SPRING KERNEL
VERSION 2 OS PRIMITIVES.

Lory Molesky
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Working Document

ABSTRACT

This report documents the current status of the Spring Kernel Version 2 development activity, and provides a framework for near term development. This summary represents work of this author and Max Kuan, Decao Mao, and Douglas Niehaus.

1 Introduction

This report documents the current status of the Spring Kernel Version 2 development activity, and provides a framework for near term development. Herein we focus on the status and ongoing development of Spring V2 as related to OS primitives. Discussions about Spring IPC primitives are not included in this document, except for necessary interface descriptions.

Spring V1 was written on top of VRTX, without using VRTX scheduling. The main features of V1 included support for the heuristic guarantee on a multiprocessor, where the SP scheduler and AP dispatcher were the main support components. Spring V2 is the first kernel version that has completely discarded VRTX. The support of Logical Memory is a key feature of this activity.

The discussion begins by focusing on processes and their relation to system resources. Of primary importance is the context in which process creation occurs. Process creation can occur at *boot* time, *mode change*, or *dynamically*. The support of these multiple contexts of process creation is a driving factor in the current stage of the Spring V2 design, the *system initialization and boot* stage.

Section 3 sketches the process related primitives. Section 4 details the MMU related primitives. Sections 5 and 6 discuss traps and interrupts used by Spring V2. Section 7 discusses system configuration related to the kernel.

2 An Overview of Process Related Activities

The process subsystem is closely related to the system boot activity. The intent of this design is to both allow for as dynamic a system as possible, yet relegate operations which are difficult to quantify to the mode change segment of the system. To this end, we separate the creation of resources required by processes from the creation of processes. Resource creation is restricted to a mode change; process creation can optionally be performed during a mode change, or dynamically (concurrently with the scheduling and execution of application tasks).

2.1 Resource Creation

One of the major components of system initialization is resource creation. At this time, the following resources (in the form of pools of objects) are created at system initialization:

- PCB Pool
- Physical Page Pool
- Memory Map Pool
- IPC Buffer Pool

- IPC Port Pool

A simple approach to implementing page resource pools follows: Page pools will be implemented an array of entries, each entry representing a reference count which indicates how many processes are currently using the page. A free list contains all the entries which have not been allocated. Allocation of a *generic* page will consist of removing the first entry of the free list, returning the address this page. When a process no longer needs to use a page, the reference count of that page will be decremented; if the reference count becomes zero, the page will be inserted into the free list. Direct access can be achieved by computing a function on the *page id*, the page id being in the range of 0 to the number of system pages minus 1.

Using an 8K page size, we have $\frac{4meg}{8K} = 500$ pages on a 136 system. Making the field width of the entries (containing the reference count) 8 bits will require 500 bytes of memory, which seems reasonable.

The following kernel resource management primitives are envisioned:

- pcb_alloc()
- pcb_free()
- physical_page_alloc()
- physical_page_free()
- memory_map_alloc()
- memory_map_free()

2.2 Process Creation

Once the resource pools have been created, process creation can commence. Process creation could occur at any one of three system scenarios: system init, mode change, or dynamically. In contrast to process creation at system initialization time, both mode change and dynamic process creation require some *preservation of system state.* We thus view process creation as an *incremental* system activity, i.e., an activity that can add another process to the system without interfering with existing processes or data structures. Thus, process creation at system initialization is modeled identically to process creation at mode change or dynamically.

The `create_process` primitive interacts with the Spring file system in order to read an executable file describing a process. The following steps are required to create a process [3]:

- Allocate the PCB and Memory Map

  - Link Map to Control Block

- $\forall$ Logical Segments in the header (including the stack)
  - Allocate a physical page (from the physical page pool)
  - Map the logical page to the physical page

- $\forall$ Segments that have an initial image
  - Seek to the start of the image
  - Read the image into memory page by page

This process creation skeleton represents process creation in its most basic form – additional operations are required for processes which share data, and thus data pages. This data sharing was omitted for clarity of illustration.

2.3 Resource Creation Revisited

In a dynamic process creation model, it behooves us to enable both system and application processors to perform process creation. To achieve this end, resource pools are allocated on all processors. Of course, successful process creation is contingent on having enough resources available in the pools to accommodate this activity.

