Solution #1

Why Use an RTOS Kernel?

<<< | >>>

See also the FAQ item "Why use an RTOS?".

Synopsis

Many applications can be produced without the use of an RTOS kernel and this page describes an approach that might be taken.

Even though the application in this case is probably too complex for such an approach the page is included to both highlight the potential problems and provide a contrast to the following RTOS based software designs.

Implementation

This solution uses a traditional infinite loop approach, whereby each component of the application is represented by a function that executes to completion.

Ideally a hardware timer would be used to schedule the time critical plant control function. However, having to wait for the arrival of data and the complex calculation performed make the control function unsuitable for execution within an interrupt

service routine.

Concept of Operation

The frequency and order in which the components are called within the infinite loop can be modified to introduce some prioritisation. A couple of such sequencing alternatives are provided in the example below.

Scheduler Configuration

The RTOS scheduler is not used.

Evaluation

 Small code size.

 No reliance on third party source code.

 No RTOS RAM, ROM or processing overhead.

 Difficult to cater for complex timing requirements.

 Does not scale well without a large increase in complexity.

 Timing hard to evaluate or maintain due to the interdependencies between the different functions.

FreeRTOS+ Lab Projects
FreeRTOS+POSIX:
POSIX threading API
FreeRTOS+FAT:
Thread aware file system

Conclusion

The simple loop approach is very good for small applications and applications with flexible timing requirements - but can become complex, difficult to analyse and difficult to maintain if scaled to larger systems.

Example

This example is a partial implementation of the hypothetical application introduced previously.

The Plant Control Function

The control function can be represented by the following pseudo code:

```
void PlantControlCycle( void )
{
    TransmitRequest();
    WaitForFirstSensorResponse();

    if( Got data from first sensor )
    {
        WaitForSecondSensorResponse();

        if( Got data from second sensor )
        {
            PerformControlAlgorithm();
            TransmitResults();
        }
    }
}
```

The Human Interface Functions

This includes the keypad, LCD, RS232 communications and embedded web server.

The following pseudo code represents a simple infinite loop structure for controlling these interfaces.

```
int main( void )
{
    Initialise();

    for( ;; )
    {
        ScanKeypad();
        UpdateLCD();
        ProcessRS232Characters();
        ProcessHTTPRequests();
    }

    // Should never get here.
    return 0;
}
```

This assumes two things: First, The communications IO is buffered by interrupt service routines so peripherals do not require polling. Second, the individual function calls within the loop execute quickly enough for all the maximum timing requirements to be met.

Scheduling the Plant Control Function

The length of the control function means it cannot simply be called from a 10ms timer interrupt.

Adding it to the infinite loop would require the introduction of some temporal control. For example ... :

```
// Flag used to mark the time at which a
// control cycle should start (mutual exclusion
// issues being ignored for this example).
int TimerExpired;

// Service routine for a timer interrupt.  This
// is configured to execute every 10ms.
void TimerInterrupt( void )
{
    TimerExpired = true;
}
```

```
// Main() still contains the infinite loop -
// within which a call to the plant control
// function has been added.
int main( void )
{
    Initialise();

    for( ;; )
    {
        // Spin until it is time for the next
        // cycle.
        if( TimerExpired )
        {
            PlantControlCycle();
            TimerExpired = false;

            ScanKeypad();
            UpdateLCD();

            // The LEDs could use a count of
            // the number of interrupts, or a
            // different timer.
            ProcessLEDs();

            // Comms buffers must be large
            // enough to hold 10ms worth of
            // data.
            ProcessRS232Characters();
            ProcessHTTPRequests();
        }

        // The processor can be put to sleep
        // here provided it is woken by any
        // interrupt.
    }

    // Should never get here.
    return 0;
}
```

... but this is not an acceptable solution:

- A delay or fault on the field bus results in an increased execution time of the plant control function. The timing requirements of the interface functions would most likely be breached.

- Executing all the functions each cycle could also result in a breach of the control cycle timing.
- Jitter in the execution time may cause cycles to be missed. For example the execution time of `ProcessHTTPRequests()` could be negligible when no HTTP requests have been received, but quite lengthy when a page was being served.
- It is not very maintainable - it relies on every function being executed within the maximum time.
- The communication buffers are only serviced once per cycle necessitating their length to be larger than would otherwise be necessary.

Alternative Structures

Two factors can be identified that limit the suitability of the simple loop structure described so far.

1. The length of each function call

   Allowing each function to execute in its entirety takes too long. This can be prevented by splitting each function into a number of states. Only one state is executed each call. Using the control function as an example:

```
// Define the states for the control cycle function.
typdef enum eCONTROL_STATES
{
    eStart, // Start new cycle.
    eWait1, // Wait for the first sensor response.
    eWait2  // Wait for the second sensor response.
} eControlStates;

void PlantControlCycle( void )
{
static eControlState eState = eStart;

    switch( eState )
    {
        case eStart :
            TransmitRequest();
            eState = eWait1;
```

```
                break;

            case eWait1;
                if( Got data from first sensor )
                {
                    eState = eWait2;
                }
                // How are time outs to be handled?
                break;

            case eWait2;
                if( Got data from first sensor )
                {
                    PerformControlAlgorithm();
                    TransmitResults();

                    eState = eStart;
                }
                // How are time outs to be handled?
                break;
        }
    }
```

This function is now structurally more complex, and introduces further scheduling problems. The code itself will become harder to understand as extra states are added - for example to handle timeout and error conditions.

2. The granularity of the timer

   A shorter timer interval will give more flexibility.

   Implementing the control function as a state machine (an in so doing making each call shorter) may allow it to be called from a timer interrupt. The timer interval will have to be short enough to ensure the function gets called at a frequency that meets its timing requirements. This option is fraught with timing and maintenance problems.

   Alternatively the infinite loop solution could be modified to call different functions on each loop - with the high priority control function called more frequently:

```
int main( void )
{
int Counter = -1;

    Initialise();
```

```
    // Each function is implemented as a state
    // machine so is guaranteed to execute
    // quickly - but must be called often.

    // Note the timer frequency has been raised.

    for( ;; )
    {
        if( TimerExpired )
        {
            Counter++;

            switch( Counter )
            {
                case 0  : ControlCycle();
                          ScanKeypad();
                          break;

                case 1  : UpdateLCD();
                          break;

                case 2  : ControlCycle();
                          ProcessRS232Characters();
                          break;

                case 3  : ProcessHTTPRequests();

                          // Go back to start
                          Counter = -1;
                          break;

            }

            TimerExpired = false;
        }
    }

    // Should never get here.
    return 0;
}
```

More intelligence can be introduced by means of event counters, whereby the lower priority functionality is only called if an event has occurred that requires servicing:

```
for( ;; )
{
    if( TimerExpired )
    {
        Counter++;

        // Process the control cycle every other loop.
        switch( Counter )
        {
            case 0  : ControlCycle();
                      break;

            case 1  : Counter = -1;
                      break;
        }

        // Process just one of the other functions.  Only process
        // a function if there is something to do.  EventStatus()
        // checks for events since the last iteration.
        switch( EventStatus() )
        {
            case EVENT_KEY  :   ScanKeypad();
                                UpdateLCD();
                                break;

            case EVENT_232  :   ProcessRS232Characters();
                                break;

            case EVENT_TCP  :   ProcessHTTPRequests();
                                break;
        }

        TimerExpired = false;
    }
}
```

Processing events in this manner will reduce wasted CPU cycles but the design will still exhibit jitter in the frequency at which the control cycle executes.

NEXT >>> Solution #2: A fully preemptive system