helium

version 1.1

real-time kernel documentation

Introduction

This document explains the inner workings of the Helium real-time kernel. It is not meant to be a user's guide. Instead, this document explains overall structure of the kernel as well as individual peculiarities of specific functions.

OS Initialization

All tasks must be created before the operating system begins to run. Tasks can be dynamically created and destroyed in this version of Helium. If a task is destroyed during the course of execution, it is just taken out of the list of tasks to be scheduled; it is not removed from memory. When it is restarted, the task's stack pointer is reset to its original value before it began running, thereby invalidating any data stored in its local variables and resetting its program execution flow. A task is created by calling the function `TaskCreate(…)`. `TaskCreate` takes four parameters in the following order:

1. The priority level of the task. This byte-long number must be in the range 0 – `NUMTASKS`-1. The highest priority level is zero. The lowest priority level is `NUMTASKS`-1. `NUMTASKS` is defined in `os.h`. It specifies the maximum number of tasks that the system will run. Reducing this number greatly decreases the amount of RAM the OS uses for bookkeeping purposes.
2. A pointer to the function that implements the task. In C, simply type the name of the function.
3. A pointer to the lowest address in the task's stack. On the HCS08, stacks grow downward in memory, but don't pass a pointer to the highest location of the stack to `TaskCreate`. It will determine the highest location of the stack using the constant `STACKSIZE` defined in `os.h`.
4. The size of the stack space in bytes. If, for example, the stack is declared as follows:

   ```
   int stack[50] ;
   ```

   then the size of the stack space is 100 bytes because variables of type `int` occupy two bytes of memory.

Operating system initialization takes place inside the function `main`. In Helium, the `main` function is an application-specific, user-defined function (i.e. the `main` function is not a part of the RTOS). It is the first function that is called when the computer starts up. It performs some initialization and then hands control of the CPU over to the OS which proceeds to schedule tasks. Inside the function main, there are four critical operations that need to be performed:

- All MCU peripherals need to be initialized, most important of which is the timer/pulse-width modulator. The TPM needs to be set to interrupt the MCU on some time interval between 10 and 100 times per second, and the ISR that handles the TPM overflow needs to be set to `TickISR`. Without initializing the TPM, the scheduler cannot schedule multiple tasks.
- Tasks need to be created. Call the function `TaskCreate` as explained above.

- Interrupts need to be enabled. Without enabling interrupts by executing the `cli` assembler instruction, the scheduler cannot function properly.
- The function `StartExecution` needs to be called. **This must be done last** because once this function is called, program flow will never return to the function `main`. `StartExecution` hands the CPU over to the Helium kernel, allowing the scheduler to start and stop user tasks as needed. It hands control of the CPU over to the highest priority task that has been created and never returns to `main`.

Organization

The Helium operating system consists of a set of interrupt service routines and support functions that can be used by tasks to control MCU peripherals and communicate with each other. In Helium, a task is nothing more than a C function that has its own stack. A task may call any other C function during the course of its execution. Two tasks may each call a common C function. It is the operating system's responsibility to keep track of which task needs to be executing at any given time and to subsequently give control of the CPU to that task.

Data about tasks is stored by Helium in two ways: in global variables and in space on each task's stack. Global variables store information about each task such as its priority, its starting execution address, and its stack pointer. When a task is preempted by the operating system, its CPU context is held on its stack[1]; that is, the contents of each CPU register that the task was using is stored in a particular order on the task's stack, occupying a total of six bytes. When the task is rescheduled, all of the task's registers are restored from the stack and program execution continues at the location where it left off, giving the appearance that the task was running continuously.

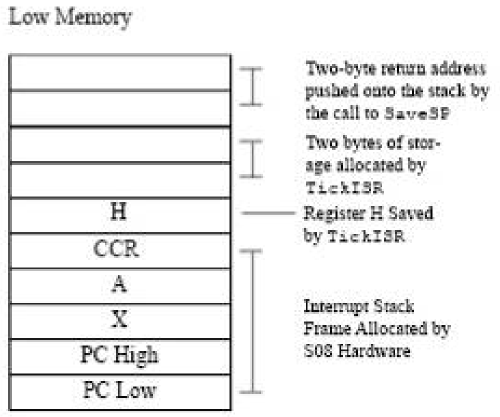
Since each task has its own stack, it is important to keep track of each stack pointer individually. When a task is preempted by the operating system (usually in response to an interrupt), the S08 CPU automatically pushes all registers onto the currently-active stack. For a new task to be scheduled, though, the stack pointer must be redirected to point to the new task's stack space. To do this, the first task's stack pointer is saved by the scheduler in a global variable and the new task's stack pointer is restored to the SP register from a global variable where it had been saved the last time it was preempted by the operating system. This is done by the assembler routines `SaveSP` and `InterruptSwitchTask`. These two functions are called in `TickISR`.

