An RTOS for small embedded systems.
Homepage

Only use this page if your browser does not support frames

If you browser supports frames all this information is contained in the menu frame on the left. Click the homepage link above if you cannot see the menu frame.

FreeRTOS™ Modules

Here is a list of all modules:

An RTOS for small embedded systems.

Fundamentals

Modules

An RTOS for small embedded systems.
Homepage

Design

[FreeRTOS Fundamentals]

Features

The following standard features are provided.

- Choice of RTOS scheduling policy
  1. Pre-emptive:
     Always runs the highest available task. Tasks of identical priority share CPU time (fully pre-emptive with round robin time slicing).
  2. Cooperative:
     Context switches only occur if a task blocks, or explicitly calls taskYIELD().
- Message queues
- Semaphores [via macros]
- Trace visualisation ability (requires more RAM)
- Majority of source code common to all supported development tools

Additional features can quickly and easily be added.

Design Philosophy

FreeRTOS is designed to be:

- Simple
- Portable
- Concise

Nearly all the code is written in C, with only a few assembler functions where completely unavoidable. This does not result in tightly optimized code, but does mean the code is readable, maintainable and easy to port. If performance were an issue it could easily be improved at the cost of portability. This will not be necessary for most applications.

The RTOS kernel uses multiple priority lists. This provides maximum application design flexibility. Unlike bitmap kernels any number of tasks can share the same priority.

An RTOS for small embedded systems.
Homepage

Tasks & Priorities
[FreeRTOS Fundamentals]

Real Time Task Priorities

Low priority numbers denote low priority tasks, with the default idle priority defined by tskIDLE_PRIORITY as being zero.

The number of available priorities is defined by tskMAX_PRIORITIES within `FreeRTOSConfig.h`. This should be set to suit your application.

Any number of real time tasks can share the same priority - facilitating application design. User tasks can also share a priority of zero with the idle task.

Priority numbers should be chosen to be as close and as low as possible. For example, if your application has 3 user tasks that must all be at different priorities then use priorities 3 (highest), 2 and 1 (lowest - the idle task uses priority 0).

Implementing a Task

A task should have the following structure:

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }
}
```

The type pdTASK_CODE is defined as a function that returns void and takes a void pointer as it's only parameter. All functions that implement a task should be of this type. The parameter can be used to pass any information into the task - see the RTOS demo application files for examples.

Task functions should never return so are typically implemented as a continuous loop. Again, see the RTOS demo application for numerous examples.

Tasks are created by calling xTaskCreate() and deleted by calling vTaskDelete().

Task functions can optionally be defined using the portTASK_FUNCTION and portTASK_FUNCTION_PROTO macros. These macro are provided to allow compiler specific syntax to be added to the function definition and prototype respectively. Their use is not required unless specifically stated in documentation for the port being used (currently only the PIC18 fedC port).

The prototype for the function shown above can be written as:

```
void vATaskFunction( void *pvParameters );
```

Or,

```
portTASK_FUNCTION_PROTO( vATaskFunction, pvParameters );
```

Likewise the function above could equally be written as:

```
portTASK_FUNCTION( vATaskFunction, pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }
}
```

The Idle Task

The idle task is created automatically by the first call to xTaskCreate ().

The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the vTaskDelete() function to ensure the idle task is not starved of processing time. The activity visualisation utility can be used to check the microcontroller time allocated to the idle task.

The idle task has no other active functions so can legitimately be starved of microcontroller time under all other conditions.

It is acceptable for application tasks to share the idle task priority. (tskIDLE_PRIORITY).

The Idle Task Hook

An idle task hook is a function that is called during each cycle of the idle task. If you want application functionality to run at the idle priority then there are two options:

1. Implement the functionality in an idle task hook.

   There must always be at least one task that is ready to run. It is therefore imperative that the hook function does not call any API functions that might cause the task to block (vTaskDelay() for example).

2. Create an idle priority task to implement the functionality.

   This is a more flexible solution but has a higher RAM usage overhead.

See the Embedded software application design section for more information on using an idle hook.

To create an idle hook:

1. Set configUSE_IDLE_HOOK to 1 within `FreeRTOSConfig.h`.

2. Define a function that has the following prototype:

   ```
   void vApplicationIdleHook( void );
   ```

A common use for an idle hook is to simply put the processor into a power saving mode.

Start/Stopping the Real Time Kernel

The real time kernel is started by calling vTaskStartScheduler(). The call will not return unless an application task calls vTaskEndScheduler() or the function cannot complete.

See the RTOS demo application file main.c for examples of creating tasks and starting/stopping the kernel.

An RTOS for small embedded systems.
Homepage

Trace Utility

[Real Time Tasks & Priorities]

The trace visualisation utility allows the RTOS activity to be examined.

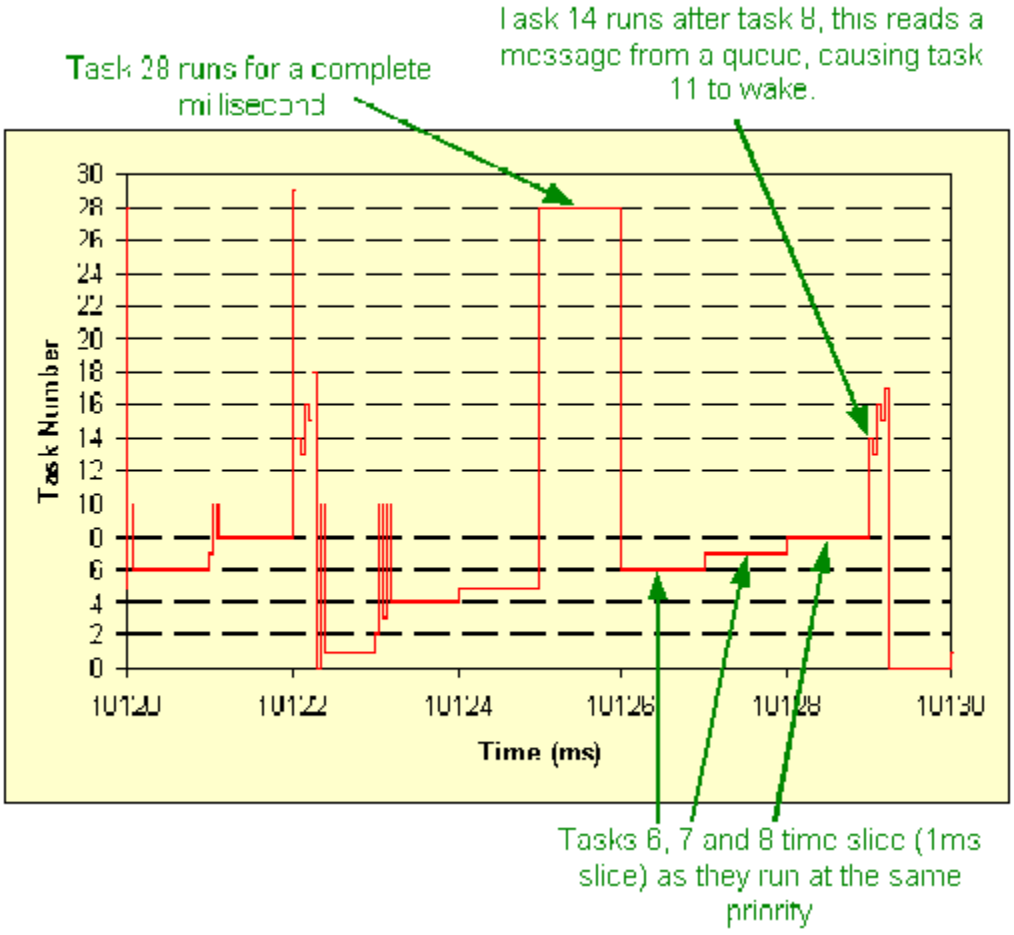
It records the sequence in which tasks are given microcontroller processing time.

To use the utility the macro configUSE_TRACE_FACILITY must be defined as 1 within `FreeRTOSConfig.h` when the application is compiled. See the *configuration* section in the RTOS API documentation for more information.

The trace is started by calling vTaskStartTrace() and ended by calling ulTaskEndTrace(). It will end automatically if it's buffer becomes full.

The completed trace buffer can be stored to disk for offline examination. The DOS/Windows utility tracecon.exe converts the stored buffer to a tab delimited text file. This can then be opened and examined in a spread sheet application.

Below is a 10 millisecond example output collected from the AMD 186 demo application. The x axis shows the passing of time, and the y axis the number of the task that is running.



Each task is automatically allocated a number when it is created. The idle task is always number 0. vTaskList () can be used to obtain the number allocated to each task, along with some other useful information. The information returned by vTaskList() during the demo application is shown below, where:

- Name - is the name given to the task when it was created. Note that the demo application creates more than one instance of some tasks.
- State - shows the state of a task. This can be either 'B'locked, 'R'eady, 'S'uspended or 'D'eleted.
- Priority - is the priority given to the task when it was created.
- Stack - shows the high water mark of the task stack. This is the minimum amount of free stack that has been available during the lifetime of the task.
- Num - is the number automatically allocated to the task.

Name	State	Priority	Stack	Num
Print	R	4	331	29
Math7	R	0	417	7
Math8	R	0	407	8
QConsB2	R	0	53	14
QProdB5	R	0	52	17
QConsB4	R	0	53	16
SEM1	R	0	50	27
SEM1	R	0	50	28
IDLE	R	0	64	0
Math1	R	0	436	1
Math2	R	0	436	2
Math3	R	0	417	3
Math4	R	0	407	4
Math5	R	0	436	5
Math6	R	0	436	6
QProdNB	B	2	52	12
LEDx	B	1	63	19
LEDx	B	1	74	22
LEDx	B	1	73	20
LEDx	B	1	63	25
LEDx	B	1	73	21
LEDx	B	1	63	24
COMTx	B	2	44	9
LEDx	B	1	63	26
LEDx	B	1	67	23
SUICIDE1	B	3	215	55
SUICIDE2	B	3	215	56
SUICIDE1	B	3	215	57
SUICIDE2	B	3	215	58
CREATOR	B	3	170	30
QProdB1	B	3	53	13
QConsB6	B	0	52	18
QProdB3	B	3	54	15
COMRx	B	3	51	10
QConsNB	B	2	57	11

In this example, it can be seen that tasks 6, 7, 8 and 14 are all running at priority 0. They therefore time slice between themselves and the other priority 0 tasks (including the idle task). Task 14 reads a message from a queue (see BlockQ.c in the demo application). This frees a space on the queue. Task 13 was blocked waiting for a space to be available, so now wakes, posts a message then blocks again.

Note: In it's current implementation, the time resolution of the trace is equal to the tick rate. Context switches can occur more frequently than the system tick (if a task blocks for example). When this occurs the trace will show that a context switch has occurred and will accurately shows the context switch sequencing. However, the timing of context switches that occur between system ticks cannot accurately be recorded. The ports could easily be modified to provide a higher resolution time stamp by making use of a free running timer.

An RTOS for small embedded systems.
Homepage

RTOS Kernel Utilities
[FreeRTOS Fundamentals]

Queue Implementation

Items are placed in a queue by copy - not by reference. It is therefore preferable, when queuing large items, to only queue a pointer to the item.

RTOS demo application files blockq.c and pollq.c demonstrate queue usage.

The queue implementation used by the RTOS is also available for application code.

Semaphore Implementation

Binary semaphore functionality is provided by a set of macros.

The macros use the queue implementation as this provides everything necessary with no extra code or testing overhead.

The macros can easily be extended to provide counting semaphores if required.

The RTOS demo application file semtest.c demonstrates semaphore usage. Also see the RTOS API documentation.

An RTOS for small embedded systems.
Homepage

API

Modules

- New for V3.0.0
- Configuration
- Task Creation
- Task Control
- Kernel Control
- Task Utilities
- Queue Management
- Semaphores

An RTOS for small embedded systems.
Homepage

Configuration
[API]

Modules

- Customisation
- Memory Management

An RTOS for small embedded systems.
Homepage

Customisation
[Configuration]

A number of configurable parameters exist that allow the FreeRTOS kernel to be tailored to your particular application. These items are located in a file called FreeRTOSConfig.h. Each demo application included in the FreeRTOS source code download has its own FreeRTOSConfig.h file. Here is a typical example, followed by an explanation of each parameter:

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/* Here is a good place to include header files that are required across
your application. */
#include "something.h"

#define configUSE_PREEMPTION           1
#define configUSE_IDLE_HOOK            0
#define configCPU_CLOCK_HZ             58982400
#define configTICK_RATE_HZ             250
#define configMAX_PRIORITIES           5
#define configMINIMAL_STACK_SIZE       128
#define configTOTAL_HEAP_SIZE          10240
#define configMAX_TASK_NAME_LEN        16
#define configUSE_TRACE_FACILITY       0
#define configUSE_16_BIT_TICKS         0
#define configIDLE_SHOULD_YIELD        1

#define INCLUDE_vTaskPrioritySet       1
#define INCLUDE_uxTaskPriorityGet      1
#define INCLUDE_vTaskDelete            1
#define INCLUDE_vTaskCleanUpResources  0
#define INCLUDE_vTaskSuspend           1
#define INCLUDE_vTaskDelayUntil        1
#define INCLUDE_vTaskDelay             1

#endif /* FREERTOS_CONFIG_H */
```