This implementation activity is ongoing.

3 Process Subsystem

- `create_process` – Create Process
  Creates and loads a process including getting the image and setting up system data structures such as memory maps, PCB, etc. By default one instance is activated and it is automatically ready to be invoked. In the current version, `create_process` is callable at system initialization and at mode changes. The current kernel design has hooks for a dynamic version of `create_process`; these hooks include the interactivity with the spring file system and the support of resource pools on all processors.

- `create_instance` Create (or Activate) Process Instance
  Additional instances of processes will faster to create than the original instance, since some of the segments can be shared. The text (process code) segment and the shared data segment can be shared by all additional process instances.

- `delete` – Delete Process
  Removes a process completely including all instances. All allocated resources must be returned

to the pools. These resources include all segments, e.g. pages for text, data, and stack, memory maps, and the PCB.

- `delete_instance` Delete (or Deactivate) Process Instance
  Removes the specified instance. By implication, process instances should be named. All shared pages will have their reference count decremented during `delete_instance`.

- `suspend` – Suspend Process
  Place a process on the wait queue; only a non-HRT task can suspend.

- `resume` Resume Process
  Take a suspended process off the queue.

- `delay` – Delay Process

- `enquire` – Get Process State
  Enquire about any state information in the PCB; and perhaps elsewhere.

- `set` – Set Process State
  Set state information in the PCB; and perhaps elsewhere.

4 MMU Primitives

In order to support dynamic task activation and provide protection (between text and data space, and between Kernel and User space), Spring uses *logical* addressing. Prior to execution, application tasks are loaded into system memory, generally at boot time or mode change time. To achieve predictability, yet provide fast address translations, the Spring approach is to use the MC68851's address translation cache (ATC), but never allow page faults. This involves preloading the ATC with the appropriate logical to physical address for both the kernel and the application task prior to runtime. Part of the ATC is dedicated to the kernel address space, while the remaining portion is dedicated to application task address space.

The MC68851's ATC is a fully-associative 64-entry cache memory. In a conventional design, a translation table is used in conjunction with this cache; when a cache-miss occurs, the logical-to-physical mapping is *faulted in.* The Spring approach uses a the translation table, but only for preloading of the logical-to-physical mappings into the ATC. There can be more than one translation table, this is specified by the *map_addr* parameter required for many of the MMU calls described in this section.

The current design of the Spring Kernel uses a two-level address translation (address-mapping) table. A one-level table has the property that the *entire* table must reside in the system's memory . Thus there is no way to indicate that a portion of the address space is not (currently) mapped except by having an entry in the table indicating this [1]. A three level table requires more time to search, but few virtual addresses are large enough to require this, so a two-level table was decided on.

In a two level map, the first level tables are referred to as *pointer tables*, while the second are called *page tables.* An entry in a page table is called a *page frame.*

4.1 The Chip Interface

The MMU chip (MC68851) provides an instruction set as a collection of assembler routines. For various reasons, a C interface has been constructed for each of these routines.

The most fundamental instructions are *load* and *flush.* A `load` instruction will search the translation tables and load the ATC for the specified logical address. The `flush` instructions enable the programmer to remove entries from the ATC. A single entry can be flushed, specified by a combination of the logical address and function code. Multiple ATC entries can be flushed with the assembler `pflusha` (all), `pflushr` (all by root pointer), and `pflushs` (all globally shared).

The following C routines have been defined, and generally are directly analogous to the assembler routines.

- `mmu_crp(rp_reg *, int)` Read/write the PMMU CPU Root Pointer Register
- `mmu_srp(rp_reg *, int)` Read/write the PMMU System Root Pointer Register
- `mmu_drp(rp_reg *, int)` Read/write the PMMU DMA Root Pointer Register
- `mmu_ps(ps_reg *, int)` Read/write the PMMU Status Register
- `mmu_pcs(pcs_reg *, int)` Read/write the PMMU Cache Status Register
- `mmu_tc(tc_reg *, int)` Read/write the PMMU Translation Control Register
- `mmu_ac(ac_reg *, int)` Read/write the PMMU Access Control Register
- `mmu_flusha(void)` Flush the entire PMMU's Address Translation Cache
- `mmu_flush_task(rp_reg *)` Flush the ATC entries associated with a particular
- `mmu_load(int func-code, addr, int R/W)` Load an entry into the ATC
- `mmu_flush_shared_entry(int func-code, addr)` Flush and entry from the TLB
- `mm_moves_write_proc_mem(char *sys_addr, char *proc_addr, int size)` Write current proc memory using MOVES
- `mm_moves_read_proc_mem(char *sys_addr, char *proc_addr, int size)` Read current proc memory using MOVES
- mmu_ptest_level0() Determine the status of the ATC.
- mmu_ptest_level1() Determine the status of the first level of the map.

