an object file conversion utility. The compiler generates

ELF object files. ELF object files must be converted into one of the three object file formats recognized by μ MPS.

- UMPS-OBJDUMP and MIPSEL-LINUX-OBJDUMP; object file analysis utilities. The later utility analyzes ELF object files while the former one is used to analyze object files that have been processed with the UMPS-ELF2UMPS utility.

Using the SDK one may produce code for:

- The kernel/OS, e.g. Kaya.
- The two ROM exception handlers; ROM-Excpt handler, ROM-TLB-Refill handler, and the Bootstrap ROM routines.
- The user programs (U-proc's) that your OS (e.g. Kaya) will run.

Furthermore, one can program either in C or the μ MPS assembler language, i.e. the MIPS R2/3000 assembler language – integer instruction set only.

This guide covers kernel/OS and U-proc programming using C. See the μ MPS Web site for information about ROM and/or assembler language programming.

7.1 A Word About Endian-ness

Unlike most CPU architectures, the MIPS R2/3000 supports both big-endian and little-endian processing - though not simultaneously, the choice is pin-settable.

Similarly, μ MPS supports both big-endian and little-endian processing; the endianness of μ MPS is whatever the endianness of the host machine μ MPS happens to be running on. (e.g. i386 architectures are little-endian, while Sun Sparcs are big-endian.) As described in Chapter 8, regardless of the endianness of the host machine, the trace window's hexadecimal output is always displayed in big-endian format while the window's ASCII output is always displayed in little-endian format.

The μ MPS SDK tools MIPSEL-LINUX-GCC, MIPSEL-LINUX-AS, MIPSEL-LINUX-LD, and MIPSEL-LINUX-OBJDUMP are the little-endian versions; for running on little-endian host machines such as i386-based machines. There is an equivalent set of SDK tools for running on big-endian machines. These are named, MIPS-LINUX-GCC, MIPS-LINUX-AS, MIPS-LINUX-LD, and MIPS-LINUX-OBJDUMP respectively.

7.2 C Language Software Development

Programming in C does not easily support module/ADT encapsulation and protection. Section 7.4 outlines a strategy for implementing encapsulation using C.

While the ISO Standard for C (C99) allows for variable declarations and statements to be freely mixed and for the first expression in a `for` loop to be a declaration, these syntactic additions are not currently supported by the cross-compiler. As before the C99 ISO standard, all variables used in a function must be declared at the beginning of the function.

Runtime C-library support utilities are –obviously– not available. This includes I/O statements (e.g. `printf` from `stdio.h`), storage allocation calls (e.g. `malloc`) and file manipulation methods. In general any C-library method that interfaces with the operating system is not supported; μ MPS does not have an OS to support these calls - unless you write one to do so. The `libumps` library, described in Section 6.3, is the only support library available.

μ MPS programming requires a number of conventions for program structure and register usage that must be followed. Most of these are automatically enforced by the compiler, nevertheless there are a few that must be explicitly followed.

- The μ MPS linker requires a small function, named `_start()`. This function is to be the entry point to the program being linked. Typically `_start()` will initialize some registers and then call `main()`. After `main()` concludes, control is returned to `_start()` which should perform some appropriate termination service. Two such functions, written in assembler, are provided:
 - `CRTSO.O` This file is to be used when linking together the files for the kernel/OS. The version of `_start()` in this file assumes that the program (i.e. kernel) is loaded in RAM beginning at `0x2000.1000`. Various registers are initialized including the stack pointer (**\$SP**) which is initialized to `RAMTOP` - stacks in μ MPS grow “downward” from high memory to low memory. When `main()` returns, `_start()` invokes the **HALT** ROM service/instruction.

- **CRTI.O** This file is to be used when linking together the files for individual U-proc's. The version of `_start()` in this file assumes that the program's (i.e. U-proc's) header has 0x8000.0000 as its starting (virtual) address. Various registers are initialized but not the stack pointer (**\$SP**). `_start()` assumes that the kernel will initialize **\$SP** - which will typically be set to the end of `kUseg2`. When `main()` returns, `_start()` loads **a0** with a meaningful value (e.g. 18) and invokes the **SYSCALL** ROM service/instruction.
- The *Global Pointer* register, denoted **\$GP**, needs to point into the middle of a data structure called the *Global Offset Table* (GOT). The compiler, by generating (the GOT and) code that uses both the **\$GP** and the GOT (located somewhere in a program's *data* section), can improve the efficiency of the linking stage and the execution speed of the resulting code. The **\$GP** therefore needs to be recomputed across procedure calls. The general purpose register **t9**, which by convention holds a procedure's starting address, is used for this purpose. While the code to do all this is automatically generated by the compiler, the OS programmer needs to initialize **t9** whenever a processor state's **PC** is set/initialized to a function. Therefore whenever one assigns a value to a processor state's **PC** one must also assign the same value to that state's **t9**. (a.k.a. `s_t9` as defined in `TYPES.H`.)
- Given the load/store nature of μ MPS and the MIPS R2/3000 architecture which it is based on, the code generated by the cross-compiler may bear

little resemblance to the original source code. This is especially true if one turns on compiler optimization; which one should NEVER do when programming for μ MPS. Nevertheless, even without optimization enabled, the compiler will endeavor to keep what it perceives to be often used variables in registers.

This behavior can present problems, especially when the memory location of a variable is part of a device register (or any other hardware dependent location). The compiler may, in this case, move the variable into a register to speed up the code. Any alteration to the original variable (i.e. hardware update of the device register) will be unseen since any subsequent reference to the original variable is replaced by a register reference – which has not been updated.

To avoid this anomalous behavior all accesses to hardware defined locations should be through pointers since “caching” the pointer’s value in a register will not affect behavior. While what the pointer might point at may be updated by the hardware, the pointer’s value itself will remain constant.

In the spirit of it being better to be safe rather than sorry it is probably a good idea to also make liberal use of C’s `volatile` modifier/keyword. Any variable declared as `volatile` is never “cached” in a register to improve code performance. It is recommended that all important variables/structures be declared as `volatile`. This would include all kernel and VM-I/O support level data structures, i.e. semaphores, `PgTbl`’s, the structure de-

scribing the swap pool, etc.

7.3 The Compiling Process

The cross-compiler and cross-linker generate code in the *Executable and Linking Format* (ELF). While the ELF format allows for efficient compilation and execution by an OS it is also quite complex. Using the ELF format would therefore un-necessarily complicate the student OS development process since there are no program loaders or support libraries available until one writes them. Hence μ MPS uses three different simpler object file formats:

- *.aout*: Based on the predecessor to the ELF format, a.out, this object format is used for the U-proc programs.
- *.core*: A simple variant to the .aout format which is used as the object format for the kernel/OS.
- *.rom*: Also a variant of the .aout format which is used as the object format for the ROM exception handlers. The .rom format is for object files and not executable programs.

The supplied object file conversion utility, UMPS-ELF2UMPS performs the necessary conversion of an ELF object file/executable program into its equivalent .aout, .core, or .rom object file/executable program.

7.3.1 The .aout Format

A program, once compiled and linked may be logically split into two *areas* or *sections*. The primary areas are:

- **.text**: This area contains all the compiled code for the executable program. All of the program's functions are placed contiguously one after another in the order the functions are presented to the linker.
- **.data**: This area contains all the global and static variables and data structures. It in turn is logically divided into two sub-sections:
 - **.data**: Those global and static variables and data structures that have a defined (i.e. initialized) value at program start time.
 - **.bss**: Those global and static variables and data structures that do NOT have a defined (i.e. initialized) value at program start time.

Local, i.e. automatic, variables are allocated/deallocated on/from the program's stack, while dynamic variables are allocated from the program's *heap*. A heap, like a stack, is an OS allocated segment of a program's (virtual) address space. Unlike stack management, which is dealt with automatically by the code produced by the compiler, heap management is performed by the OS. The compiler can produce stack management code since the number and size of each function's local variables are known at compile time. Since the number and size of dynamic variables cannot be known until run-time, heap management falls to the

OS. Heap management can safely be ignored by OS authors who are not supporting dynamic variables, i.e. there are no `malloc`-type SYSCALLs in Kaya.

