HeatheRTOS Kernel

Matt Thiffault (matt.thiffault@gmail.com)

Kernel Description:

Overview:

The structure of the kernel code is relatively simple. First, hardware initialization code is called. Then the kernel initialization code is called. This initializes all kernel data structures. Then the first user task is created (metadata structures and stack space initialized). The task creation code also adds this task to the ready queue for it's priority level.

The kernel then enters a loop which only breaks when all tasks have exited or the Shutdown system call is invoked. This loop calls the scheduling code to determine the next task to run and context switches into that task. The context switch function returns when the kernel is re-entered due to an exception. The value it returns reflects whether the exception was a hardware or software interrupt (or undefined if using VFP). Then the appropriate code is called to handle the exception and the loop begins again.

Data Structures:

Task Descriptors – These are structures containing the following fields:

- regs: A 4 byte pointer to the task state stored on it's stack. The pointer type is a structure which matches the layout of the saved registers in memory, which allows for easy access. This is the first member of the task descriptor struct so that a pointer to a task descriptor is a pointer to this value.

- state_prio: An 8 bit field which holds the task state as the high four bits and the task priority as the low four bits. There are macros to get/set these fields.

- tid_seq: An 8 bit field which is the high byte of the TID. The low byte of the TID is this task's index in the kernel's statically allocated array of task descriptors, and is calculated when needed rather than stored. The mechanics of this are fully described below in **Task Creation**.

- parent_ix: An 8 bit field which is the parent task's index in the kernel's statically allocated array of task descriptors.

- next_ix: An 8 bit field which is the index of the next task in whatever queue this task descriptor is currently participating in, if any. This index is once again an index into the kernel's statically allocated list of task descriptors.

- senders: A queue structure to be used as the task's send queue.

- cleanup: A 4 byte function pointer to a cleanup callback which is run when the task exits (if the task registers one).

- irq: An 8 bit field to hold the event number if this task calls RegisterEvent.

- time: The time in milliseconds this task has spent running.

- fpu_ctx_on_stack: A flag which is set to true if the task has it's FPU context stored on it's stack.

- task_fpu_regs: A pointer to the task's saved FPU registers if they are saved on it's stack.

Task Queue – These are singly-linked lists using the task descriptors' “next” pointers. In addition, the queue “objects” are represented by structures containing the following 2 fields in order:

- An 8 bit field which is the index of the first task in the queue. The index is an index into the kernel's statically allocated array of task descriptors.
- An 8 bit field which is the index of the last task in the queue. The index is an index into the kernel's statically allocated array of task descriptors.

Enqueue works in constant time by modifying the last element and then replacing it. Dequeue works in constant time by simply popping from the front.

Kernel State – This structure contains all major kernel state and references to kernel data structures. It has the following fields:

- user_stacks_bottom: A void pointer to the top of the memory set aside for user task stacks.
- user_stack_size: A size_t which holds the amount of memory per user task stack.
- tasks: The kernel's statically allocated array of task descriptor structs.
- rdy_queue_ne: A 16 bit integer which represents whether or not there are tasks in any of the ready queues, one bit per priority level.
- rdy_queues: An array of queue structures, one per priority level. These are the ready queues.
- free_tasks: A queue structure to hold free task descriptors.
- eventab: A structure containing the event system information.
- shutdown: A boolean flag which will stop the kernel loop when set.
- rdy_count: An integer number of tasks on the ready queues.
- evblk_count: An integer number of tasks which are blocked waiting for an IRQ.

Hardware Initialization:

The bootloader jumps into _asm_entry subroutine defined in startup.S. This code should perform initialization that should happen before jumping into C code. This includes enabling access to the VFP coprocessors if necessary/desired, setting the initial kernel stack pointer, setting up the stack pointer for the undefined instruction mode, etc. The last thing the assembly entry point should do is branch to the main function.

The main function (defined in main.c) calls a few functions before starting the kernel proper.