4.2 High Level MMU Primitives

This subsection describes the high level primitives used to program the MMU. These routines have many parameters in common; following parameters are among the most common.

- *mode* – Either KERNEL or USER.
- **map_addr* – A pointer to the address translation table.
- *page_type* – Either CODE_PAGE, RW_DATA_PAGE, or RO_DATA_PAGE.
- **log_addr* – A pointer to a logical address.
- **phys_addr* – A pointer to a physical address.

Most of these primitives have a logical address as an input parameter. By searching the translation table, the page descriptor can be located using the logical address as a search key. Once the page descriptor is found, the physical address can be replaced, or other attributes of the page descriptor can be altered. For example, the `find_page_desc` call is used to locate a page descriptor from the logical address. The `init_map` call alters the physical address of the located page descriptor. The `set_page_attr()` call alters other attributes of the located page descriptor, including the *write-protect* bit and the *modified* bit.

- `init_map`
  init_map(*map_addr, mode)
  Initialize the data structure that will be used as an address translation map. The data structure contains a sub-structure for use as the root pointer, an array of table descriptors (which constitute the map's first-level table) and arrays of page descriptors (which constitute the map's second level tables). The initialization work involves linking the tables together and setting up most of the fields in the page descriptors. The physical address field is left empty.

- `map_size`
  map_size(void)
  Determine the amount of memory (in bytes) required to store an address translation map (including those bytes wasted for alignment purpose). This is essentially a product of TIA and TIB, the first two levels' translation indices.

- `find_page_desc`
  find_page_desc(*map_addr, *log_addr)
  This primitive locates the input logical address's corresponding page descriptor within an address translation map. This primitive is exclusively and frequently called by other memory

management routines.
Return Value: 0 if the logical address is invalid, otherwise the address of the page descriptor is returned.

- `set_page_attr`
  set_page_attr(*map_addr, *log_addr, attr, new_value)
  This primitive selectively modifies fields of a page descriptor. As in many other memory management routines, by using the logical address parameter, the page descriptor is located. Seven fields can be modified by this function, the relevant ones are the Lock field, the Modified field, the Used field, and the WP (write protect) field. These fields can have only two values: 0 or 1.

  Parameters: attr: a bit field showing which attributes to be changed
  new_value: set the attribute to this value
  Return Value: -1 if log_addr is out of range.

- `mmu_read_atc`
  mmu_read_atc(*map_addr, mode, *map)
  Read the contents of the ATC, returning them in map. *Parameters:* map: A pointer to a structure to be returned by `read_atc`.

  `map_page`
  map_page(*map_addr, *log_addr, *phys_addr, page_type)
  Using the logical address parameter, the page descriptor is located and attributes of the page desciptor are modified. These attributes include the physical page address (page frame) and the write-protect and modified fields. The write-protect and modified fields are set corresponding to the page_type parameter, and various combinations of the two fields define three page_types - a CODE_PAGE, RW_DATA_PAGE, or a RO_DATA_PAGE.
  Return Value: -1: if the logical address is out of range

- `unmap_page`
  unmap_page(*map_addr, *log_addr)
  Find a logical address's page descriptor and mark it as invalid.
  Return value: -1: if the logical address is out of range

- `phys_to_log`
  phys_to_log(*map_addr, *phys_addr, *log_addr)
  Find a physical address's logical counterpart

- `log_to_phys`
  log_to_phys(*map_addr, *log_addr, *phys_addr)

Find a logical address's physical counterpart.
Return Value: -1: if a valid translation is not found

- `load_map`
  load_map(*map_addr, mode)
  This primitive loads the memory map into the ATC by performing the following steps.
  - First, the MMU"s root pointer register is set (according to the mode, KERNEL or USER).
  - Next, all first-level table descriptors are marked as valid.
  - For every valid page descriptor, the page descriptor is loaded into the ATC. This load is accomplished with the MMU's pload instruction.
  - Finally, the first-level table descriptors are reset to valid. This has the effect of detecting all invalid page references.

