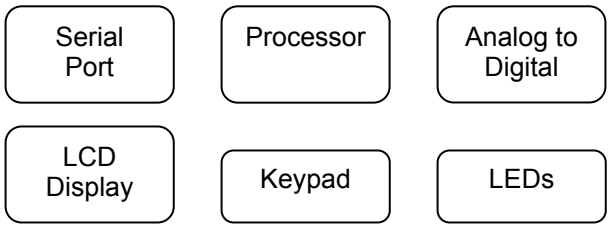
Comparing RTOS to Infinite Loop Designs

If you compare the way software is developed for a small to medium sized embedded project using a Real Time Operating System (RTOS) versus a traditional “infinite loop” type implementation, you will find that they are very similar. Yet, using a RTOS can provide significant advantages by reducing code complexity, increasing reliability and making the maintenance and addition of new features easier. Some embedded developers who have not used an RTOS may disagree with one or more of these benefits, but this document will provide an objective comparison for each point.

Lets say our embedded project is a data logger. When the project begins, we know it will have a keypad, simple LCD display, status LEDs, Analog-To-Digital sampling capability, and a RS-232 serial port. A typical development approach would be to implement and test each feature one at a time. This iterative procedure is the same whether or not a RTOS is used. So lets implement each task showing how it would be done both with and without a RTOS.



System Block Diagram

The Beginning

The first step is to just get a simple program to compile. At this point we don’t have any functional code, just the program shell so we can begin to add our functions.

No RTOS	Using RTOS

`main()`
`{`
`  while (1)`
`  {`
`  // no code yet`
`  }`
`}`
	`UserMain()`
	`{`
	`  iprintf(“Starting Program\r\n”);`
	`  while (1)`
	`  {`
	`  // no code yet`
	`  }`
	`}`

The first difference you will notice is the name of the main function. When using the RTOS, you begin in a task that the RTOS initialization code has set up. In this case it is called UserMain(). From this point forward you could actually use UserMain() in the exact same way as main() and there would be virtually no difference between them (other than the RTOS system clock tick interrupt routine running in the background). With no other tasks running, the UserMain() task would have complete control over the system just as main() does in the non-RTOS version.

Add Serial Port Functionality

Now lets add the code required to make the serial port operational. The serial port can be used for both a command interface and debugging during development. The serial port should be able to send and receive serial data as well as parse commands. For example, lets say that we want to have a simple interactive menu such as the following:

```
MAIN MENU
---------
1. Read A/D
2. Write GPIO outputs
3. Read GPIO inputs
4. Set LEDs
5. Write string to LCD display
```

The incoming command parser can be as simple as: `c = getchar()` , where c would be the value of 1-5. The difference in the function code below is that for the RTOS you would enclose the parsing code with an infinite while loop. More about this difference later.

```
void ProcessSerialPort()
{
   char s[80];

   while (1)   // for RTOS version only
   {
      char c = getchar();

      switch c
      {
         case 1: // Read and display ADC value
                 break;

         case 2: // Prompt for value and write to GPIO outputs
                 printf("Enter GPIO value: ");
                 fgets(s, 80, stdin);  // will block until user hits <enter>
                 // Convert s to a number and write GPIOs
                 break;

         case 3: // Read and display GPIO input values
                 break;

         case 4: // Prompt for value and write to LEDs
                 printf("Enter LED value: ");
                 fgets(s, 80, stdin);  // will block until user hits <enter>
                 // Convert s to a number and write LEDs
                 break;

         case 5: // Prompt for value and write to LCD display
                 printf("Enter display string: ");
                 fgets(s, 80, stdin);  // will block until user hits <enter>
                 // Write display string to LCD
                 break;

         default: // print error message
      }
   }
}
```

The complication here is that getchar() and all the functions that require the user to input a value (such as writing to the LCD display) are **blocking functions**, and in a non-RTOS environment we cannot block or the

main while loop will stop which will cause the system to be unresponsive. We can solve part of this problem by using a function to check whether or not a serial port character is available before we call the function, but that will only work for the command – in the process of writing a string to the LCD display will be a blocking function for a string input.

In contrast, we can use a RTOS and simply make the serial command parser into a task, which will only become active if there is a command to parse and process.

