Real Time Kernel v1.1

1. Task Management

`void TaskCreate( char taskNum, void (*t)(), int *stack, char stackSize )`

Creates a task with the unique priority `taskNum`. Helium allows tasks to have one of eight priority levels. A task with priority level zero has the highest priority, and one with priority level seven has the lowest priority. The task is represented as a C function pointed to by `(*t)()` (the second parameter to `TaskCreate`) and an associated stack space pointed to by `stack` (the third parameter to `TaskCreate`). For example:

```
int t1Stack[20] ; // global array to hold the task's stack

...

void t1() // Program code for the task.
{
   for( ; ; )
   {
         ...
   }
}

...

void main()
{

   ...

   // Create the task inside the function main.
   TaskCreate( 1, t1, (int*)t1Stack, 40 ) ;

   ...

   // Hand control of the CPU over to the highest priority task.
   StartExecution() ;
}
```

The above code segment begins by allocating a 40-byte block of memory for the task's stack (remember that an `int` on the S08 is two bytes wide). The memory block must be a global array, not a local variable inside the function `main`. Some time later it defines a function `t1` that will hold the task's code. All task code must be contained within a C function, but the function must never include a `return` statement. Instead, it should delete itself using the Helium API function `TaskDestroy`. Inside the main function, the task is created by calling `TaskCreate`. It is given a priority level of 1 (with priority level 0 being the highest and 7 the lowest). A function pointer is passed as the second argument, telling the operating system where to begin executing code when the task is scheduled. A pointer to the **lowest** address of the task's stack is passed as the third

argument. The fourth argument is the length of the stack in bytes. Finally, the function `StartExecution` is called, handing control of the CPU over to Helium which will begin executing the highest priority task. After calling `StartExecution`, the function `main` will never be given control of the CPU again.

```
void TaskDestroy( char taskNum )
```

Takes the task with priority taskNum out of the waiting list of tasks that are ready to run. Any resources that the task was waiting for are freed.

```
void StartExecution()
```

Gives control of the CPU to the highest priority task that is ready to run. This function should be the last function called inside the `main` routine after initializing all peripherals and creating all tasks. Once `StartExecution` is called, control will never be returned to function `main`. Instead, `StartExecution` will give control of the CPU to the highest priority task ready to run.

2. Timer Management

```
void TimeBlock(char numTicks)
```

Causes the task to block for a specified number of clock ticks. Clock ticks are interrupts that are generated at some frequency between 10 and 100 times per second. The S08's on-chip TPM module generates the interrupts, and Helium is responsible for rescheduling the highest priority task ready to run every time an interrupt is generated. By calling `TimeBlock`, the programmer is indicating that a task will not need to run for the specified time interval, and Helium will allow a lower priority task to run in that time interval. When the number of ticks specified has expired, Helium will again reschedule the task provided that no higher priority task is ready to run.

For example, if the scheduler clock interrupts the CPU at a frequency of 100 Hz and the task with priority level 2 needs to block for a period of 1 second (and the next highest priority task is level 3), the system call

```
TimeBlock( 100 ) ;
```

would cause the scheduler to start the task with priority level 3 and run it for 1 second.

3. Message Passing

```
void mBoxPost( char mBoxNum, int data )
```

Sends a message to another task via the message mailbox `mBoxNum`. The message is a 16-bit integer. If a task had been waiting for a message to be posted to the mailbox, it is immediately rescheduled. If no task was waiting, `mBoxPost` returns to the task that called it and continues running.

```
int mBoxPend( char mBoxNum )
```

Retrieves a message from the mailbox `mBoxNum`. If a message is waiting in the mailbox, `mBoxPend` returns immediately. If a message has not yet been posted to the mailbox, the task that called `mBoxPend` blocks until a message is posted. When a message is posted in the mailbox, the task that called `mBoxPend` will again be made ready to run.

4. Starting Helium

There are several steps in the process of configuring the Helium RTOS. Peripheral devices must be initialized first. In particular, Helium needs to be interrupted several times per second by a so-called “clock-tick interrupt” source. This predictable, periodic interrupt source makes it possible for Helium to facilitate cooperative use of the CPU by several different tasks. Without a periodic interrupt source, one task would gain and hold control of the CPU indefinitely.

Also crucial to the configuration process is task creation. Tasks in Helium are analogous to processes on a PC: they have their own memory sections allocated for instruction and data storage and they can be started or stopped at any time during system operation. It is prudent to create all tasks the system will use consistently during the configuration process.

Example: An implementation that keeps track of how long the system has been running.

Overview

This implementation will rely on an RS232 terminal interface to display a shell prompt to the user. When the user types the command *time* at the shell prompt, the amount of time the system has been running will be displayed using hexadecimal representation.

How It Works

This implementation will use the 9S08QG8 RS232 driver which has provisions for sending characters, strings, and hexadecimal representations of individual bytes to the terminal. One task will be in charge of the user interface, while a second task will be in charge of keeping an accurate count of the number of hours, minutes, and seconds elapsed since startup. When neither of those tasks is executing, a third idle task will run.

The user interface task will call RS232 driver functions to display the command prompt and interrogate the serial port for user input. The RS232 driver functions will cause the task to block until their job has been completed. For example, if the user interface task needs to send a string of characters, it would call the function `sciPutStr`:

```
sciPutStr( “Mary Had a Little Lamb” ) ;
```

Since the process of sending the whole string across the serial port will be relatively slow, `sciPutStr` will cause the task that called it to block until the string has been sent and the next highest priority task ready to run will be given control of the CPU in the interim. Since the user interface task will do little more than send character strings over the serial line, it will spend most of its time blocked.