- cache_enable: Any work necessary to enable data and instruction caches.
- pll_setup: Any work necessary to configure system clocks and PLLs.
- dbg_tmr_setup: Setup a free-running system timer to be used for debugging.
- bwio_uart_setup: Configures a UART to use for polling debug I/O.

Lastly, kern_main is called to start the kernel.

Kernel Initialization:

First, the hardware specific code to load the exception vector table is called. Then, the kernel event table (a structure mapping IRQ numbers to task ID's/callback functions) is initialized and the interrupt controller reset.

Next, the size of the individual user stacks is calculated by dividing the size of the memory region allocated for user stacks by the number of concurrent tasks supported by the kernel (a constant, currently set to 128).

Finally, all task descriptors are added to the free list and the ready queues initialized to empty. The IDLE and u_init tasks are created as well.

Task Creation:

The kernel's task creation code first ensures that the caller has supplied a valid priority level, and that there is a free task descriptor that can be used. If either of these is not the case, it returns the appropriate error code.

A free task descriptor is then taken and initialized. It's initial stack pointer location is calculated by multiplying the size of a user task stack by this task descriptor's index in the kernel's array and subtracting the result from the top of the entire user stack region. To obtain the pointer to the task's saved state, the size taken by all saved user registers is subtracted from the task's initial stack pointer. By initializing the values in the saved state, the kernel can jump into the task from the beginning as if it was already running.

Into the saved state the kernel writes the value that will get loaded into the SPSR (user mode, interrupts enabled), the value that will get loaded into the user link register (the address of the Exit() code so it will be called automatically when the user tasks return), and the value that will get loaded into the program counter (the supplied function pointer to the task entry point).

Last, the task descriptor is placed into the appropriate ready queue for its priority.

The whole TID is not explicitly stored in the task descriptor in an attempt to save space. The kernel initialization code sets the high byte of the TID (stored in the task descriptor) to zero. Whenever the task descriptor is put back in the free list (the user task exits), this field is incremented. The low byte of the TID is the index of the task descriptor in the kernel's array (and can be calculated in constant time by subtracting the pointer to the task descriptor from the pointer to the start of the task descriptor array). This way, the kernel can reuse task descriptors but not reuse a TID until the high byte overflows. There are macros available to generate the full 16 bit TID whenever it's needed, but that is usually only when writing or reading TID values to or from user tasks.

Scheduling:

Scheduling works in the following way: first check to see whether any of the “ready queue not empty” bits are set in the kernel state (i.e. whether the bit field value is greater than zero). If not, there are no tasks to schedule. Otherwise, determine the highest priority queue with ready tasks by counting the trailing zeros in the bit field. (Trailing zeros are used because lower numeric values represent higher priorities.) Then dequeue a task from that priority's queue, set its state to active and return it.

All operations in this scheduling algorithm run in constant time.

Context Switch:

Our context switch is written in assembly and is 6 instructions long. The kernel state (r0, r4-11, lr_svc) is stored on the kernel stack using a store-multiple (full descending) which modifies the value of the kernel stack pointer as it goes.

Then the address of the saved user task state is loaded into the ip register from the memory address stored in r0. The r0 register contains the pointer to the task descriptor (the only argument to the context switch function), the first element of which is a pointer to the saved task state.

Next the task SPSR value and the user task program counter are popped off the user stack using a load-multiple (full descending). The SPSR value is loaded into r3 and the program counter is loaded into the kernel link register. The value in r3 is then put into the SPSR.

The remaining user task register values (r0-r12, sp_user, lr_user) are then loaded using the load-multiple variant which loads into the banked user mode registers.

Last, the user mode program counter is loaded into the actual program counter using the move instruction which also restores the SPSR to the CPSR. This causes a branch into the user task.