'config' Parameters

configUSE_PREEMPTION

Set to 1 to use the preemptive kernel, or 0 to use the cooperative kernel.

configUSE_IDLE_HOOK

Set to 1 if you wish to use an idle hook, or zero to omit an idle hook.

configCPU_CLOCK_HZ

Enter the frequency in Hz at which the *internal* processor core will be executing. This value is required in order to correctly configure timer peripherals.

configTICK_RATE_HZ

The frequency of the RTOS tick interrupt.

The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the kernel and is higher than would normally be required.

More than one task can share the same priority. The kernel will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

configMAX_PRIORITIES

The number of priorities available to the application. Any number of tasks can share the same priority.

Each available priority consumes RAM within the kernel so this value should not be set any higher than actually required by your application.

configMINIMAL_STACK_SIZE

The size of the stack used by the idle task. Generally this should not be reduced from the value set in the FreeRTOSConfig.h file provided with the demo application for the port you are using.

configTOTAL_HEAP_SIZE

The total amount of RAM available to the kernel.

This value will only be used if your application makes use of one of the sample memory allocation schemes provided in the FreeRTOS source code download. See the memory configuration section for further details.

configMAX_TASK_NAME_LEN

The maximum permissible length of the descriptive name given to a task when the task is created. The length is specified in the number of characters *including* the NULL termination byte.

configUSE_TRACE_FACILITY

Set to 1 if you wish the trace visualisation functionality to be available, or zero if the trace functionality is not going to be used. If you use the trace functionality a trace buffer must also be provided.

configUSE_16_BIT_TICKS

Time is measured in 'ticks' - which is the number of times the tick interrupt has executed since the kernel was started. The tick count is held in a variable of type portTickType.