  This function will also set up the CRP register for the user task.

- `config_mmu`
  config_mmu(void)
  Configure the 68851 PMMU and enable it. This function utilizes the macros PS, TIA, TIB to the corresponding fields in the PMMU's translation control register. The SRE field is always set to 1 (i.e., the SRP register is used for kernel space) and the FCL field is always set to 0 (i.e., function code lookup is disabled).

- `make_valid(*map_addr)` Allow accesses to the translation table by the MMU. After an ATC cache miss, the address translation will be accessed to fault an address into the ATC.

- `make_invalid(*map_addr)` Do not allow accesses to the translation table by the MMU. Causes a bus error when an ATC cache miss occurs.

- `is_code_page(*map_addr,*log_addr)` Return 1 if the page is a code page. This call is used by the debugger to remap a page from code to data an back when writing (and curiously enough *reading*) breakpoints.

- `mmu_reload_page(*map_addr,mode,*log_addr,page_type)` This function serves two purposes, to dynamically load a single page into the ATC, and to alter the page type of a page in the ATC. The init code performs dynamic loading of pages when setting up shared memory segments. The debugger requires the ability to modify a page type when a code page needs to be modified (for setting breakpoints).

5 Soft Real-Time Primitives

This section describes system calls that are *not* hard real-time. Since there is no direct mechanism to access the system disk from the 136A board, disk access routines have been implemented indirectly via calls to a server on the 147 (Unix) board. These *Spring File System* primitives are used primarily in the initialization sequence (to read files into 136 memory) and at runtime for performing debugging (for writing 136 memory to files).

5.1 Spring File System

The Spring File System allows programs on the Motorola 136A boards to access files on the Motorola 147 boards. The set of routines described below are the duals of the standard UNIX System V routines, open(2), close(2), lseek(2), read(2) and write(2). The only divergence between the routines is between open(2) and spring_open(); spring_open expects a relative path name, since all path names passed to spring_open() are prepended with the /spring/fs directory.

- `#include<sys/spr_fcntl.h>`
- `spring_open(path,oflag,mode)`
- `spring_close(fildes)`
- `spring_lseek(fildes,offset,whence)`
- `spring_read(fildes,buf,nbyte)`
- `spring_write(fildes,buf,nbyte)`

The Spring File System is implemented as part of the Spring Debugger (sbug(l)). The interface calls are part of the 136 monitor (dbg_c_monitor.c), which is normally linked with the kernel one is building. The 147 debug monitor server (dbg_mon), implements the 147 end of the Spring File System protocol. Thus, in order to use the Spring File System, one must link the kernel with the Spring Debugger, and run the kernel in the context of the debugger.

Since these Spring File System calls wind up executing code on the 147 running generic System V Unix, little can be quantified about their worst case execution times. These calls are thus classified as *soft (non ?)* real-time, and their use should be restricted to the initialization or mode change phase of system operation.

5.2 Other Soft Real-Time Primitives

6 Traps

There are 15 different user definable (Spring is the user in this context) traps. The other 14 traps are unused at this time.

- `trap 3` The debugger trap.
  A breakpoint is inserted at address *addr* into program text by writing a trap 3 instruction at *addr*.

7 Use of Interrupt Vectors assigned by the Motorola 136A Board

The following interrupt vectors are defined by the 136A board. Complete descriptions are listed on pages 4-15 and 4-16 of [2].