No RTOS	Using RTOS

`void ProcessSerialPort()`
`{`
`  // serial port code goes here`
`}`
 	
`main()`
`{`
`  while (1)`
`  {`
`  if ( SerialCharAvail() )`
`  ProcessSerialPort();`
`  }`
`}`
	`void ProcessSerialPort()`
	`{`
	`  // serial port code goes here`
	`}`
	
	`UserMain()`
	`{`
	`  iprintf("Starting Program\r\n");`
	
	`  OSCreateTask( ProcessSerialPort, SerialPrio );`
	
	`  while (1)`
	`  {`
	`  // no code yet`
	`  }`
	`}`

In the above code sets you can see that the ProcessSerialPort() function is declared for both methods. The difference is that the non-RTOS version requires the check for an available character before deciding to call the function. The problem remains that inside ProcessSerialPort() there are function calls that block in order to get text data to write the GPIOs, write the LEDs, and write to the LCD display. For example, in case 2 where a user can enter GPIO settings, the fgets() function will block until the user enters the data and hits the <enter> key. During this data entry period, the system is unresponsive using the non-RTOS implementation.

In the RTOS version, the OSCreateTask() function uses the ProcessSerialPort() function as a parameter. In effect, the ProcessSerialPort() function becomes the task. This new "task" will block on the getchar() function until data is received by the serial port. We do not need to add any code to the UserMain() infinite while loop, the RTOS will handle calling the ProcessSerialPort() code when necessary! Looking again at the case 2 example, a blocking function such as fgets() is not a problem since the RTOS will enable other tasks to run. This is accomplished without any coding on our part.

Add Keypad Functionality

The 4 x 4 keypad implementation will consist of 16 contacts in a matrix with 4 outputs and 4 inputs to the processor. Whenever a key is pressed it will create a signal on one of the 4 inputs, which will generate an interrupt and set a flag telling the system a key was pressed. The keypad matrix is then scanned by driving the outputs and reading the inputs in sequence to determine which key was pressed. All this must happen before the person releases the depressed key.

The functions for RTOS and non-RTOS are shown below. They both rely on an interrupt service routine to set a flag called "Keypressed". In the non-RTOS version the flag is checked in the main while loop. The RTOS implementation uses ProcessKeypad() as a task with it's own infinite while loop, and the check on the "Keypressed" flag is inside the ProcessKeypad() function. In RTOS lingo we use a Semaphore to be the flag and a blocking function called OSSemPend() to wait for the Semaphore to be set.

No RTOS	Using RTOS

`void ProcessKeypad()`
`{`
`  char key = ScanKeypad();`
`  // code to process key goes here`
`}`
	`void ProcessKeypad()`
	`{`
	`  while (1)`
	`  {`
	`  OSSemPend( Keypressed );`
	`  char key = ScanKeypad();`
	`  // code to process key goes here`
	`  OSSemPost( Keypressed );`
	`  }`
	`}`

No RTOS	**Using RTOS**

`void ProcessSerialPort()`
`{`
`  // serial port code goes here`
`}`
 	
`void ProcessKeypad()`
`{`
`  // keypad code goes here`
`}`
 	
`main()`
`{`
`  while (1)`
`  {`
`  if ( SerialCharAvail() )`
`  ProcessSerialPort();`
 	
`  if ( Keypressed )`
`  ProcessKeypad();`
 	
`  }`
`}`
	`void ProcessSerialPort()`
	`{`
	`  // serial port code goes here`
	`}`
	`void ProcessKeypad()`
	`{`
	`  // keypad code goes here`
	`}`
	`UserMain()`
	`{`
	`  iprintf(“Starting Program\r\n”);`
	
	`  OSCreateTask( ProcessSerialPort, SerialPrio );`
	`  OSCreateTask( ProcessKeypad, KeypadPrio );`
	
	`  while (1)`
	`  {`
	`  // no code yet`
	`  }`
	`}`

Again, the above code sets are similar in that they both create a function called ProcessKeypad() that scans the keypad and processes the keypress. In the non-RTOS implementation we check the “keypressed” flag that would be set the an interrupt service routine to decide whether or not to call the ProcessKeypad() function. In the RTOS implementation, we simply add another task that is ProcessKeypad().

There is a major difference here worth noting. Let’s say that we are using the non-RTOS implementation. The serial port interface is used to select menu item #5 to enter a text string that will be sent to the LCD display, and the user has only entered half of the characters. This is a blocked state and the main while loop is completely stalled. If a user types characters on the keypad, they will be missed. One way to around this would be to write a more complex interrupt service routine that scanned the keypad and buffered the keypress (essentially what the ProcessKeypad() function does) but now the interrupt routine is taking a lot of time and the system latency goes up. While this might be acceptable if all we needed was to handle the keypad, what about all the other interrupts such as a system timer, analog to digital conversion, serial port, etc.? The further a programmer goes down this path, the closer he is to creating a task scheduler, which is one of the primary features of an RTOS! The major difference is that the RTOS has a large user base and has been debugged.
If we consider the RTOS implementation in which the task that processes keypad input is a higher priority than the serial port, the system will immediately switch from the ProcessSerialPort() task to the ProcessKeypad() task, process the keypad entry, and jump right back to the ProcessSerialPort() task so the operator can continue to enter characters that will be sent to the LCD display. This should happen so fast that it is transparent to the user.