.aout File Format		
Field Name	File Offset	Explanation
.aout Magic File No.	0x0000	Special identifier used for file type recognition.
Program Start Addr.	0x0004	Address (virtual) from which program execution should begin. Typically this is 0x8000.00B0
.text Start Addr.	0x0008	Address (virtual) for the start of the .text area. It is fixed to 0x8000.0000
.text Memory Size	0x000C	Size of the memory space occupied by the .text section.
.text File Start Offset	0x0010	Offset into .aout file where .text begins. Since the header is part of .text , this is always 0x0000.0000
.text File Size	0x0014	Size of .text area in the .aout file. Larger than .text Mem. Size since its padded to the nearest 4KB block boundary.
.data Start Addr.	0x0018	Address (virtual) for the start of the .data area. The .data area is placed immediately after the .text area at the start of a 4KB block, i.e. .text Start Addr. + .text File Size.
.data Memory Size	0x001C	Size of the memory space occupied by the full .data area, including the .bss area.
.data File Start Offset	0x0020	Offset into the .aout file where .data begins. This should be the same as the .text File Size.
.data File Size	0x0024	Size of .data area in the .aout file. Different from the .data Mem. Size since it doesn't include the .bss area but is padded to the nearest 4KB block boundary.
\$GP Start Value	0x00A8	Starting value for \$GP , computed during linking. It is usually loaded by <code>_start()</code> into \$GP at program start time
.text	0x00B0	The program's .text area
.data	.text File Size	The program's .data area

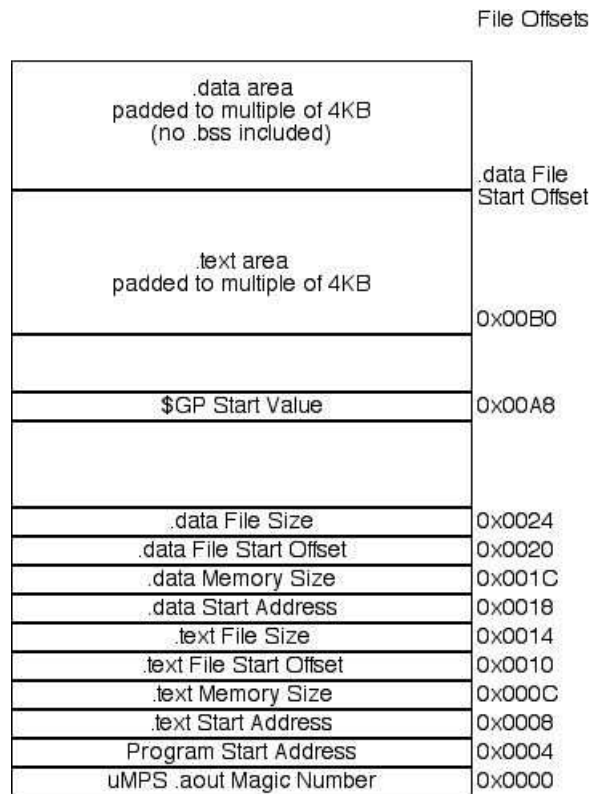


Figure 7.1: .aout File Format

Important Point: The **.data** area is given an address space immediately after the **.text** address space, aligned to the next 4KB block –insuring that **.text** and **.data** areas are completely separated. The **.bss** area immediately follows the **.data** area and is NOT aligned to a separate 4KB block.

.text and **.data** Memory Sizes are provided for sophisticated memory allocation purposes:

- The size of each U-proc’s PgTbl can be determined dynamically, instead of Kaya’s “one size fits all” approach.

- PTE's that represent the **.text** area can be marked as read-only, while entries that represent the **.data** area can be marked as writable.

The program loader which reads in the contents of a U-proc's **.aout** file, needs to be aware that the **.text** and **.data** areas are contiguous and have a starting virtual address of 0x8000.0000. The **.bss** area, while not explicitly described in the **.aout** file will occupy the virtual address space immediately after the **.data** area. The specification for Kaya does not require zero'ing out the **.bss** area, though doing so will insure that all uninitialized global and static variables and data structures begin with an initial value of zero. Finally, the loader loads the **PC** (and **t9**) with the Program Start Addr.; i.e. the contents of the second word of the U-proc's **.aout** program header (the address found at 0x8000.0004).

.aout (and **.core**) files have padded **.text** and **.data** sections to facilitate file reading/loading. Each section is padded to a multiple of the frame size/disk & tape block size. This allows the kernel/OS to easily load the program and insure that the program's **.text** and **.data** occupy disjoint frame sets.

7.3.2 The **.core** Format

The **.aout** file format provides enough information for an already-running OS to load and run such a file (i.e. U-proc). The **.core** file format must provide enough information for a Bootstrap ROM routine to load and run the OS itself. See Section 6.1 for more information about the functionality of the Bootstrap Rom routines.

The `.core` file format is identical to the `.aout` file format with the following exceptions:

- The address space begins with the address of the second frame of RAM, 0x2000.1000, instead of the (virtual) address 0x8000.0000. The first frame of RAM is reserved for the ROM Reserved Frame. The **.text** Start Addr. is now 0x2000.1000 and the Program Start Addr. is 0x2000.10B0.
- The **.data** area explicitly contains the zero-filled **.bss** area.

7.3.3 The `.rom` Format

The μ MPS distribution comes with

- 2+ Bootstrap ROM files (`COREBOOT.S`, `TAPEBOOT.S` and the skeleton `DISKBOOT.S`), which contain the Bootstrap counterparts to the ROM-Excp handler and ROM-TLB-Refill handler.
- One Execution ROM file (`EXEC.S`) containing the ROM-Excp handler and ROM-TLB-Refill handler.

Of course the intrepid OS writer may still opt to create their own ROM functions.

Important Point: Given the need for ROM code to directly manipulate μ MPS registers, ROM code development must be done using μ MPS assembler.

A `.rom` file contains only the **.text** area of its source object file. Furthermore, this **.text** area is stripped of any header information; it is just bare machine code.

The .rom format is used when translating an object file into an Execution or Bootstrap ROM file. The μ MPS simulator will load these files, place them at their correct addresses and execute their code at the appropriate times. See Chapter 8 for how to load/specify your own ROM file(s).

7.3.4 Using the Compiler, Linker, and Assembler

The compiler, assembler and linker are the “out of the box” gcc cross-platform development tools. As such they accept vast array of command line arguments/parameters.

While the linker does not require any special flags, it does require a *linker script*. Two linker scripts are provided; one for producing an object file that will eventually be converted into the .aout file format ELF32LTSMP.H.UMPSAOUT.X and one for producing an object file that will eventually be converted into the .core file format ELF32LTSMP.H.UMPSCORE.X For the curious; this is how using the same “out of the box” compiler, one can generate an object file for the kernel/OS with one address profile and object files for the U-proc programs with a different address profile.

For those who elect to write code in μ MPS assembler, e.g. (re)write a ROM Bootstrap routine or alter `_start()` in CRTI.S, it is necessary to use the -KPIC assembler flag. This flag forces the assembler to generate position independent code.

7.3.5 Using The UMPS-ELF2UMPS Object File Conversion Utility

The command-line UMPS-ELF2UMPS utility is used to convert the ELF formatted executable and object files produced by the gcc cross-platform development tools into the .aout, .core, and .rom formatted files required by μ MPS.

```
UMPS-ELF2UMPS [-V] [-M] {-K | -B | -A} <file>
```

where

- **file** is the executable or object file to be converted.
- -V: optional Flag to produce verbose output during the conversion process.
- -M: optional flag to generate the .stab symbol table map file associated with **file**.
- -K: Flag to produce a .core formatted file. This flag can only be used with an executable file. A .stab file is automatically produced with this option.
- -B: Flag to produce a .rom formatted file. This flag can only be used with an object file that does not contain relocations.
- -A: Flag to produce a .aout formatted file. This flag can only be used with an executable file.

A successful conversion will produce a file by the name of **file.core.umps**, **file.rom.umps**, or **file.aout.umps** accordingly.

A `.stab` file is a text file containing a one-line μ MPS-specific header and the contents of the symbol table from the ELF-formatted input **file**. It is used by the μ MPS simulator to map **.text** and **.data** locations to their symbolic, i.e. kernel/OS source code, names. Hence the automatic generation of the `.stab` file whenever a `.core` file is produced. Since `.stab` files are text files one can also examine/modify them using traditional text-processing tools.

In addition to its utility in tracking down errors in the UMPS-ELF2UMPS program (which hopefully no longer exist), the `-v` flag is of general interest since it illustrates which ELF sections were found and produced and the resulting header data for `.core` and `.aout` files. For `.rom` files, the `-v` flag also displays the ROM size obtained during file conversion.

7.3.6 Using The UMPS-OBJDUMP Object File Analysis Utility

The command-line UMPS-OBJDUMP utility is used to analyze object files created by the UMPS-ELF2UMPS. This utility performs the same functions as MIPS-LINUX-OBJDUMP (or MIPS-LINUX-OBJDUMP) which is included in the cross-platform development tool set. UMPS-OBJDUMP is used to analyze `.core`, `.rom`, and `.aout` object files while MIPS-LINUX-OBJDUMP is used to analyze ELF-formatted object files.

```
UMPS-OBJDUMP [-H] [-D] [-X] [-B] [-A] <file.mps>
```

where

- **file.mps** is the `.core`, `.rom`, or `.aout` object file to be analyzed.