Defining configUSE_16_BIT_TICKS as 1 causes portTickType to be defined (typedef'ed) as an unsigned 16bit type. Defining configUSE_16_BIT_TICKS as 0 causes portTickType to be defined (typedef'ed) as an unsigned 32bit type.

Using a 16 bit type will greatly improve performance on 8 and 16 bit architectures, but limits the maximum specifiable time period to 65535 'ticks'. Therefore, assuming a tick frequency of 250Hz, the maximum time a task can delay or block when a 16bit counter is used is 262 seconds, compared to 17179869 seconds when using a 32bit counter.

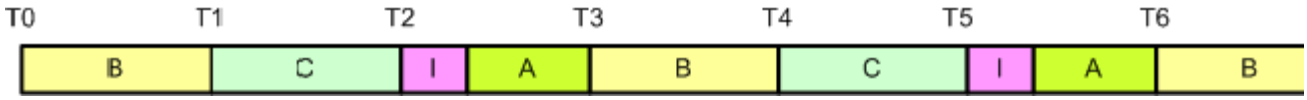
configIDLE_SHOULD_YIELD

This parameter controls the behaviour of tasks at the idle priority. It only has an effect if:

1. The preemptive scheduler is being used.
2. The users application creates tasks that run at the idle priority.

Tasks that share the same priority will time slice. Assuming none of the tasks get preempted, it might be assumed that each task of at a given priority will be allocated an equal amount of processing time - and if the shared priority is above the idle priority then this is indeed the case.

When tasks share the idle priority the behaviour can be slightly different. When configIDLE_SHOULD_YIELD is set to 1 the idle task will yield immediately should any other task at the idle priority be ready to run. This ensures the minimum amount of time is spent in the idle task when application tasks are available for scheduling. This behaviour can however have undesirable effects (depending on the needs of your application) as depicted below:



This diagram shows the execution pattern of four tasks at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task. A context switch occurs with regular period at times T0, T1, ..., T6. When the idle task yields task A starts to execute - but the idle task has already taken up some of the current time slice. This results in task I and task A effectively sharing a time slice. The application tasks B and C therefore get more processing time than the application task A.

This situation can be avoided by:

- If appropriate, using an idle hook in place of separate tasks at the idle priority.
- Creating all application tasks at a priority greater than the idle priority.
- Setting configIDLE_SHOULD_YIELD to 0.

Setting configIDLE_SHOULD_YIELD prevents the idle task from yielding processing time until the end of its time slice. This ensure all tasks at the idle priority are allocated an equal amount of processing time - but at the cost of a greater proportion of the total processing time being allocated to the idle task.

INCLUDE Parameters

The macros starting 'INCLUDE' allow those components of the real time kernel not utilized by your application to be excluded from your build. This ensures the RTOS does not use any more ROM or RAM than necessary for your particular embedded application.

Each macro takes the form ...

```
INCLUDE_FunctionName
```

... where FunctionName indicates the API function (or set of functions) that can optionally be excluded. To include the API function set the macro to 1, to exclude the function set the macro to 0. For example, to include the vTaskDelete() API function use:

```
#define INCLUDE_vTaskDelete    1
```

To exclude vTaskDelete() from your build use:

```
#define INCLUDE_vTaskDelete    0
```

Homepage

Memory Management
[Configuration]

The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. The malloc() and free() functions can sometimes be used for this purpose, but ...

1. they are not always available on embedded systems,
2. take up valuable code space,
3. are not thread safe, and
4. are not deterministic (the amount of time taken to execute the function will differ from call to call)

... so more often than not an alternative scheme is required.

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem the memory allocation API is included in the RTOS portable layer - where an application specific implementation appropriate for the real time system being developed can be provided. When the real time kernel requires RAM, instead of calling malloc() it makes a call to pvPortMalloc(). When RAM is being freed, instead of calling free() the real time kernel makes a call to vPortFree().

Schemes included in the source code download

Three sample RAM allocation schemes are included in the FreeRTOS source code download (V2.5.0 onwards). These are used by the various demo applications as appropriate. The following subsections describe the available schemes, when they should be used, and highlight the demo applications that demonstrate their use.

Each scheme is contained in a separate source file (heap_1.c, heap_2.c and heap_3.c respectively) which can be located in the `Source/Portable/MemMang` directory. Other schemes can be added if required.

Scheme 1 - heap_1.c

This is the simplest scheme of all. It does *not* permit memory to be freed once it has been allocated, but despite this is suitable for a surprisingly large number of applications.

The algorithm simply subdivides a single array into smaller blocks as requests for RAM are made. The total size of the array is set by the definition configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

This scheme:

- Can be used if your application never deletes a task or queue (no calls to vTaskDelete() or vQueueDelete() are ever made).
- Is always deterministic (always takes the same amount of time to return a block).
- Is used by the PIC, AVR and 8051 demo applications - as these do not dynamically create or delete tasks after vTaskStartScheduler() has been called.

heap_1.c is suitable for a lot of small real time systems provided that all tasks and queues are created before the kernel is started.

Scheme 2 - heap_2.c

This scheme uses a best fit algorithm and, unlike scheme 1, allows previously allocated blocks to be freed. It does *not* however combine adjacent free blocks into a single large block.