Kernel Entry:

The three kernel entry points all do approximately the same thing. They switch into SYS mode to save the user registers on the stack (with store multiple). Then they take note of the task's stack pointer and switch back to the inital mode to save the link register and SPSR. Next they restore the kernel context (with load multiple) and set an appropriate return value for the ctx_switch function to signify which of the three path's into the kernel is being taken (SWI, IRQ, UNDEF).

Software Interrupt Handling:

After the context switch returns, the interrupt handling code is invoked. To get the software interrupt number, the kernel reads the user task program counter from the saved task state and use that to find the software interrupt instruction (which contains the interrupt number). After handling interrupts, the kernel loops back to the scheduling code.

If the software interrupt was a call to Create(), the task creation code is invoked, the return value stored as the r0 value in the task's saved state, and the calling task is put back in the ready queue. For MyTid(), the task TID is calculated, inserted as the return value, and the task is put back on the ready queue. MyParentTid() is identical except it calculates the TID of the parent task. To handle Pass(), the kernel simply re-adds the task to the ready queue. On Exit(), the high byte of the TID saved in the task descriptor is incremented, the task state is set to “free”, the task descriptor is added to the free list, and the number of tasks stored in the kernel state is decremented.

These “basic” system calls are implemented using the algorithms and datastructures described above. With the exception of hardware interrupt related system calls (described in the section on hardware interrupts) and a few general system calls (outlined below), the rest of the system calls are wrapped calls to IPC functions which communicate with the appropriate server (user task).

IPC Calls:

Rendezvous – This is not directly callable by user tasks. It is a function to contain the common code which must be executed when a send is "matched up" with a receive. This copies the sender's message into the receiver's buffer. The sender is then put into reply-blocked state and the receiver is made ready again with the sender's TID and the sent message size as the return value.

Send - Send first puts the sending task into receive-blocked state. It then checks the state of the receiving task. If it is send-blocked, there is an immediate rendezvous. Otherwise, the sender is added to the receiver's send queue. If the receiving task exits before receiving the message, the sender will block forever.

Receive - Receive first checks the receiving task's send queue. If there are tasks in the send queue, it immediately pops the first one and rendezvous with it. Otherwise, it enters send-blocked state and will remain blocked until awoken by a send.

Reply - Reply immediately copies the reply message into the sender's buffer and makes both the sender and the replier ready.

Note: Any TIDs are validated before performing any of these actions. The low byte of the TID is the index into the kernel's task descriptor table. This index must be in range and the sequence number of the corresponding task descriptor must match the high byte of the TID. In this way the kernel can look up tasks by TID in constant time.

Nameserver Calls:

RegisterAs – The nameserver finds or creates a name record for the requested name, and sets its TID to the TID of the registering task. If the name is new and there is no room for further name records, the name server replies with -3. If the name already has a name record, then the TID for that record is overwritten with the registering task's TID. Then the name server replies to the registering task with a successful acknowledgement, and to tasks waiting for that name (if any) with the appropriate TID. The waiting tasks are replied to in an order opposite to that in which they made their requests due to their storage in a stack. This is a wrapped call to send.

WhoIs – The nameserver finds or creates a name record for the requested name. If the name has an associated TID, replies immediately with that TID to the requesting task. Otherwise, adds the requesting task's TID to the stack of waiting tasks for that name. If a new name record is required, but can't be allocated, the requesting task is simply allowed to hang forever, since no task will ever successfully register that name. This is a wrapped call to send.

Clock Server Calls:

clkctx_init – Initialize a clock context structure which contains the TID of the clock server. This is a wrapped call to WhoIs. The clock context is passed as the first argument to the other clock server calls.

Time – Requests the current system time in ticks from the clock server (wrapped call to send).

Delay – This is a wrapped call to send. The clock server doesn't reply until after the specified amount of time, leaving the calling task blocked.

DelayUntil – This works on the same principle as Delay, but the calling task is blocked until after the absolute time value specified instead of a time value relative to when it was called.