- `ABORT* (0x40)` The interrupt caused by the abort pushbutton.
- `ACFIRQ* (0x41)` AC power fail interrupt.
- `TMRIRQ* (0x42)` The Z8036 timer interrupt.
- `SFIRQ* (0x43)` VMEbus system fail interrupt.
- `SIOIRQ* (0x44)` Serial I/O interrupt.
- `SIGHP* (0x45)` MPCSR High Priority interrupt.
- `VSBIRQ* (0x47)` VSB bus interrupt. Not used.
- `LMIRQ* (0x4a)` Location Monitor interrupt.
  This is the interrupt that is generated in the event that a VME bus broadcast was generated.
- `SIGLP* (0x4b)` MPCSR Low Priority interrupt.

The interrupts currently in use relate to the clock and serial I/O.

- Printing to the Console
  A character interface has been implemented to write to the console. When a putc() system call is executed, the following operations occur internally to the kernel. The character is inserted into a circular *put* buffer in kernel memory. (If the put buffer is full, operations are delayed until the buffer is non-full). Next, the character is transmitted (asynchronously) to the MC68681 chip's output buffer, and control returns. Once the MC68681 has successfully received this character, it generates the `SIOIRQ*` interrupt. The interrupt handler for `SIOIRQ*` transmits the character from the MC68681 to the serial I/O (console) port.
- Maintaining a Clock
  The Z8036 chip contains three 16-bit countdown timers. These countdown timers have been programmed to generate a periodic (TMRIRQ*) interrupt every 1 ms. (The length of this period can be easily reprogrammed). When this interrupt occurs, the SP board updates a *software* clock on its own board, and on every application board. This software clock is a just a memory location local to each board. The software clock is thus a global view of time, accurate to 1 ms., which is available on every board via local (as opposed to VME) memory access.

page number	*logical address*	*physical address*	*description*
0x0	0x0	0x0	EVT, Interrupt Handlers
0x1	0x2000	0x7e0000[1]	147/136 Communication
0x2 – 0x7	0x4000 – 0xe000	–	Unused
0x8	0x10000	0xfffb0000[1]	MVME Short I/O Space
0x9	0x12000	0xffff0000[1]	MVME Short I/O Space
0xa – 0xe	0x14000 – 0x1c000	–	Unused
0xf	0x1e000	0x1e000	System Stack
0x10 – $\frac{_etext-}{0x2000}^2$	0x20000 – _etext-	0x20000 – _etext-	System Text
$\frac{_etext}{0x2000}$ – $\frac{_end-}{0x2000}^2$	_etext – _end-	_etext – _end-	System Data
$\frac{_end}{0x2000}$ – $\frac{_eshared-}{0x2000}^2$	_end – _eshared-	_end – _eshared-	System Shared Data

Figure 1: The System Processor (SP) Memory Map)

8 Kernel Memory Maps

8.1 Protection via Separate Pages for System Text and Data

By separating the system text and data regions such that no page contains both system text and data, protection can be achieved by the appropriate MMU configuration. System code and data can be protected from inadvertent dynamic data overruns. Catching these errors at runtime when the occur simplifies the debugging and testing of the system.

8.2 Sharing Data between the SP and AP's

In order to support independent scheduling and dispatching of application tasks, the Spring Kernel used shared data structures for communication [4]. Among these data structures are the STT (System Task Table), pointers to the STT, scalar variables used in resource reclaiming, and data structures used for the runtime verification of the schedule and resource usage. The features of and the techniques used for supporting the kernel shared data in the context of logical memory follow.

- Separate Pages (segment) for Shared Data - The ability to begin a shared data segment at a page boundary facilitates data sharing across processors by enabling each processor to map

[0] This address is out of the range managed by the physical pool.

[1] _etext, _end, and _eshared are symbols generated by spr_ld. _etext, _end, and _eshared denote the end (address) of the text, data, and shared data segments respectively. The table assumes that _etext, _end, and _eshared are on page boundaries. The symbol _etext- is used as shorthand notation for the last page of the text segment (_etext - 0x2000). Similarly definitions hold for _end- and _eshared-.

a logical page to a unique physical pages.

- Transparent Reconfiguration - Growth or shrinkage of System code and/or data does not interfere with the mechanism for sharing data. As System code and data grow, the shared data segment is transparently relocated by the Spring compilation environment.