Again the total amount of available RAM is set by the definition configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

This scheme:

- Can be used even when the application repeatedly calls vTaskCreate()/vTaskDelete() or vQueueCreate()/vQueueDelete() (causing multiple calls to pvPortMalloc() and vPortFree()).
- Should *not* be used if the memory being allocated and freed is of a random size - this would only be the case if tasks being deleted each had a different stack depth, or queues being deleted were of different lengths.
- Could possible result in memory fragmentation problems should your application create blocks of queues and tasks in an unpredictable order. This would be unlikely for nearly all applications but should be kept in mind.
- Is not deterministic - but is also not particularly inefficient.
- Is used by the ARM7, and Flashlite demo applications - as these dynamically create and delete tasks.

heap_2.c is suitable for most small real time systems that have to dynamically create tasks.

Scheme 3 - heap_3.c

This is just a wrapper for the standard malloc() and free() functions. It makes them thread safe.

This scheme:

- Requires the linker to setup a heap, and the compiler library to provide malloc() and free() implementations.
- Is not deterministic.
- Will probably considerably increase the kernel code size.
- Is used by the PC (x86 single board computer) demo application.

A Free RTOS for small embedded systems.

Homepage | FAQ

FreeRTOS FAQ - Memory and Memory Management

How much RAM does FreeRTOS use?

This depends on your application. Below is a guide based on an 8bit architecture:

Item	Bytes Used
Scheduler Itself	83
For each priority add	16
For each queue you create, add	45 + queue storage area (see FAQ Why do queues use that much RAM?)
For each task you create, add	20 (includes 2 characters for the task name) + the task stack size.
For each semaphore you create, add	45

Why do queues use that much RAM?

Event management is built into the queue functionality. This means the queue data structures contain all the RAM that other RTOS systems sometimes allocate separately. There is no concept of an event control block within FreeRTOS.

How much ROM does FreeRTOS use?

This depends on your compiler and architecture.

Using GCC and the AVR as an example, the base kernel including queues and semaphores consumes approximately 4.4KBytes.

It is difficult to compare this across other RTOS systems as no two are identical in their functionality or performance.

How can I reduce the amount of RAM used?

- Set configMAX_PRIORITIES and configMINIMAL_STACK_SIZE (found in portmacro.h) to the minimum values acceptable to your application.

- If supported by the compiler - define task functions and main() as "naked". This prevents the compiler saving registers to the stack when the function is entered. As the function will never be exited the registers will never get restored and are not required.

- Recover the stack used by main(). The stack used upon program entry is not required once the scheduler has been started (unless your application calls vTaskEndScheduler(), which is only supported directly in the distribution for the PC and Flashlite ports). Every task has it's own stack allocated so the stack allocated to main() is available for reuse once the scheduler has started.

- Minimise the stack used by main(). The idle task is automatically created when you create the first application task. The stack used upon program entry (before the scheduler has started) must therefore be large enough for a nested call to xTaskCreate(). Creating the idle task manually can half this stack requirement. To create the idle task manually:

    1. Locate the function prvInitialiseTaskLists() in Source\tasks.c.
    2. The idle task is created at the bottom of the function by a call to xTaskCreate(). Cut this line from Source\tasks.c and paste it into main().

- Rationalise the number of tasks. The idle task is not required if:

    1. Your application has a task that never blocks, and ...
    2. Your application does not make any calls to vTaskDelete().

- There are other minor tweaks that can be performed (for example the task priority queues don't require event management), but if you get down to this level - you need more RAM!

How is RAM allocated to tasks?

To create a task the kernel makes two calls to pvPortMalloc(). The first allocates the task control block, the second allocates the task stack.

Please read the memory configuration section of the API documentation for details of the allocation scheme.

How is RAM allocated to queues?

To create a queue the kernel makes two calls to pvPortMalloc(). The first allocates the queue structure, the second the queue storage area (the size of which is a parameter to xQueueCreate()).

How big should the stack be?

This is completely application dependent, and not always easy to calculate. It depends on the function call depth, number of local variables allocated, number of parameters in function calls made, interrupt stack requirements, etc. The stack must always be large enough to contain the execution context (all the processor registers).

The stack of each task is filled with 0xa5 bytes upon creation which allows the high water mark to be viewed using suitable debugging tools. Also see the function usPortCheckFreeStackSpace() implemented in the x86 port for an example of how the high watermark can be measured.

An RTOS for small embedded systems.
Homepage

Task Creation
[API]

Modules

- xTaskCreate
- vTaskDelete

Detailed Description

---

xTaskHandle

task. h

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an xTaskHandle variable that can then be used as a parameter to vTaskDelete to delete the task.

An RTOS for small embedded systems.
Homepage

xTaskCreate
[Task Creation]

task. h

```
portBASE_TYPE xTaskCreate(
                            pdTASK_CODE pvTaskCode,
                            const portCHAR * const pcName,
                            unsigned portSHORT usStackDepth,
                            void *pvParameters,
                            unsigned portBASE_TYPE uxPriority,
                            xTaskHandle *pvCreatedTask
                          );
```

Create a new task and add it to the list of tasks that are ready to run.