(Currently Unimplemented) Serial Server Calls:

serialctx_init – Initialize a serial server context structure. This is a wrapped call to WhoIs. It asks for a different server depending on whether the caller specified COM1 or COM2.

Getc – This blocks the user task until the serial server has a character from the UART to send back to the caller. It is a wrapped call to Send.

Putc – This sends a single character to be added to the serial server's output buffer and eventually sent out on the UART. It is a wrapped call to send, but typically doesn't block for very long as we only allow one task at a time to make calls to a particular serial server. This means that we should get a reply almost immediately after the serial server is scheduled next.

Write – This sends a caller specified number of bytes to be added to the serial server's output buffer. It is implemented with our fast assembly memcpy so it is faster than Putc in a loop. It is a wrapped call to send, but typically doesn't block for very long.

Print – Wrapped call to Write which determines number of bytes to send by looking for a null-terminator at the end of the given string.

Printf – Wrapped call to Write after generating the output from the format string and arguments.

Flush – This blocks the calling task untill all the bytes in the serial server's transmit buffer are sent to the UART. It is a wrapped call to send.

General System Calls:

RegisterCleanup – A way to be able to run clean up code on a per task basis (usually to restore hardware to a nice state) whenver a task exits or when the kernel is shut down. RegisterCleanup can be called to install a cleanup callback (the address of which is stored in the task descriptor) for the calling user task.

Shutdown – If a task calls shutdown, the kernel calls any registered task cleanup callbacks, puts the hardware it's responsible for into a consistant state and returns to the assembly startup code/bootloader.

Hardware Interrupts Overview

RegisterEvent and AwaitEvent:

If a task wants to be notified when an IRQ occurs, it makes a call to RegisterEvent. A task can only register a single event, and a particular event can only be registered by a single task. When registering, the caller chooses a priority. However, the interrupt isn't enabled in the INTC until the task calls AwaitEvent.

The task also passes a callback function pointer to RegisterEvent. This callback will be run with interrupts disabled inside the kernel when the interrupt occurs. It is the job of the callback to store any volatile data and clear the interrupt in the hardware that caused it. After the callback is run, the interrupt is disabled again in the INTC and the associated task made ready with the callback's return value as the return value from AwaitEvent.

The callback takes two arguments, a void pointer (buffer) and a size_t, which is generally the buffer size. The values of these arguments are specified as arguments to AwaitEvent. The callback function returns an integer. If the integer is greater than or equal to zero, it is returned to the task as the return value of AwaitEvent. If the return value is negative one, the kernel doesn't disable the interrupt in the INTC, and the task is left event blocked. This allows a user task to "ignore" an interrupt which doesn't merit unblocking. This is useful for something like the CTS line, which causes an interrupt whenever it changes state, but we are generally only wanting to do something while it is actually asserted.

Data Structures:

The number of an event which a task has registered is stored in that task's descriptor. The kernel keeps an event table, indexed by event number. Each table entry contains the TID of the registering task, the callback pointer, and the values of the callback arguments from the last call to AwaitEvent (initially set to invalid values). The kernel also keeps a bit mask (64 bits long), where each bit represents whether or not the corresponding IRQ has been registered. This way we can detect which IRQ's have already been registered in constant time.

After an interrupt occurs and the kernel is entered, the interrupt handling code checks the interrupt controller for the highest priority asserted IRQ in constant time (by checking a specific register). This is used to look up the event in the kernel event table in constant time. Only the highest priority event's callback is called. This is alright because the hardware which caused the lower priority interrupt will still be asserting it's IRQ since it's callback wasn't called. It is the responsibility of the callback author to insure that it runs in constant time. Disabling the interrupt and making the event-blocked task ready again are also constant time operations. A register is also written to clear the INTC.

Why Callbacks?