Configuring the shared data region between the SP and the AP involves compile time, load time, and boot time actions. At compile time, space is allocated for the shared data, in the logical address space of the kernel image. At load time, logical addresses are assigned to all shared variables. At boot time, the SP maps the shared data logical address range to a (dynamically defined) physical address range. After determining the base physical address, the SP transmits this page frame to all the AP's. Finally, the AP's map the logical pages comprising the shared data area to the contiguous page frames starting at the transmitted base physical address. For reasons having to do with indirect access to data (i.e. pointer variables), the logical address range on the AP's must be identical to the logical address range on the SP.

Note that the SP and AP kernels *are* different. Although there is a large amount of common code, there are differences, one being that the SP supports scheduling and the AP's do not. This implies different system text and system data region sizes, an important fact to keep in mind when designing mechanism for sharing data between the SP and the AP's. In our implementation, we assume that the SP kernel + data size is less than the AP kernel + data size. Thus, the SP kernel loads first, determining the logical address range for the shared data. Next the AP kernels load, using the logical address range defined by the SP for their shared address range.

Most Unix loaders do not support the notion of a separate segment for shared data. However, conventional Unix loaders do provide support for aligning the data segment on the first page past the text segment. In order to support the notion of a separate segment for shared data, the GNU loader was modified. By convention, all (C source) variables which begin with a "$", i.e. $Stt1, are designated as shared variables. The modified loader recognizes these symbols, assigning absolute values to them beginning at the end of the ordinary data section. In addition to the standard auxiliary symbols output by the loader (_etext, _edata, and _end), the modified loader outputs another auxiliary symbol, _eshared, which is the address of the end of the shared data section. These symbols are used by the kernel at runtime when mapping the text, data, and shared data sections.

In addition to the ability to construct a separate segment for shared data, loader extensions to handle the AP kernel involve the specification of the starting address of the shared data section. This specification is needed in order to match the logical address range of the AP shared data segment to the logical address range of the SP shared data segment.

The C source code for the shared data definitions is kept in a single .c file. This allows both the owner code and code referencing it to compile in this code in order to have space allocated for it. If a processor maps shared data into its own physical memory, it is termed the owner of the data.

At boot time the owning code maps it to physical locations on its board, and transmits the base of this physical location to all other boards referencing this code. The boards referencing the kernel shared data, the AP's, map the logical address range defined at compile time to the physical range starting at this transmitted base physical location.

9 System Configuration

9.1 Distinguishing between the 8 and 16 Meg. 147's

In order to communicate between the 147 and 136 using shared memory (as is required in the Spring debugger), an area of physical memory has been reserved on the 147 using the sysgen(8) Unix system configuration utility. This area is required to be at the end of the physical memory. Currently, we have 147 systems with both 8 and 16 megabytes. The Spring debugger must know which type of system it is operating on in order to access the appropriate physical communication area.

This system heterogeneity is further complicated by the use of logical memory on the 136 boards. An example best illustrates the subtleties involved. On an 8 meg. system, the communication buffer starts at 0x7e0000. Referring directly to this address on the 136 is fine until the MMU is turned on. At that point, 0x7e0000 would need to be mapped. However, we do not wish to waste an entire page mapping this address; we subsequently added another level of indirection whenever referencing the communication buffer to solve this problem. A word in kernel memory is reserved to store the address of the communication area, this address happens to be 0x1ff8, and is called COMM_147_BASE. At boot time, the kernel initializes COMM_147_BASE with the default identifier, indicating an 8 meg. machine. When the debugger is instantiated, prior to 147/136 communication, the debugger will alter the default if appropriate (if we are running on the 16 meg. machine). This scenario as implemented follows:

```
#define COMM\_147\_BASE 0x1ff8
#define COMM\_147\_LOG 0x78000
#define BASE\_8MB 0X7e0000
#define BASE\_16MB 0Xfe0000

int *Comm\_147\_base = (int *) COMM\_147\_BASE;

init()
{
     ...
    *Comm\_147\_base = (int *) BASE\_8MB;
    *Sys\_type = TYPE\_8MB;
```

```
    ...
}

mm\_map()
{
    ...
    *Comm\_147\_base = COMM\_147\_LOG;

    if (Sys\_type == TYPE\_8MB)
       map\_page(map,*COMM\_147\_LOG,*BASE\_8MB)
    else
       map\_page(map,*COMM\_147\_LOG,*BASE\_16MB)
    ...
}
```

The pointer variable Comm_147_base points to address 0x1ff8. Prior to turning the MMU on, the contents of this pointer is 0x7e0000. After the MMU is turned on, the contents of the pointer is changed to the new, logical address of 0x7e0000 (for an 8 meg machine), 0x78000. The mapping of logical 0x78000 to physical 0x7e000 is performed with the map_page() routine.