Add Analog To Digital Functionality

To implement the ADC we will use an interrupt to signal that a sample reading is available. An interrupt service routine will then read the value and set a flag indicating a new value is ready to be read. As with the keypad processing, the non-RTOS version can use a flag, and the RTOS version can use a semaphore. Once a ADC value has been read it is multiplied by a calibration value and stored in an array. The functions are shown below:

No RTOS	Using RTOS

`void ProcessADC()`
`{`
`  if ( SampleReady )`
`  // code to read and store ADC value`
`}`
	`void ProcessADC()`
	`{`
	`  while (1)`
	`  {`
	`  OSSemPend( SampleReady );`
	`  // code to read and store ADC value`
	`  }`
	`}`

The program now looks like:

No RTOS	Using RTOS

`void ProcessSerialPort()`
`{`
`  // serial port code goes here`
`}`
 	
`void ProcessKeypad()`
`{`
`  // keypad code goes here`
`}`
 	
`void ProcessADC()`
`{`
`  // code to store ADC value`
`}`
 	
`main()`
`{`
`  while (1)`
`  {`
`  if ( SerialCharAvail() )`
`  ProcessSerialPort();`
 	
`  if ( Keypressed )`
`  ProcessKeypad();`
 	
`  if ( SampleReady )`
`  ProcessADC();`
 	
`  }`
`}`
	`void ProcessSerialPort()`
	`{`
	`  // serial port code goes here`
	`}`
	
	`void ProcessKeypad()`
	`{`
	`  // keypad code goes here`
	`}`
	
	`void ProcessADC()`
	`{`
	`  // code to store ADC value`
	`}`
	
	`UserMain()`
	`{`
	`  OSCreateTask( ProcessSerialPort, SerialPrio );`
	`  OSCreateTask( ProcessKeypad, KeypadPrio );`
	`  OSCreateTask( ProcessADC, ADCPrio );`
	
	`  while (1)`
	`  {`
	`  // no code yet`
	`  }`
	`}`

There are two concerns here for the non-RTOS implementation. First, there is the possibility a reading could be lost because the main while loop did not call the ProcessADC() function before the next ADC reading was done. Second, there is the possibility of contention or data corruption if both the interrupt routine and the ProcessADC

code access the global adc_val at the same time. A mechanism for mutual exclusion to adc_val is required to prevent this situation.

The RTOS implementation solves both these problems. As with the keypad, the priority of the ProcessADC() task would be set so that it will take momentary control from the serial port task to quickly process the sample. Mutual exclusion of the adc_val is handled by the OSSemPend() and OSSemPost(), so data corruption cannot occur. Then end result is that with very little coding we have guaranteed the sample will be processed in time, and the sample data will not be corrupted.

As a side note, the RTOS also provides a more advantageous way to handle this case. Instead of a semaphore or flag that gets set, we could read the ADC values in the ISR and store them in a Queue. Calling OSQPend() will block until one or more samples have been read and are in the Queue. The code would look like this:

```
void ProcessADC()
{
   while (1)
   {
      OSQPend( Sample, timeout, status );
      // code to store the Sample value obtained above
    }
}
```

All we had to do was change the OSSemPend() to OSQPend(). If for any reason the ProcessADC() task could not catch up with the interrupt service routine, multiple readings would be stored in the Queue. If more than 1 reading is in the Queue, then the OSQPend() function would return immediately so the additional readings could be processed.

Add GPIO, LED and LCD Display Functionality

The GPIO, LED and LCD display functions in this example would be called from the functions we have already defined in response to key presses, serial input or sample readings. They probably would not need to be called directly from the main while loop or be tasks of their own.

CONCLUSION

Writing a small to medium size embedded application can be nearly identical from a code function standpoint whether an infinite main loop or RTOS is used. In many cases when a RTOS is not used, a programmer must write the equivalent of a limited scheduler to handle events. The advantage of a RTOS is that a programmer can use a tested and reliable solution to achieve the same goal. In addition, the RTOS supplies features such as priority, semaphores and queues that make the structure of the application much easier than if a RTOS was not used.