Parameters:

pvTaskCode	Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
pcName	A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by tskMAX_TASK_NAME_LEN - default is 16.
usStackDepth	The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.
pvParameters	Pointer that will be used as the parameter for the task being created.
uxPriority	The priority at which the task should run.
pvCreatedTask	Used to pass back a handle by which the created task can be referenced.

Returns:

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs. h

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
unsigned char ucParameterToPass;
xTaskHandle xHandle;

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY, &xH

    // Use the handle to delete the task.
    vTaskDelete( xHandle );
}
```

An RTOS for small embedded systems.
Homepage

vTaskDelete
[Task Creation]

task. h

```
void vTaskDelete( xTaskHandle pxTask );
```

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

Remove a task from the RTOS real time kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death. c for sample code that utilises vTaskDelete ().

Parameters:

pxTask The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

```
void vOtherFunction( void )
{
xTaskHandle xHandle;

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // Use the handle to delete the task.
    vTaskDelete( xHandle );
}
```

An RTOS for small embedded systems.
Homepage

Task Control
[API]

Modules

- vTaskDelay
- vTaskDelayUntil
- uxTaskPriorityGet
- vTaskPrioritySet
- vTaskSuspend
- vTaskResume

An RTOS for small embedded systems.
Homepage

vTaskDelay
[Task Control]

task. h

```
void vTaskDelay( portTickType xTicksToDelay );
```

INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters:

xTicksToDelay The amount of time, in tick periods, that the calling task should block.

Example usage:

```
// Perform an action every 10 ticks.
// NOTE:
// This is for demonstration only and would be better achieved
// using vTaskDelayUntil().
void vTaskFunction( void * pvParameters )
{
portTickType xDelay, xNextTime;

    // Calc the time at which we want to perform the action
    // next.
    xNextTime = xTaskGetTickCount () + ( portTickType ) 10;

    for( ;; )
    {
        xDelay = xNextTime - xTaskGetTickCount ();
        xNextTime += ( portTickType ) 10;

        // Guard against overflow
        if( xDelay <= ( portTickType ) 10 )
        {
            vTaskDelay( xDelay );
        }

        // Perform action here.
    }
}
```

An RTOS for small embedded systems.
Homepage

vTaskDelayUntil
[Task Control]

task. h

```
void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement );
```

INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

Delay a task until a specified time. This function can be used by cyclical tasks to ensure a constant execution frequency.

This function differs from vTaskDelay() in one important aspect: vTaskDelay() will cause a task to block for the specified number of ticks from the time vTaskDelay() is called. It is therefore difficult to use vTaskDelay() by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTaskDelay() may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.

The constant configTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters:

pxPreviousWakeTime	Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use (see the example below). Following this the variable is automatically updated within vTaskDelayUntil().
xTimeIncrement	The cycle time period. The task will be unblocked at time (*pxPreviousWakeTime + xTimeIncrement). Calling vTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interval period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
portTickType xLastWakeTime;
const portTickType xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```

An RTOS for small embedded systems.
Homepage

uxTaskPriorityGet
[Task Control]

task. h

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

INCLUDE_vTaskPriorityGet must be defined as 1 for this function to be available. See the configuration section for more information.

Obtain the priority of any task.

Parameters:

pxTask Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns:

The priority of pxTask.

Example usage:

```
void vAFunction( void )
{
xTaskHandle xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to obtain the priority of the created task.
    // It was created with tskIDLE_PRIORITY, but may have changed
    // it itself.
    if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
    {
        // The task has changed it's priority.
    }

    // ...

    // Is our priority higher than the created task?
    if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
    {
        // Our priority (obtained using NULL handle) is higher.
    }
}
```

An RTOS for small embedded systems.
Homepage

vTaskPrioritySet
[Task Control]

task. h

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

INCLUDE_vTaskPrioritySet must be defined as 1 for this function to be available. See the configuration section for more information.

Set the priority of any task.

A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

Parameters:

pxTask	Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
uxNewPriority	The priority to which the task will be set.

Example usage:

```
void vAFunction( void )
{
xTaskHandle xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to raise the priority of the created task.
    vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

    // ...

    // Use a NULL handle to raise our priority to the same value.
    vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}
```

Homepage

vTaskSuspend

[Task Control]

task. h

```
void vTaskSuspend( xTaskHandle pxTaskToSuspend );
```

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Parameters:

pxTaskToSuspend Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

Example usage:

```
void vAFunction( void )
{
xTaskHandle xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );

    // ...

    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).

    //...

    // Suspend ourselves.
    vTaskSuspend( NULL );

    // We cannot get here unless another task calls vTaskResume
    // with our handle as the parameter.
}
```

trade marks of Richard Barry.

An RTOS for small embedded systems.
Homepage

vTaskResume

[Task Control]

task. h

```
void vTaskResume( xTaskHandle pxTaskToResume );
```

INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

Resumes a suspended task.

A task that has been suspended by one of more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Parameters:

pxTaskToResume Handle to the task being readied.

Example usage:

```
void vAFunction( void )
{
xTaskHandle xHandle;

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );

    // ...

    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).

    //...