- -H: Optional flag to show the .aout program header, if present.
- -D: Optional flag to “disassemble” and display the **.text** area in **file.mps**. This is an “assembly” dump of the code, thus it will contain load and branch delay slots; differing from the machine language version of the same code.
- -X: Optional flag to produce a complete little-endian format hexadecimal word dump of **file.mps**. Zero-filled blocks will be skipped and marked with *asterisks*. The output will appear identical regardless of whether **file.mps** is little-endian or big-endian.
- -B: Optional flag to produce a complete byte dump of **file.mps**. Zero-filled blocks will be skipped and marked with *asterisks*. Unlike with the -X flag, the endian-format of the output will depend on the endian-ness of **file.mps**; i.e. if **file.mps** is big-endian then the output will be big-endian.
- -A: flag to perform all of the above optional operations.

The output from UMPS-OBJDUMP is directed to stdout.

7.4 Encapsulation Strategy for C Programming

It is expected that your operating system will be implemented in C (and not C++ or Java). While C is not an object-oriented language, you are encouraged to divide your code into modules and to try to take advantage, as much as possible, of encapsulation.

You are strongly encouraged to create $i + 1$ (or even $i + 2$) subdirectories in your home directory. i of these directories will contain the code (“.c” files) for each of the i phases you will implement, and the $i + 1^{st}$ directory, called H, will contain your “.h” (header) files. The optional $i + 2^{nd}$ directory, called E, will contain your “.e” (external declarations) files. Instead of putting all your “.e” files into one directory, you may optionally keep each “.e” file in the phase- i directory to which it belongs.

The μ MPS distribution contains two files defining certain hardware-related constants, CONST.H, and types, TYPES.H. These will be very useful for you. Copy them into the H subdirectory of your account and make additions (deletions) as needed.

7.4.1 Module Encapsulation in C

You are encouraged to adopt the following set of conventions for programming in C. These conventions were worked out so as to provide programmers working in C some of the benefits of classes and encapsulation.

For an example consider a file (or module) that contains all the functions related to a specific well-defined purpose. This file will contain

- “public” functions: functions that the programmer wishes to be externally visible to users of the module.
- “private” functions: functions that are *helper* functions; ones which the programmer does not wish to be externally visible to users of the module.

- “public” global variables: Variables which are defined outside the scope of any individual function within the file and which the programmer wishes to be externally visible to users of the module.
- “private” global variables: Variables which are defined outside the scope of any individual function within the file and which the programmer does not wish to be externally visible to users of the module.
- “persistent” local variables: Variables which are defined inside a particular function (and hence “private”) but, like global variables, have a lifetime equal to that of the program itself (and not just the lifetime, like automatic variables, of the function within which it is defined).

Private components; functions and variables should be declared using the C keyword **static**. A static object, while visible throughout the file it is declared in cannot be accessed from outside the file; effectively creating “private” functions and variables.

A persistent variable is also declared using the keyword **static**. Any variable declared inside a function whose declaration is preceded with the keyword **static**, becomes persistent retaining its value between function calls. Static, or persistent, variables are allocated not on the stack (like automatic variables) but from the same section used for the allocation of global variables.

It is unfortunate that the keyword **static** is overloaded in C. To help differentiate their two uses it is helpful to alias the keyword **static** to *HIDDEN*.

```
#define HIDDEN static
```

Now, private components can be declared as `HIDDEN` while persistent components can be declared as `static`.

For each file/module there should also be an external declarations (“`.e`”) file. This file should contain the prototypes for each public function and global variable. Each prototype should be preceded by the keyword **`extern`**. Like a C++ “`.h`” file, any other module that makes use of one module’s public functions or variables will `#include` that module’s corresponding “`.e`” file. For example:

```
#include '../e/asl.e'
```

Finally, global structures (i.e. `typedef`’s) and constants should be defined in appropriate “`.h`” files; e.g. `CONST.H` and `TYPES.H`

Chapter 8

The μ MPS GUI

8.1 The μ MPS Simulator

The μ MPS simulator, UMPS, emulates all of the μ MPS system as described in Part I of this guide. UMPS runs on UNIX-compatible platforms (including Mac OS X) under the X-Window environment.

The UMPS simulator loads and executes programs developed for a μ MPS machine. As detailed in Section 7.3.5, all μ MPS specific files have a typical identifying “middle” extension (e.g. .CORE) and the .UMPS common final extension. While UMPS acts as a faithful emulator of a μ MPS machine, it may also be considered a sophisticated testing and debugging environment for μ MPS programs. As such, the feature-set of UMPS in general and its graphical user interface (GUI) in particular were designed to assist students in the creation of operating systems. The UMPS graphical interface provides one with the tools to exercise complete

control over the emulated machine, not only through extensive breakpoint, suspect, and tracing facilities, but also by allowing the user to modify both RAM and control registers during execution.

In the hopeful spirit that the UMPS GUI, like actual well designed GUI's, require no instruction and the observation that students rarely read GUI manuals anyway, the following sections are rather cursory. While the Section describing the Main Window (Section 8.3), is complete, the sections describing the other windows are limited to documenting their non-obvious or potentially confusing aspects. It is hoped that anyone with familiarity using a modern debugging facility will quickly be comfortable with the UMPS GUI.

8.2 UMPS Invocation and the .UMPSRC Configuration File

The μ MPS simulator is executed by entering UMPS at a shell (hopefully running in an XTerm window) prompt. In addition to recognizing X-Window command-line parameters, the execution of UMPS is controlled by a configuration file, .UMPSRC.

The search for the configuration file is as follows:

1. Search the current directory (./UMPSRC)
2. Search the user's home directory (\$HOME/UMPSRC)
3. Search for a system-wide configuration file (/USR/LOCAL/UMPS/SUPPORT/UMPSRC)

The search stops at the first .UMPSRC file found.

A sample .UMPSRC file, which also documents the default values for all .UMPSRC parameters, is provided as part of the μ MPS distribution.¹

As described in Section 6.1, the OS can be presented to μ MPS on either TAPE0, a disk, or the default option of already loaded into RAM (core). The parameter **BOOTROM** in .UMPSRC needs to be set to point to the file containing the appropriate Bootstrap ROM code; either one of the provided Bootstrap ROM files (COREBOOT.ROM.UMPS or TAPEBOOT.ROM.UMPS), or one developed by the OS author.

The .UMPSRC **EXECROM** parameter points to the file containing the execution ROM code (ROM-Except handler and ROM-TLB-Refill handler).

Possibly more important than the above ROM code parameters, since most users will be satisfied with their default values, are the **COREFILE**, **STABFILE**, and **LOADCOREFILE** parameters.

When presenting the OS as already loaded into RAM the simulator needs the name of the .core file of the OS to be loaded. The **COREFILE** parameter indicates the name of the .core file of the OS to be loaded/executed; XXXX.CORE.UMPS and the **STABFILE** parameter is the name of the corresponding symbol table file (XXXX.STAB.UMPS). Whether the **COREFILE** is actually loaded into RAM is controlled by the **LOADCOREFILE** flag; which needs to be set to true whenever the OS is to be presented as already loaded into RAM. Note that whenever

¹The recommended installation directory for this file is /USR/LOCAL/UMPS/SUPPORT/UMPSRC.SAMPLE

the OS is to be presented on either TAPE0 or a disk device, **LOADCOREFILE** should be set to false which causes UMPS to ignore the **COREFILE** parameter. The symbol table file (**STABFILE**) is always loaded regardless of how the OS is presented to the system.

Other configuration parameters describe

- **RAMPAGESIZE**: number of 4KB RAM frames present in the simulated μ MPS machine.
- **PROCSPEED**: The clock speed of the simulated μ MPS machine. This parameter controls the Time Scale Bus Register value; the number of (simulated) clock ticks that occur in a (simulated) microsecond. A **PROCSPEED** value of 1 corresponds to a simulated clock speed of 1MHz – a Time Scale value of 1.
- **TLBSIZE**: The size of the TLB in the simulated μ MPS machine.
- **SIMSPEED**: The initial speed of the simulation.
- The files to be used for active disk, tape, and terminal devices. If a device's file is not defined then that device is initially inactive. (i.e. Status is Not Operational.)

The files associated with printer and terminal devices are text files which will hold the characters output/transmitted to each device; i.e. a log file. If a printer or terminal's log file does not exist when UMPS starts, its file is automatically created.

The files associated with disk and tape devices are special files created using the UMPS-MKDEV device creation utility. These files must already exist when UMPS is started. See Section 8.8 for a description of the UMPS-MKDEV utility.