9.2 Mapping "Memory Mapped" I/O Locations

136 access to peripheral devices is memory mapped. This means that, from the programmers perspective, peripheral registers are addressed just as any other memory location are addressed. For example, to access the current value of the z8036 timer register merely requires reading a memory address; this address happens to be 0xfffb0013.

When using logical memory addressing, these memory mapped locations must be mapped by the MMU, just as any other memory addresses. Given that the logical address space for the kernel is from 0 - 0x80000, the page starting at 0x10000 was reserved for mapping this memory mapped I/O space. For example, during logical addressing, the logical address 0x10013 would be used to access the z8036 timer register. The MMU would perform the translation from logical address 0x10013 to physical address 0xfffb0013.

Normally, the memory mapped constants are implemented by define constants. This method is find so long as the memory mapped peripherals are accessed in either physical or logical mode, but not both. In the Spring kernel, most memory mapped peripherals are only accessed in logical mode. However, some require access in both modes. For these peripherals, the defined constant is the physical address. A logical address is formed from the physical address by adding (*by hand*) a constant in the program code.

Appendix

A

This table was constructed by combining information in page 6-5 of [1], and page 4-15 of [2]. Of particular interest are the "user defined vectors", numbers 63 - 255. A number of these have been assigned to 136 hardware functions, such as the broadcast interrupt ($LMIRQ^*$) and the abort switch ($ABORT^*$) located on the front panel of each 136 board.

vector number		*vector offset*	
decimal	*hex*	*(hex)*	*Assignment*
0	0	000	Reset: Initial Interrupt Stack Pointer
1	1	004	Reset: Initial Program Counter
2	2	008	Bus Error
3	3	00c	Address Error
4	4	010	Illegal Instruction
5	5	014	Zero Divide
6	6	018	CHK, CHK2 Instruction
7	7	01c	cpTRAPcc, TRAPcc, TRAPV Instructions
8	8	020	Privilege Violation
9	9	024	Trace
10	a	028	Line 1010 Emulator
11	b	02c	Line 1111 Emulator
12	c	030	(Unassigned, Reserved)
13	d	034	Coprocessor Protocol Violation
14	e	038	Format Error
15	f	03c	Uninitialized Interrupt
16	10	040	
through			(Unassigned, Reserved)
23	17	05c	
24	18	060	Spurious Interrupt
25	19	064	
through			Levels 1 - 7 Interrupt Auto Vectors
31	1f	07c	
32	20	080	
through			Trap 0 to 15
47	2f	0bc	
48	30	0c0	
through			(Unassigned, Reserved)
63	3f	0fc	
64	40	100	$ABORT^*$
65	41	104	$ACFIRQ^*$
66	42	108	$TMRIRQ^*$
67	43	10c	$SFIRQ^*$
68	44	110	$SIOIRQ^*$
69	45	114	$SIGHP^*$
70	46	118	(Unassigned)
71	47	11c	$VSBIRQ^*$
72	48	120	(Unassigned)
73	49	124	(Unassigned)
74	4a	128	$LMIRQ^*$
75	4b	12c	$SIGLP^*$
76	4c	130	
through			Remaining User defined vectors (192)
255	ff	3fc	

Figure 2: The EVT (Exception Vector Table)

References

[1] Motorola Inc. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1985.

[2] Motorola Inc. *MVME135, MVME135-1, MVME135A, MVME136, and MVME136A 32-Bit Microcomputers User's Manual.* Motorola Inc., 1989.

[3] Douglas Niehaus. Process Activation Scenario and Commentary. *Spring Internal Documentation*, October 3 1990.

[4] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource Reclaiming in Multiprocessor Real-Time Systems . *submitted to Transactions on Parallel and Distributed Systems*, Feb. 1991.