    // Resume the suspended task ourselves.
    vTaskResume( xHandle );

    // The created task will once again get microcontroller processing
    // time in accordance with it priority within the system.
}
```

An RTOS for small embedded systems.
Homepage

Kernel Control
[API]

Modules

- vTaskStartScheduler
- vTaskEndScheduler
- vTaskSuspendAll
- xTaskResumeAll

Detailed Description

---

taskYIELD

task. h

Macro for forcing a context switch.

---

taskENTER_CRITICAL

task. h

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

---

taskEXIT_CRITICAL

task. h

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

---

taskDISABLE_INTERRUPTS

task. h

Macro to disable all maskable interrupts.

taskENABLE_INTERRUPTS

task. h

Macro to enable microcontroller interrupts.

An RTOS for small embedded systems.
Homepage

vTaskStartScheduler

[Kernel Control]

task. h

```
void vTaskStartScheduler( void );
```

Starts the real time kernel tick processing. After calling the kernel has control over which tasks are executed and when.

The idle task is created automatically when vTaskStartScheduler() is called.

If vTaskStartScheduler() is successful the function will not return until an executing task calls vTaskEndScheduler(). The function might fail and return immediately if there is insufficient RAM available for the idle task to be created.

See the demo application file main. c for an example of creating tasks and starting the kernel.

Example usage:

```
void vAFunction( void )
{
    // Create at least one task before starting the kernel.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );

    // Start the real time kernel with preemption.
    vTaskStartScheduler();

    // Will not get here unless a task calls vTaskEndScheduler ()
}
```

An RTOS for small embedded systems.
Homepage

vTaskEndScheduler
[Kernel Control]

task. h

```
void vTaskEndScheduler( void );
```

Stops the real time kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where vTaskStartScheduler() was called, as if vTaskStartScheduler() had just returned.

See the demo application file main. c in the demo/PC directory for an example that uses vTaskEndScheduler ().

vTaskEndScheduler () requires an exit function to be defined within the portable layer (see vPortEndScheduler () in port. c for the PC port). This performs hardware specific operations such as stopping the kernel tick.

vTaskEndScheduler () will cause all of the resources allocated by the kernel to be freed - but will not free resources allocated by application tasks.

Example usage:

```
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // At some point we want to end the real time kernel processing
        // so call ...
        vTaskEndScheduler ();
    }
}

void vAFunction( void )
{
    // Create at least one task before starting the kernel.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );

    // Start the real time kernel with preemption.
    vTaskStartScheduler();

    // Will only get here when the vTaskCode () task has called
    // vTaskEndScheduler ().  When we get here we are back to single task
    // execution.
}
```

An RTOS for small embedded systems.
Homepage

vTaskSuspendAll
[Kernel Control]

task. h

```
void vTaskSuspendAll( void );
```

Suspends all real time kernel activity while keeping interrupts (including the kernel tick) enabled.

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

Example usage:

```
 void vTask1( void * pvParameters )
 {
     for( ;; )
     {
         // Task code goes here.

         // ...

         // At some point the task wants to perform a long operation during
         // which it does not want to get swapped out.  It cannot use
         // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
         // operation may cause interrupts to be missed - including the
         // ticks.

         // Prevent the real time kernel swapping out the task.
         vTaskSuspendAll ();

         // Perform the operation here.  There is no need to use critical
         // sections as we have all the microcontroller processing time.
         // During this time interrupts will still operate and the kernel
         // tick count will be maintained.

         // ...

         // The operation is complete.  Restart the kernel.
         xTaskResumeAll ();
     }
 }
```

Homepage

xTaskResumeAll

[Kernel Control]

task. h

```
portBASE_TYPE xTaskResumeAll( void );
```

Resumes real time kernel activity following a call to vTaskSuspendAll (). After a call to xTaskSuspendAll () the kernel will take control of which task is executing at any time.

Returns:

If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out.  It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        xTaskSuspendAll ();

        // Perform the operation here.  There is no need to use critical
        // sections as we have all the microcontroller processing time.
        // During this time interrupts will still operate and the real
        // time kernel tick count will be maintained.

        // ...

        // The operation is complete.  Restart the kernel.  We want to force
        // a context switch - but there is no point if resuming the scheduler
        // caused a context switch already.
        if( !xTaskResumeAll () )
        {
             taskYIELD ();
        }
    }
}
```

An RTOS for small embedded systems.
Homepage

Task Utilities
[API]

---

xTaskGetTickCount

task.h

```
volatile portTickType xTaskGetTickCount( void );
```

Returns:
The count of ticks since vTaskStartScheduler was called.

---

uxTaskGetNumberOfTasks

task.h

```
unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void );
```

Returns:
The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not yet freed by the idle task will also be included in the count.

---

vTaskList

task.h

```
void vTaskList( portCHAR *pcWriteBuffer );
```

configUSE_TRACE_FACILITY, INCLUDE_vTaskDelete and INCLUDE_vTaskSuspend must all be defined as 1 for this function to be available. See the configuration section for more information.

NOTE: This function will disable interrupts for its duration. It is not intended for normal application runtime use but as a debug aid.

Lists all the current tasks, along with their current state and stack usage high water mark.

Tasks are reported as blocked ('B'), ready ('R'), deleted ('D') or suspended ('S').