- The initial simulator “Stop-on” flags. These flags indicate whether the UMPS will initially stop on exception (**EXCSTOP**), breakpoint (**BRKPT-STOP**) or suspect (**SUSPSTOP**) events. The “Stop-on” flags for TLB-Refill events is further broken down to those that occur when **Status.KUc=0** (**TLBK0STOP**) and when **Status.KUc=1** (**TLBK1STOP**). UMPS will not stop on TLB-Refill events unless **EXCSTOP=1**.
- **EXPERTMODE**: The simulation “mode.” UMPS allows for two different modes of user interaction; beginner and expert. In beginner mode the user is asked to confirm all interactions, while in expert mode most confirmation steps are skipped.

Regardless of how the OS is presented to μ MPS, the indicated Bootstrap ROM code is loaded, and if core loading is to occur, the contents of XXXX.CORE.UMPS is loaded into RAM as well. For tape or disk OS loading, it is the responsibility of the Bootstrap ROM code to load the OS. After the Bootstrap ROM code, and optionally a core presented OS is loaded, the Execution ROM code and the OS symbol table (XXXX.STAB.UMPS) are loaded. Next any persistent devices (e.g. tape and disk) are loaded and the log files for any other device are created (e.g. files to hold terminal output). Finally, the UMPS begins execution simulation by

executing the Bootstrap ROM code.

8.3 The UMPS Main Window

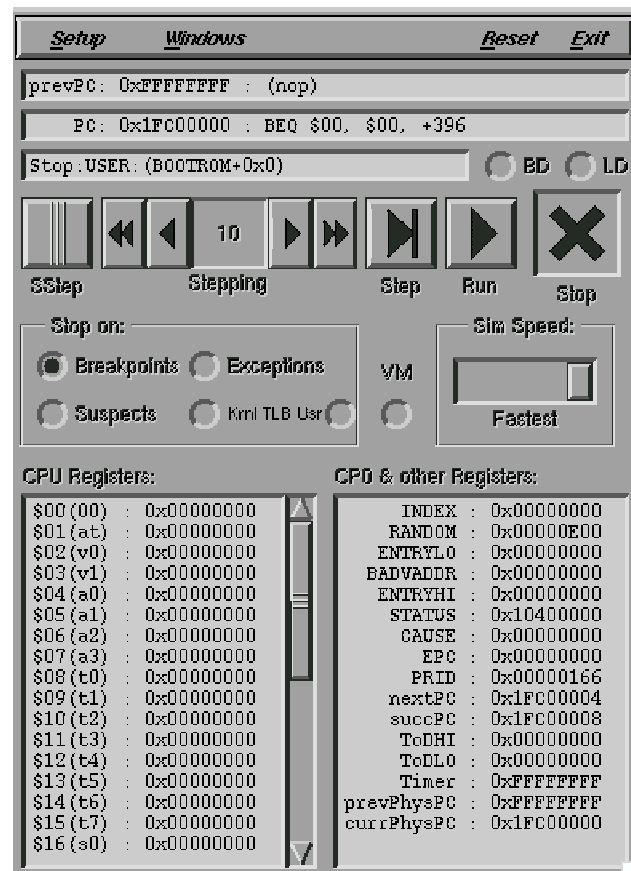


Figure 8.1: The UMPS Main Window

8.3.1 The Status Lines

The first three lines, called the *status lines*, indicate:

1. The last instruction the virtual machine executed, together with its (virtual) address - i.e. the “previous” value of the **PC**.
2. The instruction that will be executed during the next clock tick together with the “current” value of the **PC** - the (virtual) address of the instruction. The two LED’s on the third line indicate if this instruction is in a load or branch delay slot.
3. The third status line shows the simulation status: Stopped or Running. If Stopped, the reason is given:
 - **USER** for user initiated stops; clicking on the **Stop** button during execution.
 - **BRKPT** for breakpoint range accesses.
 - **EXC** for exceptions.
 - **SUSP** for suspect range accesses.

Additionally, this line will, if possible, show the location in the OS source code of the instruction to be executed. This functionality is controlled by the contents of the loaded .stab file. Hence a **PC** value not found in the symbol table (e.g. U-proc code) will not resolve to a function name and offset. This feature allows one to correlate simulator execution with the original source code.

8.3.2 Run-Control Bar

The Run-Control bar contains the **SStep** (single step), **Step**, **Run**, and **Stop** buttons.

- The **SStep** button executes one instruction and then stops.
- The **Stop** button stops instruction execution.
- The **Run** button (re)commences with instruction execution. The machine will continue to execution instructions until either a “Stop-on” event occurs, or the user clicks the **Stop** button.
- The **Step** button executes at most the number of instructions indicated in the **Stepping** range and then stops. The **Stepping** range, in addition to displaying the number of instructions in the current step-range, has four buttons: increase/decrease the step-range by either 10 or 1 instructions.

The **Step** button may not execute the indicated number of instructions because a “Stop-on” event or a **Stop** button click may occur first.

8.3.3 “Stop-on” Area

The “Stop-on” buttons area contains five buttons that control if and when instruction execution is automatically stopped; a “Stop-on” event. The initial value of these five buttons are controlled by the settings found in the .UMPSRC configuration file. Specifically:

- **Breakpoints:** Instruction execution stops when the instruction to be executed is in a breakpoint range selected by the user. See Section 8.5.1 for how to set remove breakpoint ranges.
- **Suspects:** Instruction execution stops when a memory location contained in a suspect range selected by the user is accessed by a memory read or write operation. Instruction execution stops **after** the access has occurred. See Section 8.5.2 for how to set and remove suspect ranges.
- **Exceptions:** Instruction execution stops when any exception is raised. The **Krnl TLB** and **Usr TLB** sub-buttons control whether instruction execution will stop on TLB-Refill events. The two TLB sub-buttons are not independent of the **Exceptions** button. Instruction execution will stop on TLB-Refill events (when executing in kernel-mode, user-mode or both) only if the **Exceptions** button is selected. i.e. One cannot cause instruction execution to stop on TLB-Refill events exclusive of other exception types. Typically, given the ubiquity of TLB-Refill events, one might have the **Exceptions** button selected and both TLB sub-buttons not selected - to avoid unwanted non-critical instruction execution stops while debugging.

8.3.4 VM indicator button

The **VM** LED is used to indicate whether **Status.VMc** is currently 0 (off) or 1 (on).

8.3.5 Sim Speed Slider

The **Sim Speed** Slider controls the speed of the simulator itself and should not be confused with the simulated processor speed (see Section 8.4). For each speed selection, the interface is updated after a progressively increasing number of instructions. This allows for the running of a fast test or a slow debugging session. Specifically:

- **Slowest:** The simulator shows all the instructions it executes and updates all other open windows. Instruction execution rate is approximately 50-60 instructions/second. While your mileage will vary, being dependent on the speed and load of the host machine, this figure nevertheless allows one make relative comparisons between sim speeds.
- **Slower:** The simulator displays one instruction in ten that it executes and updates all open windows accordingly. Instruction execution rate is approximately 150-500 instructions/second. If the **Memory Browser** window is open, execution speed will be on the slower end of this range due to the (frequent) need to update the memory-traced locations as changes to these locations occur. See Section 8.5.3 for how to set and remove memory trace ranges.
- **Slow:** Behaves like the **Slower** setting, but displays one instruction in 100. Execution rate is 1500-5000 instructions/second.

- **Normal:** Behaves like the **Slower** setting, but displays one instruction in 1000. Execution rate is 1500-10,000 instructions/second.
- **Fast:** The simulator displays one instruction in 1000 that it executes and updates all open windows accordingly. Unlike the **Normal** setting, execution does not slow down for updates to the **Memory Browser** window. Execution rate is 10,000-20,000 instructions/second.
- **Faster:** The simulator displays one instruction in 10,000 that it executes and updates all other open windows only then. Execution rate is 15,000-20,000 instructions/second.
- **Fastest:** The simulator displays one instruction in 20,000 that it executes and updates all other open windows only then. Execution rate is 20,000-40,000 instructions/second.

8.3.6 The CPU and CP0 & Other Registers Area

These two windows display the 32 general CPU registers, the **CP0** control registers, and some other useful register and system status values. These values may be changed by double-clicking on a selected register or system status value. This causes a small pop-up window to appear allowing the user to enter a new value for that register.

With regard to the “Other” registers and system status values:

- **PRID:** Processor ID. This is a **CP0** register containing a unique processor

ID value.

- **nextPC**: The “next” value for the **PC**.
- **succPC**: The next “next” value for the **PC**.
- **ToDHI** and **ToDLO**: The current value of the Time of Day Clock - High and Low Bus registers.
- **Timer**: The current value of the Interval Timer Bus register.
- **prevPhysPC** and **currPhysPC**: The physical addresses for the “previous” and “current” **PC** values shown in the first two status lines. Since the values in the status lines may be virtual addresses, this allows for the easy tracing of the physical location of the code being executed.