There are two schools of thought on hardware interrupt handling. The first involves making a kernel magically know how to handle things like copying volatile data from/to specific places. This is odd because the servers/notifiers which are supposed to handle those peripherals are user tasks, and this option divides the code between the kernel and those tasks.

Another option is to jump back into the associated user task with interrupts disabled so that it could handle the hardware (and trust the user code to make a system call shortly thereafter to return to the kernel). This works if the authors of the user code are aware of this and very careful, but seems like a bad design for an operating system in general.

With the user supplied callback, the user code specifies how the interrupts are to be handled in a more contained manner than jumping into the task itself. The callbacks send/get data from the user task via the parameters passed in from AwaitEvent, and run in supervisor mode with interrupts disabled. This is also potentially dangerous, as somebody could supply a callback which doesn't terminate. However, in a kernel intended to run on embedded systems, “user tasks” provide a nice way for concurrent code to share the processor, but are not truly meant to be secure. The “user space” developer maintains some responsibility for the stability/robustness of the system, as they have direct hardware access.

This callback functionality is comparable to linux kernel modules, which privileged users can use to insert code into the kernel.

Servers:

Nameserver:

The name server is the second user task started. This must be guaranteed by the initial user task ("u_init"), which starts the name server before doing anything else. Thus the TID of the name server is known ahead-of-time to be 1.

The name server keeps an array of name records. The array has a fixed length, and the server uses a growing prefix of it. Each name record has a name, a TID, and a stack of the TIDs of waiting tasks. Records are found in this array by a simple linear search. There is no concern about the performance of this, since it is expected that it will only run during initialization. Tasks should remember the TIDs of other tasks of interest as much as possible.

New records are appended to the array either due to a registration or a lookup for a previously nonexistent name. If a registration, then the TID is known, and all subsequent lookups will find it. Otherwise WhoIs will block until a task registers with the requested name. Accordingly, a record is created with a sentinel for an unknown TID, and until the name is registered, any further lookups are added to the stack. A stack was chosen because the order in which the waiting tasks are signaled seems unimportant, though this can easily be made into a singly-linked queue if need be.

Clock Server:

The clock server first runs some initialization code. The initialization code sets up the necessary data structures, starts the notifier, and registers with the name server. After initialization, the server receives messages in a loop.

If the message is a Time system call, the currently stored time value is sent as the reply.

If the message is a Delay or DelayUntil system call, a check is performed to see if the delay is for an amount of time that is greater than zero. If it is, the task is added to a priority queue with it's wakeup time as the key. Otherwise the server replies to the task immediately, potentially with an error code if the delay time was less than zero.

If the message is a tick notification from the notifier, it increments the stored time value. Then, for as long as the first wakeup time in the queue is less than or equal to the new currently stored time, it pops the first task and replies to it.

The performance of the clock server primarily revolves around the queue data structure used to handle the Delay and DelayUntil system calls. Two types of priority queues are implemented. One was implemented using a ring buffer with linear time insertions but constant time extraction. The other was implemented using a heap with logarithmic insertions and extractions.

To compare the performance of each, a 40-bit timer (on the system at the time) was used to calculate the average insertion and extraction times (per task) for a few different amounts of tasks. Here are the results:

```
Heap:

Tasks        : 04   08   16   32   64   128
Insert  (us): 0.76 0.51 0.57 0.57 0.59 0.60
Extract (us): 1.02 1.02 1.21 1.53 1.88 2.22
Total   (us): 1.78 1.53 1.78 2.10 2.47 2.82

Ring Buffer:

Tasks        : 04   08   16   32   64   128
Insert  (us): 1.27 0.89 0.89 0.89 0.91 0.85
Extract (us): 0.51 0.38 0.45 0.51 0.48 0.48
Total   (us): 1.78 1.27 1.34 1.40 1.39 1.33
```

128 was the maximum number of insertions/extractions because it was the maximum number of tasks in the system. The sets of results for smaller numbers of tasks are informative since those cases will be more common. Based upon these results, we chose the ring buffer implementation.