The timekeeper task will use the TimeBlock function (native to the operating system) to relinquish control of the CPU for one second at a time. Every time the task is rescheduled, it will know that one second has passed, and it will add one second to the amount of time the system has been running for. Like the user interface task, the timekeeper task will spend most of its time blocked.

When the system starts up, the main function will begin executing. All initialization needs to be done inside the main function:

- Initialize peripheral devices (clock-tick interrupt and RS232 port)
- Create all three tasks.
- Call `StartExecution`

```
#include "helium.h"
#include <string.h>

// Declare the functions that hold program data
// for each task.
void Timekeeper();  // Pointer to program data for timekeeper task
void Idle() ;       // Pointer to program data for idle task
void ShellTask();   // Pointer to program data for shell task

void MCU_init(void) ;

char IdleStack[20] ;   // Stack space for idle task
char TimeStack[20] ;   // Stack space for timekeeper task
char ShellStack[50] ;  // Stack space for shell task

void main(void) {

  EnableInterrupts; /* enable interrupts */

  // Create all three tasks by calling TaskCreate.
  TaskCreate( 0, ShellTask, (int*)ShellStack, 50 ) ;
  TaskCreate( 1, Timekeeper, (int*)TimeStack, 20 ) ;
```

```
    TaskCreate( 2, Idle, (int*)IdleStack, 20 ) ;

    // Initialize serial port and clock-tick interrupt.
    MCU_init() ;

    // Hand control of the CPU over to ShellTask,
    // which is the highest priority task ready to run.
    StartExecution() ;

}
```

This listing includes pointers to the program data and stack for each of three tasks—a function declaration serves as a pointer to program data in Helium, and an array serves as a stack. The three tasks are created by passing a priority level, a pointer to the program data, a pointer to the stack, and the length of the stack in bytes.

After the tasks are created, the MCU's peripherals are initialized by calling an external function. Since the process of peripheral initialization is specific to the derivative and to the application, it will not be discussed in detail.

Once the tasks have been created and the clock-tick interrupt source is initialized, `StartExecution` is called. In this case, `StartExecution` will run the shell task because it has the highest priority level[1]. Once the shell task blocks, the timekeeper task will be run because it has the second highest priority level. When both the timekeeper task and the shell task are blocked, the idle task will be run because it has the lowest priority level.

After initialization, the implementation defines three tasks that run continuously. Note that all three tasks run in an infinite loop. No task should ever be allowed to execute a `return` statement. If a task needs to terminate itself, it should always call the API function `TaskDestroy`. Executing a `return` statement as would be typical in PC programs would cause Helium to crash.

The shell task begins by printing a welcome message. When the serial port driver function `sciPutStr` is called, the task will attempt to fill a buffer[2] with characters from the string. If for some reason the string in question will not fit in the transmit buffer (which by default is ten characters in length), the task will block until some characters have been sent and space is freed up.

When the task `ShellTask` blocks, Helium will start running `Timekeeper`, the second highest priority task. `Timekeeper` exists only to keep track of how long the system has been running for. It uses three global variables—hours, minutes, and seconds—to perform its task. Shortly after being started, `Timekeeper` will block for one second. The two tasks that actually perform work for the system are now blocked. In

---

[1] A task with priority level zero has the highest priority level, and the scheduler gives it control of the CPU whenever it is marked as ready to run.

[2] Refers to the serial port transmit buffer, `sciTxBuf` (defined in serial.c) which is an array of type `char` with a default length of 10. The buffer holds a queue of characters to be sent across the serial port. If a string to be sent across the serial port will fit completely in the transmit buffer, the task that requested to send it will not have to block until that string is completely sent before continuing its calculations. If, on the other hand, a string to be sent cannot fit completely in the transmit buffer, the task that requested to send it will have to be blocked until more space becomes available in the transmit buffer. Thus, the bigger the transmit buffer in bytes, the less the likelihood that a task will have to block when requesting to send a string.

order to keep the system functioning, Helium must be able to schedule a task. Since the Idle task never blocks, it can always be scheduled. The idle task should be present in every implementation that uses Helium v1.1. It should be the lowest priority task, and it should never block. It is used to prevent a system failure in the event that all other tasks need to block for some time interval. If the lowest priority task were to block and no other task would be ready to run, the system would begin exhibiting unexpected behavior.

```
void ShellTask(){

  char cmd[10] ;

  // Send a welcome message. This command will cause
  // ShellTask to block because the string to be transmitted
  // is longer than the transmit buffer.
  sciPutStr( "Welcome to Helium.\n" ) ;


  for(;;){
     sciGetLine( cmd, 10 ) ;
     if( strcmp(cmd, "time") == 0 ) PrintTime() ;
  }
}

void Timekeeper() {

   for( ;; ){
      if( seconds > 59 ) {
         seconds = 0 ;
         minutes++ ;
      }
      if( minutes > 59 ){
         minutes = 0 ;
         hours++ ;
      }

      TimeBlock( 100 ) ;
      seconds++ ;
   }
}

void Idle() {
  int idleCtr ;
  for(;;){
     DisableInterrupts ;
     idleCtr++ ;
     EnableInterrupts ;
  }
}
```