8.3.7 Main Window Pull-Down Menus

In addition to all the various areas of the main window, there are four pull-down menus. Actually two are real pull-down menus while the other two are “buttons.” While **Exit** is self explanatory, **Reset** will be explained below.

The Setup Pull-Down Menu

The Setup pull-down menu supports three options:

- **Setup Window**: This option brings up the **Setup** Window whose functions are described in Section 8.4.

- The **Reset** options: The parameters that control a simulated run are conceptually stored in two locations; the “defaults” set, which are the settings found in the .UMPSRC file, and the “setup” set which are the settings found in the **Setup Window**. When one first executes UMPS these two settings sets are identical since the “setup” set is initialized from the values found in .UMPSRC. As one proceeds these two sets will diverge through interaction with the **Setup Window**. The two **Reset** options differ by which settings set is read and applied:

- **Reset to Setup. . .**: Reset the emulated μ MPS machine and reset all simulation parameters (e.g. “Stop-on” options, simulation speed, RAM size, etc) to the values currently found in the **Setup Window**; i.e. the “setup” set. The **Reset** button in the title bar is simply a shortcut for this option.
- **Reset to Defaults. . .**: Reset the emulated μ MPS machine and reset all the simulation parameters to the values described in .UMPSRC; i.e. the “defaults” set.

During a debugging/testing session, one will frequently wish to restart the emulated machine at the initial system boot time; use **Reset to Setup. . .** or its **Reset** title bar button shortcut.

In addition to restarting the same emulation over and over using differing breakpoint, suspect, and “Stop-on” settings one may wish to re-source .UMPSRC. Instead of quitting UMPS and restarting it, one may accomplish the same goal by

selecting **Reset to Defaults**. . . Since this will re-read .UMPSRC, the .core (i.e. kernel/OS), .stab, and .rom files will be re-loaded as well. This option is useful if one made an alteration to the kernel, recompiled it and then wished to test it.

The Windows Pull-Down Menu

The Windows pull-down menu supports 15 options:

- **Symbol Table**: This option brings up the **Symbol Table** Window whose functions are described in Section 8.6.
- **Memory Browser**: This option brings up the **Memory Browser** Window whose functions are described in Section 8.5.
- **TLB Display**: This option brings up the **TLB Display** Window whose functions are described in Section 8.7.
- **Int i Device Status**: These four options ($i = [3 \dots 6]$) bring up the **Device Status** Windows; each of which describe the status of the devices attached to the indicated device interrupt line. As described in Chapter 5, disk devices are attached to line 3, tape devices are attached to line 4, network devices are attached to line 5, and printer devices are attached to line 6. For each of the eight devices attached to the interrupt line, the current status (typically the contents of the **STATUS** device register), the value of the Time of Day Clock when the device last completed an operation, and a button to have the device simulate a hardware failure if selected.

n LED indicating whether the device has suffered a hardware failure is shown.

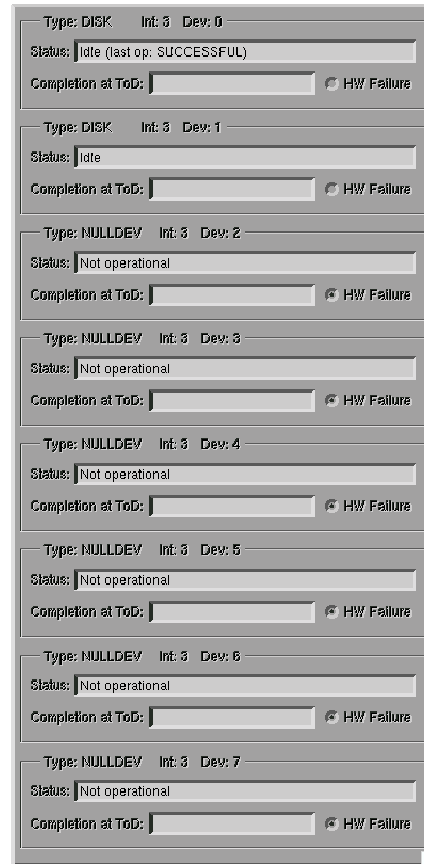


Figure 8.2: A UMPS Interrupt Line Device Status Window

- **Terminal i :** These eight options ($i = [0 \dots 7]$) bring up the **Terminal i** Windows; each of which describe the status of, display the output to, and allow for the input from an individual terminal device. A **Terminal i** Window displays the character transmission and receive status values, the separate Time of Day Clock completion values for both transmission and receive

operations, and a button to have the device simulate a hardware failure if selected. Additionally, there is a large terminal output window where all successfully transmitted characters are displayed and an **Input...** button to facilitate/simulate the typing of characters at the terminal.

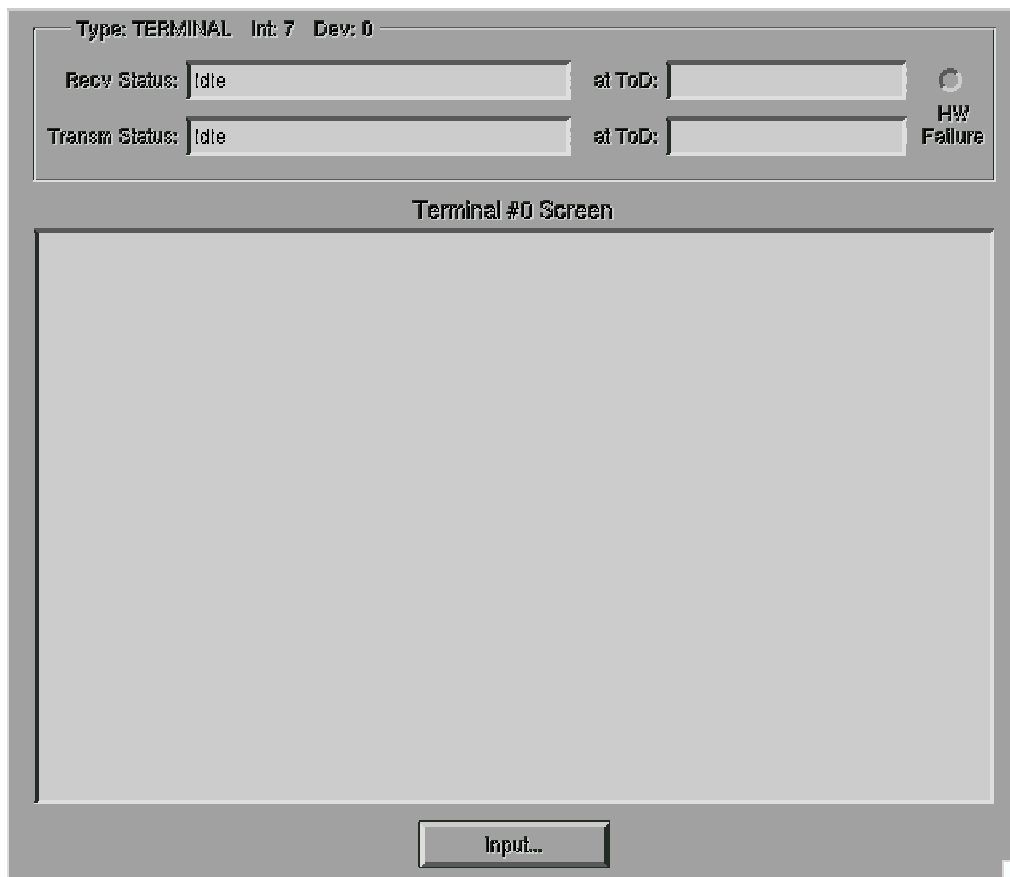


Figure 8.3: A UMPS Terminal Window

Terminal input is both immediately displayed in the terminal output window and internally buffered until a `RECEIVECHAR` is issued. While terminals are read from and written to on a character-by-character basis, user terminal

input in UMPS is on a line-by-line basis, with each line being terminated by a newline (`\N`) character.