Clock Server Notifier:

First the notifier finds the TID of the clock server using the name server. After clearing and initializing the 32-bit timer to generate an interrupt every 10 ms, the notifier calls RegisterEvent and registers the 32-bit timer IRQ number with the highest interrupt priority (0). The callback it provides clears the IRQ in the timer. The notifier then calls AwaitEvent in a loop, sending a message to the clock server every time the timer interrupt is asserted.

Serial Server (Currently Not Implemented):

When the serial server starts, the first thing it does is call Receive to wait for a configuration message, usually sent by the task that starts the serial server. The configuration message includes the desired uart number, the baud rate, the number of bits per frame, parity checking settings, number of stop bits, whether or not to pay attention to CTS, and whether or not the fifo's should be enabled (although FIFO's are not currently implemented). Once it has this information it configures the UART hardware, starts up the notifiers and registers with the nameserver.

After initialization the serial server enters it's main loop. It receives a messages and deals with them appropriately. If there is a message from the receive notifier, we take the byte it's given us and put it in the receive buffer (and send an empty reply to the notifier so it can unblock and wait for the next byte). The buffers used in the server are FIFO ring buffers with constant time add/remove operations. We only allow one task to call Getc on a particular serial server at a time. This means that we only have to store one TID to reply to with a byte if the receive buffer is empty when the Getc call is made, and all other tasks that try to call Getc concurrently will be replied to with an error message.

Depending on whether or not the server is configured to respect the CTS line when transmitting, there are two or three things which must come together in order to transmit a byte. First, we must have a byte to transmit. These come from client Putc/Write messages and are put into the transmit ring buffer. Also, the UART must be ready to transmit the next byte, which we find out about via messages from a notifier that listens for the transmit FIFO/holding register empty interrupt. If the server is configured to check the CTS state, we also need to wait for a CTS asserted message from the notifier that listens for the combined UART interrupt. Every time one of these three types of messages are received, the server checks if all two/three conditionas are met. If they are, the server writes a byte to the UART and sets it's internal state variables to false in order to wait for all the correct signals to come in again before sending the next byte.

One extra feature we've added to the serial server is the Flush call. When a flush message is received from a user task (and again, only one task at a time is supported), it is replied to imediately if the transmit ring buffer is empty. If it isn't, it's TID is stored and it is replied to when the last byte goes out.

Serial Server Notifiers:

There are three individually accessable interrupts per UART: Receive FIFO/holding register full, Transmit FIFO/holding register empty and the combined UART interrupt which includes the modem status interrupt. Each interrupt is handled by a notifier task which spends most of it's time event blocked. The RX/TX FIFO interrupts are registered at higher priorities in the vector module than the combined interrupt so that we never have to handle the FIFO/HR interrupts in the combined interrupt's handler.

The receive notifier registers to receive the RX FIFO/holding register full interrupt with a callback that reads a byte from the UART (to clear the interrupt) and returns it to the nofier task as the result of AwaitEvent. It sends this byte to the serial server and then loops back around to wait for the next interrupt.

The transmit ready notifier registers to receive the TX FIFO/holding register empty interrupt. When it does, it signals the serial server and loops around to wait for the next one. Due to this interrupt only being de-asserted by the UART when a byte is written to its data register, the callback which is executed when this interrupt is asserted masks out the interrupt in the UART CTRL register. This prevents continuously trapping back into the kernel if we don't yet have a byte in the transmit ring buffer. When a byte to transmit is received by the server, the server unmasks the interrupt.

The combined UART interrupt notifier currently only checks the status of the CTS bit in the modem status register since we haven't yet implemented FIFO's. The callback it registers checks to see if the CTS bit is set. If it is, it returns normally and the notifier task is woken up to signal the serial server. If it isn't (since the interrupt is asserted for changes in either direction), the callback returns -1 and the notifier task remains event blocked.