`TickISR`

This ISR is the main part of the scheduler. It is executed every time a clock tick interrupt occurs. It declares two variables of type char, and space for them is allocated on the currently-executing task's stack. The function `SaveSP` called by `TickISR` assumes that `TickISR` allocates two bytes of local variable storage space on the stack. In the process of allocating two bytes of local storage, `TickISR` decrements the stack pointer by 2. Since `SaveSP` needs to preserve the stack pointer as it was before that storage space was allocated, it adds four to the stack pointer it saves—two for local storage space

[1] CPU context includes the contents of all registers except for the stack pointer. The stack pointer is saved separately. Helium stores a task's CPU context in a standard S08 interrupt stack frame. See an S08 CPU reference for more information about the interrupt stack frame data structure.

within `TickISR` and two for a return address when `SaveSP` returns. This is done in the fifth line of the assembler function `SaveSP` (see the figure below).


Low Memory
H
CCR
A
X
PC High
PC Low
Two-byte return address pushed onto the stack by the call to SaveSP
Two bytes of storage allocated by TickISR
Register H Saved by TickISR
Interrupt Stack Frame Allocated by S08 Hardware

The stack frame as it exists inside the function SaveSP. *The S08 hardware pushes all CPU registers except for H and SP onto the stack when it detects an interrupt. H is pushed in software by the ISR, and two bytes of work space are allocated by the ISR as well. A return address is pushed when the ISR calls SaveSP, causing the stack pointer to point one byte below 10-byte data structure. We want the saved copy of the stack pointer to point one byte past H so that when the task is rescheduled, all registers are restored. To do this, we will add 4 to the stack pointer's value as it exists inside SaveSP.*

When `TickISR` is called, it assumes that an interrupt stack frame has been allocated as the result of an interrupt. The interrupt stack frame on the S08 is a five-byte structure that saves the CPU registers to ensure that the task that was running will continue executing correctly when it is resumed. If this function is invoked by a non-exceptional condition (i.e. calling it as a regular function, not an ISR) the interrupt stack frame will have to be created by the programmer. This is done using the following steps in order[2]:

1. Push the address of the next instruction to be executed after `TickISR` returns.
2. Subtract 3 bytes from the stack pointer.
3. Use a `jmp` instruction (rather than a `bsr`) to begin executing `TickISR`.

Alternatively, the same steps could be accomplished by using the `swi` assembler instruction.

The first thing that `TickISR` does is clear the interrupt source, assumed to be the timer peripheral. The assembler macro `ClearTimerIntr()` accomplishes this. Details about clearing the timer interrupt are available in the S08 datasheet.

Next, it loops through an array called `TickBlock` to find any nonzero elements. Each element of `TickBlock` holds the number of clock ticks that the associated task

[2] This process creates a data structure on the stack that has the same size and configuration as an interrupt stack frame that would created by the S08 hardware in response to an interrupt. See the S08 CPU reference for more information regarding interrupt stack frames.

has requested to block for. All nonzero elements are decremented, and if the decrement results in the element being zero, the associated task is made ready to run.

The final operation that `TickISR` performs is scheduling the highest priority task ready to run. A task's ready status is indicated by a single bit in the 8-bit variable `Tasks`. The task with priority 0 (the highest priority task) is ready to run if bit 0 in the variable `Tasks` is set and is not ready to run if bit 0 is cleared. The highest priority task ready to run is determined by using a lookup table to determine the bit number of the least significant bit that is set in the variable `Tasks`.

If a new task has to be scheduled, the assembler function `SaveSP` stores the old task's stack pointer. **When `SaveSP` is called, the variable `currTask` must hold the value of the old task's priority.** The value of `currTask` should not be changed until `SaveSP` has finished saving the old stack pointer. `SaveSP` preserves the value of the stack pointer in the array element `Stacks[currTask]`. `SaveSP` does not save the value of the stack pointer as it exists inside its own function body. This is because the stack pointer has been decremented to allocate space for the two byte-sized values in `TickISR` and the return address when `TickISR` called `SaveSP`. To account for those modifications, `SaveSP` actually saves the stack pointer incremented by 4. **The incremented number reflects the position of the stack pointer before `TickISR` allocated a variable on the stack (two bytes) and called `SaveSP` (pushing two bytes onto the stack for a return address).** The number that actually gets saved in the `Stacks` array points to the top of the stack frame that stores the CPU context.