8.4 The Setup Window

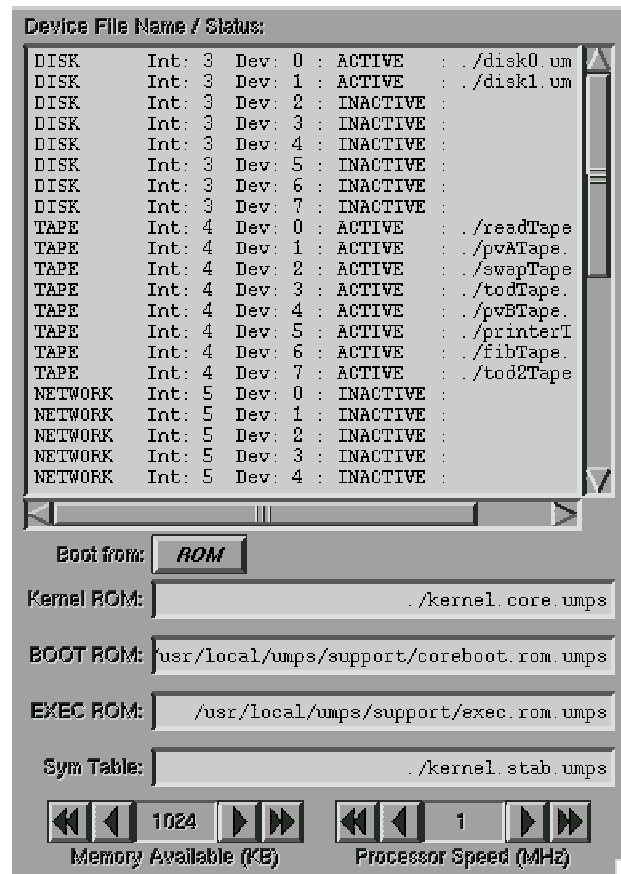


Figure 8.4: The UMPS Setup Window

The **Setup Window** essentially allows for the run-time setting of most of the .UMPSRC controlled options. The initial values displayed in the **Setup Window**

come from the contents of the .UMPSRC file.

All of the values displayed in this window are alterable. In particular a device's file mapping can be modified by clicking on the line describing the device.

Not all of the parameters one can set in .UMPSRC can be examined/alterd via the **Setup Window**. In particular the following parameters cannot be altered via the **Setup Window**.

- The size of the TLB. The current size of the TLB can nonetheless be examined via the **TLB Display Window**. (See Section 8.7.)
- The Expert Mode setting.
- The values of the five “Stop-on” flags. These values are directly available on the UMPS Main Window.
- The value of the simulation speed. Like the “Stop-on” flags, this can be controlled directly on the UMPS Main Window.

While the size of the RAM can be altered via the **Setup Window**, the semantics for the window differ from that used in .UMPSRC. In .UMPSRC one indicates the number of 4KB RAM frames present in the simulated μ MPS machine. In the **Setup Window** one works with the total amount of RAM in KB, i.e. 4 times the number of frames present.

Important Point: any change to a device configuration, system-specific parameter (e.g. Processor Speed), or support file will force the simulator to reset the system to the current setup parameters. i.e. This is as if the **Reset to Setup**. . . was

selected. The only exception to this is the loading and unloading of tape cartridges if the tape device is not busy. This allows one to perform multiple load/unload sequences on a tape device during a single simulation run.

8.5 The Memory Browser Window

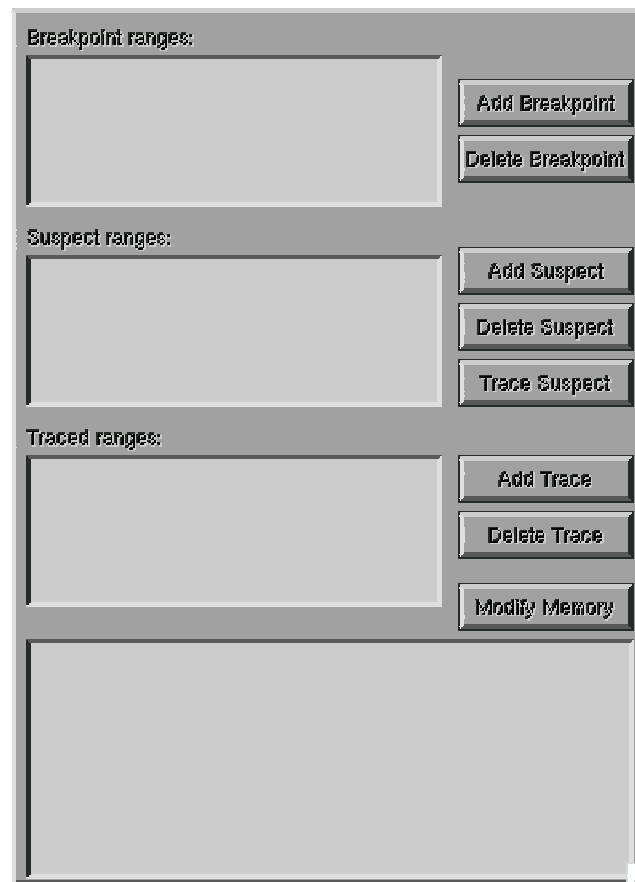


Figure 8.5: The UMPS Memory Browser Window

After the UMPS Main Window, the **Memory Browser** Window is where users

devote most of their attention since it is the most debugging-oriented tool available. This window has the following three purposes:

8.5.1 Breakpoint Ranges

Set or remove breakpoint ranges. A breakpoint range is a set of contiguous addresses located in either virtual or physical memory. If the range is located in virtual memory, an ASID must be associated with it. ASID=0x40 is used to designate a range in physical memory. Breakpoint ranges refer to code; that is the simulator stops before executing any instruction contained in a breakpoint range and the **Breakpoints** “Stop-on” button is on.

To set a breakpoint range associated with a function use the **Add Breakpoint’s Symbolic/Virtual** option - as opposed to its **Physical** option. This brings up the **Symbolic/Virtual Range Insert** Window which displays the contents of the symbol table. Simply double click on a function’s entry in the displayed table to automatically fill in the starting and ending addresses for this breakpoint range. The ASID field is always pre-filled with 0x40, designating a physical address range, and must be manually modified to designate a virtual address range. Each successfully added breakpoint range is identified by a breakpoint range description which can be found in the **Breakpoint ranges:** sub-window. This description will display a **PHYS** prefix for all physical breakpoint ranges.

The default behavior is for “step-in” debugging; that is the filled in starting and ending addresses for the breakpoint range are the starting and ending addresses for the indicated function. Hence the simulator will stop prior to each instruction in

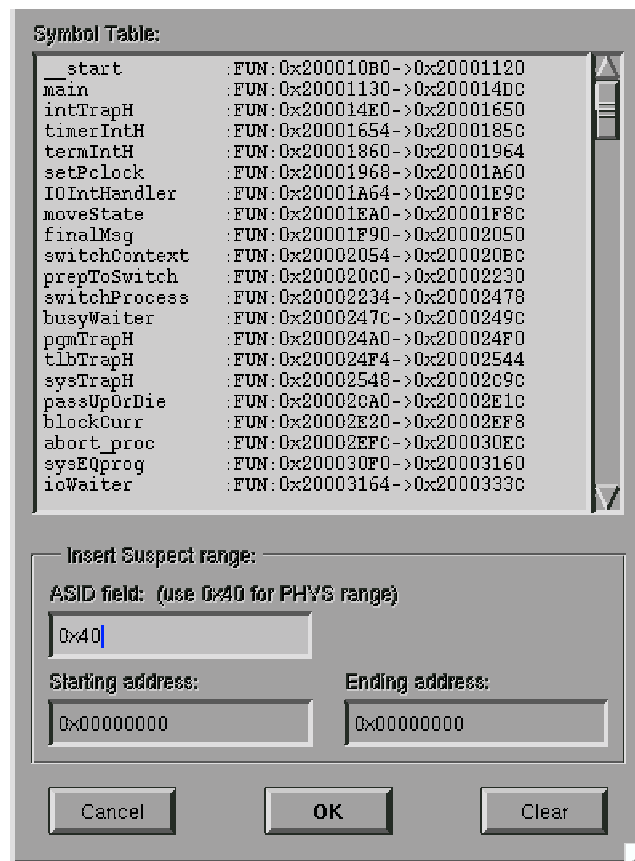


Figure 8.6: The UMPS Symbolic/Virtual Range Insert Window

the function as execution proceeds through the function; single-stepping through the function.

Alternatively, there is “step-over” debugging; the breakpoint range encompasses only a function’s entry point and not the whole function. To use “step-over” debugging double click on the function’s entry and then alter the ending address for the breakpoint range to be the same as the range’s starting address; a one word breakpoint range that only encompasses the first instruction of the selected func-

tion. Alternatively, as described in Section 8.6, one can use the **Symbol Table** Window to set a “step-over” breakpoint range that does not require any manual editing of the range’s ending address.

A breakpoint range is removed by either double clicking on the breakpoint range’s description in the **Breakpoint ranges:** sub-window, or by selecting the breakpoint range’s description and then clicking on the **Delete Breakpoint** button.

It is important to remember that breakpoint ranges only halt execution when the **Breakpoints** “Stop-on” button is on.

8.5.2 Suspect Ranges

Set or remove suspect ranges. A suspect range is for data objects what a breakpoint range is for code with a few differences. For suspect ranges the simulator stops after any instruction which accesses a memory location contained in a suspect range and the **Suspects** “Stop-on” button is on.

While suspect ranges are set and removed using a procedure identical to setting and removing breakpoint ranges, there are some differences. Whenever a suspect range is set, additionally one is prompted to indicate which type of memory access will cause a “Stop-on” event; read access only (**R**), write access only (**W**), or either (**RW**). Each successfully added suspect range is identified by a suspect range description which can be found in the **Suspect ranges:** sub-window. This description will display a **PHYS** prefix for all physical suspect ranges in addition to the appropriate **R** and **W** tags for that suspect range.

It is important to remember that suspect ranges only halt execution when the

Suspects “Stop-on” button is on.

8.5.3 Traced Ranges

Set or remove trace ranges. A trace range is a set of contiguous physical addresses. Trace ranges typically refer to data objects; that is the simulator displays the contents of the trace range in the memory dump sub-window at the bottom of the **Memory Browser** Window. Trace ranges are used to keep track of the contents of specified physical address ranges; trace ranges do not cause execution to stop. Nevertheless, the selected simulation speed determines the frequency at which the memory dump sub-window is updated.

To add a trace range one may:

- Explicitly enter the starting and ending physical addresses of the trace range, which one is prompted for by clicking on the **Add Trace** button. Typically one uses the **Symbol Table** Window to find the starting and ending physical addresses of data objects.
- Add a range you wish to trace as a suspect range, select the suspect range description and the click on the **Trace Suspect** button.

Trace ranges are removed using the same method as breakpoint and suspect ranges.

The memory-dump window has three columns:

- The address field: The address of the first (lower addressed) of the two words displayed on the line.

- The memory field: The contents of two words in physical memory. Regardless of the endian-ness of the host machine these two words are always displayed in big-endian format.
- The ASCII translation field: For all bytes whose values are between 32 and 127 the ASCII translation values are displayed. The two words displayed translates into a possible eight characters per line. Regardless of the endian-ness of the host machine these eight characters are always translated and displayed in little-endian format.

In addition to setting and removing trace ranges, one may also modify the contents of any word in RAM memory including the device registers. As with the display of memory locations, the input is always expected to be in big-endian format.

It is important to keep in mind that when tracing a memory location that will be changed by instruction execution and the simulation speed is set to **Normal** or below, the memory-dump sub-window will be updated on every write in the traced area

8.6 The Symbol Table Window

This window displays the contents of the .stab symbol table map file. A .stab entry contains:

- The symbolic name of the entry.

- A designation of the entry type; **FUN** for functions and **OBJ** for data objects.
- The starting and ending addresses for the entry.

The **Symbol Table** Window displays the .stab file's contents stable sorted by address (ascending) and then by entry type.

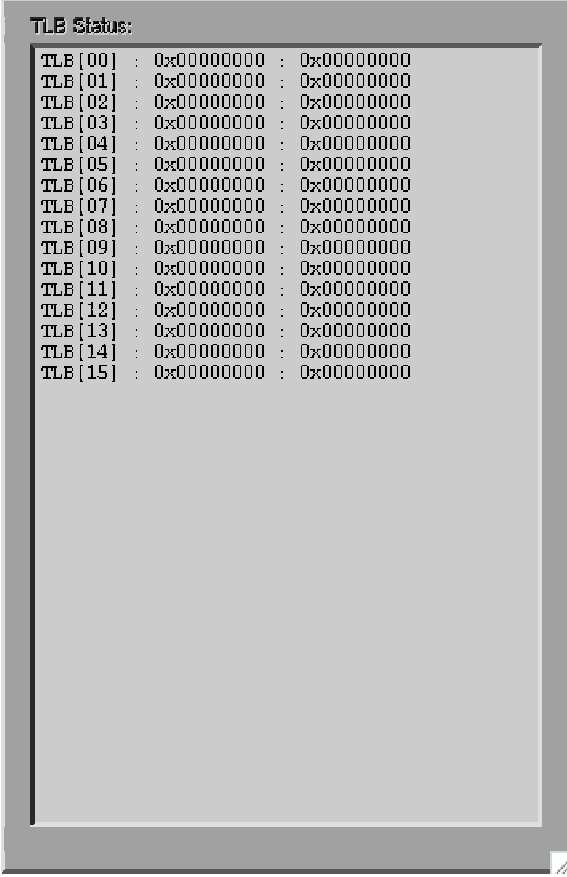
The symbol table also provides some short cuts for setting breakpoint and suspect ranges.

- Double clicking on a function entry automatically generates a “step-over” breakpoint range for that function, i.e. a one word breakpoint range that only encompasses the first instruction of the selected function.
- Double clicking on a data object generates a suspect range for that data object and also adds traces the suspect range; a memory trace for the range is automatically generated.

8.7 The TLB Display Window

This window displays the complete contents of the TLB, including each entry's **EntryHi** and **EntryLo** portions. Regardless of the endian-ness of the host machine the TLB is always displayed in big-endian format.

By double clicking on either portion, **EntryHi** or **EntryLo**, of a TLB entry one may enter a new value for that portion (in big-endian format).



The image shows a window titled "TLB Status:" containing a list of 16 TLB entries. Each entry is displayed as "TLB [index] : 0x00000000 : 0x00000000", where the index ranges from 00 to 15. The window has a standard graphical border with a title bar and a small icon in the bottom right corner.

TLB Entry	Value 1	Value 2
TLB [00]	0x00000000	0x00000000
TLB [01]	0x00000000	0x00000000
TLB [02]	0x00000000	0x00000000
TLB [03]	0x00000000	0x00000000
TLB [04]	0x00000000	0x00000000
TLB [05]	0x00000000	0x00000000
TLB [06]	0x00000000	0x00000000
TLB [07]	0x00000000	0x00000000
TLB [08]	0x00000000	0x00000000
TLB [09]	0x00000000	0x00000000
TLB [10]	0x00000000	0x00000000
TLB [11]	0x00000000	0x00000000
TLB [12]	0x00000000	0x00000000
TLB [13]	0x00000000	0x00000000
TLB [14]	0x00000000	0x00000000
TLB [15]	0x00000000	0x00000000

Figure 8.7: The UMPS TLB Display Window

8.8 Using The UMPS-MKDEV Device Creation Utility

While the log files for holding terminal and printer output are standard text files, and which if not present for any active printer or terminal, will be automatically created by UMPS at startup time, the disk and tape cartridge files must be explicitly created beforehand. One uses the UMPS-MKDEV device creation utility to create

the files that represent these persistent memory devices.

8.8.1 Creating Disk Devices

Disks in μ MPS are read/write sealed devices with specific performance figures. The UMPS-MKDEV utility allows one to create an **empty** disk only; this way an OS developer may elect any desired disk data organization.

The created “disk” file represents the entire disk contents, even when empty. Hence this file may be very large. It is recommended to create small disks which can be used to represent a little portion of an otherwise very large disk unit.

Disks are created via:

```
UMPS-MKDEV -D <diskfile.mps> [CYL [HEAD [SECT [RPM [SEEK  
[DATAS]]]]]]
```

where:

- -D instructs the utility to build a disk file image.
- **diskfile.mps** is the name of the disk file image to be created.
- The following six optional parameters allow one to set the drive’s geometry: number of cylinders, heads/surfaces, and sectors, and the drive’s performance statistics: the disk rotation speed in rotations per minute, the average cylinder-to-cylinder seek time, and the sector data occupancy percentage.

As with real disks, differing performance statistics result in differing simulated drive performance. e.g. A faster rotation speed results in less latency delay and a smaller sector data occupancy percentage results in shorter read/write times.

The default values for all these parameters are shown when entering the UMPS-MKDEV alone without any parameters.

8.8.2 Creating Tape Cartridges

Tape devices in μ MPS are read-only devices which are typically used for the fast loading of large quantities of data into the simulation without having to resort to typing the data directly into a terminal. Tapes are typically used to load user programs (U-proc's) as well as the OS/kernel itself.

A tape cartridge file image will contain a properly-formatted copy of the file(s) the user wishes loaded onto it.

Tape cartridge image files are created via:

```
UMPS-MKDEV -T <tapefile.mps> <file> [<file>] ... [<file>]
```

where:

- -T instructs the utility to build a tape cartridge file image.
- **tapefile.mps** is the name of the tape cartridge file image to be created.
- The concluding space-separated list of file names are the files that will be included on the tape cartridge file image. These files, of which there must be at least one, are .aout or .core formatted files. Each file will be zero-padded to a multiple of the 4KB blocksize and sliced up using the **EOB** and **EOF** block markers. The tape's end will be marked with a **EOT** marker.