After the old stack pointer has been preserved, the byte-sized number `currTask` that represents the currently-scheduled task is updated to reflect the new task. The function `InterruptSwitchTasks` is then called. This is not actually a function call but a jump to an assembler routine. It was written this way because it will be unnecessary to return to the function `TickISR` after `InterruptSwitchTasks` has finished executing. Since a function call would push a return address onto the stack that would clutter the stack and eventually have to be removed anyway, it's better to just execute a `jmp` instruction.

`InterruptSwitchTasks` is implemented in the assembler function `INTR_SW_TSK`. Its functions are as follows:

1. Restore the stack pointer for the new task. The priority level of the new task was written to `currTask` by `TickISR`, so all `InterruptSwitchTasks` has to do is load that number from the array element `Stacks[currTask]` into the SP register.
2. Pop the H register off of the stack. H is not automatically saved and restored by the CPU during an interrupt, so the software must do that for itself.
3. Execute an `RTI` instruction which restores all other CPU registers including the program counter.

StartExecution

This function starts the multitasking. First, it prepares each task's stack frame to appear as if it has already been preempted by an interrupt. This is done because the scheduler assumes that every task's stack contains an interrupt stack frame that it can use to restore the task's CPU registers and program counter. This is clearly not the case before multitasking has started. Therefore, a stack frame must be created for each task so that the first time that each task is scheduled there will be valid data to fill the initial program counter and registers with. After the stack frames have been created, it `StartExecution` determines which task has the highest priority, loads that number into the SP register, and executes an `rti` instruction.

Mailboxes

This code adds message mailbox functionality to the kernel. Message mailboxes are implemented in the file mbox.c, and the idea was to make each module of the OS independent. As such, the kernel does not make any special checks of the mailbox variables when scheduling tasks. Message mailbox functionality can be removed from the OS to save space by simply deleting mbox.c from the project and deleting references to mailbox variables in os.h.

There are two message mailbox API functions in Helium:

1. void mBoxPost(char mBoxNum, int data)

   Puts a 16-bit integer into the message mailbox specified by mBoxNum. mBoxNum can take values from 0 – NUMMBOX-1. NUMMBOX is a constant defined in os.h. For each additional mailbox specified by NUMMBOX, four bytes of RAM are used.

2. int mBoxPend(char mBoxNum)

   Returns the contents of the message mailbox specified by mBoxNum. If that mailbox does not contain valid data, the task that called it will block until valid data is posted to the mailbox.

`mBoxPost()`

mBoxPost handles two separate possibilities: either a task has already blocked, waiting for a message to be posted to the specified mailbox or a task has not blocked to wait for the mailbox.

If a task has blocked to wait for the mailbox to fill up, it will have an interrupt stack frame allocated in its stack. The message contained in the variable data will need to be returned to the waiting task when it is rescheduled, and the calling convention for the S08 dictates that 16-bit data be returned in the index register H:X. To accomplish this, the MSB of data will be placed in the space allocated for register H on the stack frame, and the LSB of data will be placed in the space allocated for X. If the task is already blocked, the SP entry in the Stacks[] array for that task will point to the location from which H will be restored. SP+3 will point to the location where X will be restored from. Thus, the MSB and LSB need to be stored in these respective locations.

After the data is placed in the interrupt stack frame for the waiting task, the task will be made ready to run, and execution of the currently-scheduled task will continued until either it blocks or it is preempted.

If no task is already waiting for data in the mailbox, the data will simply be put in an element of a global array that holds values waiting to be passed to tasks. Bit 1 of the mailbox status byte will be set, indicating that the array element holds valid data, and the function will return to its caller. Bit 1 is not set if a task is already waiting on the mailbox because the box never gets valid data. Instead, the data is passed directly to the waiting task's interrupt stack frame.

`mBoxPend()`

This function also handles two possibilities: either the mailbox contains valid data or it does not. If it contains valid data, control will immediately return to the calling task, the data will be unloaded from the mailbox, and the mailbox data will be marked invalid. If no valid data is present, the requesting task will block until valid data is posted. Bit 1 of the mailbox status byte indicates whether the mailbox contains valid data.

If the mailbox contains valid data, bit 1 of the mailbox status byte is cleared to indicate that the data in the mailbox is no longer valid. The data in the mailbox is then returned to the caller.

If the mailbox does not contain valid data, the task must set some flags to indicate that it is waiting for data in the mailbox.

1. Set bit 0 of the mailbox status byte to indicate that a task is waiting for data to become valid in that mailbox.
2. Put its task number into `mBoxWait` to indicate which task is waiting for that mailbox.
3. Clear bit 0 of the task's status byte to indicate that it is no longer ready to run.

An interrupt stack frame for the stack is then created by subtracting 3 from the stack pointer and the scheduler is called, just as is done in `TimeBlock`.