Chapter 9

Debugging in μ MPS

As described in Section 7.2 writing code for an OS requires some special considerations. Debugging an OS, unfortunately, is even more challenging. In the authors' experience, most undergraduates, even when supplied with sophisticated debugging tools, primarily rely on output statements (e.g. `cout` or `printf`) for debugging. By examining the generated output stream, students infer both the flow of execution and the program state at each output statement. This can be called “debugging by side-effect.” When debugging an OS there is no support for output statements; at least not until the OS author has written and debugged support for them.¹

Debugging an OS is further complicated by its inherent interconnectedness; frustrating the desire to perform unit testing. One cannot test a scheduler with-

¹While Phase 1 of the Kaya project comes with its own very rudimentary support for terminal output, in Phase 2, successfully generating any terminal output represents the achievement of a major debugging milestone along the path towards the completion of that phase.

out support for timing services. One cannot test timing services without support for interrupt handling. One cannot test interrupt handling without support for semaphores and a scheduler.

The lack of students' traditional debugging tool, output statements, and the inability to do module testing due to an OS's interconnectedness presents a unique debugging challenge. It is important to start thinking about debugging, not in terms of side effects, but in terms of current program state. Unlike with traditional undergraduate programming projects, where it is possible to test all possible control paths and all meaningful program states, there are too many possible meaningful program states during the execution of an OS for exhaustive testing; at least within the constraints of a term-long undergraduate project. Nevertheless, by debugging with an emphasis on program state, instead of side effect, one can start to gain a degree of confidence regarding the correctness of the OS.

9.1 μ MPS Debugging Strategies

The μ MPS simulator, from one perspective, can be thought of as a sophisticated debugging tool/environment. As described in Chapter 8 it provides three primary mechanisms to assist in the debugging process; breakpoints, suspect ranges, and memory tracing. The following is a description of two debugging strategies.

9.1.1 Using a Character Buffer to Mimic `printf`

In the spirit of attempting to force a square peg into a round whole, it is possible to use a RAM buffer to behave like an output stream; allowing the use of the “familiar” debugging technique. To do this one declares a global character array and instead of issuing an output statement, one moves a character string or meaningful value into the buffer. The trace facility is then used to display the buffer’s contents. Running one’s OS while monitoring the contents of the buffer is isomorphic to running a traditional program and monitoring the output stream.

Writing to the buffer can be done in an accumulative fashion, similar to an output stream, or each line of “output” can overwrite the previous one.²

Under μ MPS one has the option to improve this approach by placing the buffer in the suspect list and enabling the simulator to halt on suspect matches. Now whenever an “output statement” is reached the simulator will stop, allowing for the examination, via the trace window, of the state of OS variables.

9.1.2 Implementing Debugging Functions

The above approach, while useful, has its limitations. There is no `itoa` (integer-to-ascii) function –unless you write your own– so one is limited, via the global buffer, to the display of character strings only. Also while program execution can be halted prior to each output message, only global variables can be examined via

²The test program that accompanies Phase 1 of the Kayaproject, in addition to generating output on `TERMINAL0` illustrating the test program’s progress, also illustrates this accumulative writing to a character buffer technique.

the trace window.

An improvement on this approach is to implement either a debug function, or a suite of such functions; e.g. `debugA`, `debugB`, `debugC`, etc. Each of these functions can be defined to accept four integer parameters. Now, at a point of desired program inspection, instead of generating an output string (e.g. “you are here”) one calls a debug function. In this scenario, the first parameter is usually a unique “key” value (e.g. 10, 20, 42, etc) that unambiguously identifies where in the program the function call statement is. The other three parameters can be used to pass along local function variables, global variables, expressions or any other value that will help the debugger understand the program state at that point in the program.

By setting a “step-over” breakpoint for each debug function (and enabling the simulator to halt on breakpoints), the simulator will stop on entry to each debug function. Furthermore, registers **a0**, **a1**, **a2**, and **a3** will contain the four parameters passed to the debug function. The contents of these registers are always displayed on the μ MPS simulator’s Main Window eliminating the need to use the trace window to display OS state information. Furthermore, unlike the small trace window which always displays all the traced memory ranges, with a debug function one can elect which variables to inspect on a call statement by call statement basis. True, one is limited to only three values, but the trace window is still available to display additional information.

Using a suite of debug functions allows for a greater degree of debugging sophistication. For example `debugA` can be used for scheduling issues, while

`debugB` can be used interrupt handling. One doesn't wish to step over n breakpoints related to scheduling while endeavoring to get to a breakpoint related to interrupt handling; just enable the `debugB` breakpoint. A suite of debug functions can also help in the following scenario: one suspects that the Ready Queue is somehow getting corrupted, but only after the first "warm" page fault. Enabling a debug function, say in the scheduler, is inefficient. There will be hundreds of scheduler breakpoints that will occur prior to the one in question. Instead enable a different debug function in the pager. When that breakpoint occurs, then enable the debug function in the scheduler. Thus one has the ability to enable a breakpoint in a frequently occurring location only after some epoch has occurred, instead of the breakpoint being enabled from OS boot-time.

9.2 Common Pitfalls to Watch Out For

While every OS author seems to generate their own unique errors, and concomitant debugging challenges, a number of errors do seem to reoccur with regularity. The following is a list of some of the more difficult ones to track down. By enumerating them here, it is hoped to save some lucky OS authors from some long and frustrating debugging sessions. Check the μ MPS Web site for possible additions to this list.

9.2.1 Errors in Syntax

There is not much one can do for a logic error except track it down and fix it. Yet sometimes the logic appears flawless and the code still does not work as expected. This may be due to a syntax error. Some of the structures in an OS can be quite complex; an array of structures, where each structure contains arrays of processor states, each of which in turn contains an array, arrays of PTE's and other data, all of which is accessed through a pointer. While the syntax used to access some value deep in the structure may compile and even run, it can nevertheless be incorrect. It is recommended that by using a debug function to display some appropriate value deep within the structure, one verify that one's syntax is indeed correct. Even the most experienced of programmers can make a syntax error when mixing together structures, arrays, structures of arrays, arrays of structures, dot notation, and pointer notation.

9.2.2 Errors in Structure Initialization

Errors in initialization are also quite common. Most programmers have grown used to an environment where uninitialized variables are “zeroed” out. This is even true of the μ MPS cross platform development tools; the **.bss** area for **.core** files is explicitly included in the **.core** file and zeroed out. While the initial values for **.bss** kernel/OS variables and structures is zero, many of these structures get used and re-used over and over. Kernel maintained Process-Blocks are the canonical example. It is important to remember to initialize all of such a structure's

fields prior to re-use. Not doing so can make an uninitialized value incorrectly appear to have been initialized.³

9.2.3 Overlapping Stack Spaces and Other Program Components

The OS data for one U-proc must be kept separate from the OS data for other U-proc's. This is rather easy with respect to each U-proc's virtual address space through the magic of virtual memory. The OS structures that reside in ksegOS for each U-proc are a different matter. Therefore care must be taken to insure that the OS's data structures for each U-proc (which may include one or more stack areas in addition to a PgTbl) are both large enough and completely disjoint. Given the very difficult nature of debugging overlapping stack spaces, it is recommended that this be considered whenever one's OS behaves in an unpredictable and erratic manner.

9.2.4 Compiler Anomalies

As outlined in Section 7.2 the supplied cross-compiler, even when instructed to behave as conservative as possible, will both reorder one's code and cache frequently used variables. This is especially dangerous when dealing with hardware defined locations –which for compiler-related safety reasons should always be

³One example of this in the Kaya project is with the SYS5 pointer fields in a ProcBlk. If they are not re-set to NULL upon ProcBlk re-use then when the next user of the ProcBlk attempts to issue its first SYS5 it will appear as a duplicate SYS5 attempt and trigger the process termination routine.

accessed through pointers.

One reasonably consistent, though not surefire way to determine if correct code is being altered into incorrect code by the compiler is through the use of debug functions. Specifically when code runs correctly when “littered” with debug function calls, and runs incorrectly when they are removed, one is probably dealing with the compiler code reordering/variable caching problem. As one can imagine it is quite frustrating for a student to believe they have successfully completed phase *i* of their OS project only to remove all their debug function calls and learn their OS no longer behaves the same.

A (debug) function call is a compiler epoch or bottleneck. A compiler cannot reorder assembler statements that occur after a function call to before it, or visa versa. Also any register-cached variables must be restored to memory prior to the function call. Function calls force a compiler, regardless of the optimization it is performing, to synchronize the generated code with the original source code.

There are a number of fixes one might try when this occurs:

- Do nothing. The additional debug function calls merely slows down the OS, but does not affect its correctness.
- Try all of the options described in Section 7.2. That is use pointers to access hardware defined locations and use the `volatile` keyword on appropriate variables and structures.

Bibliography

- [1] AMDAHL, G., BLAAUW, G., AND BROOKS, F. Architecture of the IBM system/360. *IBM Journal of Research and Development* (1964).
- [2] KANE, G., AND HEINRICH, J. *MIPS RISC Architecture*. Prentice Hall, 1992.