Parameters:
pcWriteBuffer A buffer into which the above mentioned details will be written, in ascii form. This buffer is assumed to be large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.

---

vTaskStartTrace

task.h

```
void vTaskStartTrace( portCHAR * pcBuffer, unsigned portLONG ulBufferSize );
```

Starts a real time kernel activity trace. The trace logs the identity of which task is running when.

The trace file is stored in binary format. A separate DOS utility called convtrce.exe is used to convert this into a tab delimited text file which can be viewed and plotted in a spread sheet.

Parameters:

pcBuffer	The buffer into which the trace will be written.
ulBufferSize	The size of pcBuffer in bytes. The trace will continue until either the buffer in full, or ulTaskEndTrace() is called.

---

ulTaskEndTrace

task.h

```
unsigned portLONG ulTaskEndTrace( void );
```

Stops a kernel activity trace. See vTaskStartTrace().

Returns:

The number of bytes that have been written into the trace buffer.

An RTOS for small embedded systems.
Homepage

Queue Management
[API]

Modules

- xQueueCreate
- xQueueSend
- xQueueReceive
- xQueueSendFromISR
- xQueueReceiveFromISR

Detailed Description

---

uxQueueMessagesWaiting

queue.h

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

Return the number of messages stored in a queue.

Parameters:
xQueue A handle to the queue being queried.

Returns:
The number of messages available in the queue.

---

vQueueDelete

queue.h

```
void vQueueDelete( xQueueHandle xQueue );
```

Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

Parameters:
xQueue A handle to the queue to be deleted.

An RTOS for small embedded systems.
Homepage

xQueueCreate

[Queue Management]

queue. h

```
xQueueHandle xQueueCreate(
                            unsigned portBASE_TYPE uxQueueLength,
                            unsigned portBASE_TYPE uxItemSize
                          );
```

Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

Parameters:

uxQueueLength	The maximum number of items that the queue can contain.
uxItemSize	The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns:

If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Example usage:

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
};

void vATask( void *pvParameters )
{
xQueueHandle xQueue1, xQueue2;

   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );
   if( xQueue1 == 0 )
   {
       // Queue was not created and must not be used.
   }

   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue2 == 0 )
   {
       // Queue was not created and must not be used.
   }

   // ... Rest of task code.
}
```

An RTOS for small embedded systems.
Homepage

xQueueSend

[Queue Management]

queue.h

```
portBASE_TYPE xQueueSend(
                            xQueueHandle xQueue,
                            const void * pvItemToQueue,
                            portTickType xTicksToWait
                        );
```

Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine. See xQueueSendFromISR() for an alternative which may be used in an ISR.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

Returns:

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

Example usage:

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

unsigned portLONG ulVar = 10UL;

void vATask( void *pvParameters )
{
xQueueHandle xQueue1, xQueue2;
struct AMessage *pxMessage;

   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // Send an unsigned long.  Wait for 10 ticks for space to become
       // available if necessary.
```

```
        if( xQueueSend( xQueue1, ( void * ) &ulVar, ( portTickType ) 10 ) != pdPASS )
        {
            // Failed to post the message, even after 10 ticks.
        }
    }

    if( xQueue2 != 0 )
    {
        // Send a pointer to a struct AMessage object.  Don't block if the
        // queue is already full.
        pxMessage = & xMessage;
        xQueueSend( xQueue2, ( void * ) &pxMessage, ( portTickType ) 0 );
    }

     // ... Rest of task code.
}
```

An RTOS for small embedded systems.
Homepage

xQueueReceive

[Queue Management]

queue. h

```
portBASE_TYPE xQueueReceive(
                                xQueueHandle xQueue,
                                void *pcBuffer,
                                portTickType xTicksToWait
                            );
```

Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

This function must not be used in an interrupt service routine. See xQueueReceiveFromISR for an alternative that can.

Parameters:

pxQueue	The handle to the queue from which the item is to be received.
pcBuffer	Pointer to the buffer into which the received item will be copied.
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

Returns:

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

xQueueHandle xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;

   // Create a queue capable of containing 10 pointers to AMessage structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
       // Failed to create the queue.
   }

   // ...

   // Send a pointer to a struct AMessage object.  Don't block if the
   // queue is already full.
   pxMessage = & xMessage;
```

```
    xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );

         // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // Receive a message on the created queue.  Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable posted
            // by vATask.
        }
    }

         // ... Rest of task code.
}
```

An RTOS for small embedded systems.
Homepage

xQueueSendFromISR
[Queue Management]

queue.h

```
portBASE_TYPE xQueueSendFromISR(
                                   xQueueHandle pxQueue,
                                   const void *pvItemToQueue,
                                   portBASE_TYPE xTaskPreviouslyWoken
                                );
```

Post an item on a queue. It is safe to use this function from within an interrupt service routine.

Items are queued by copy not reference so it is preferable to only queue small items, especially when called from an ISR. In most cases it would be preferable to store a pointer to the item being queued.

Parameters:

xQueue	The handle to the queue on which the item is to be posted.
pvItemToQueue	A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
xTaskPreviouslyWoken	This is included so an ISR can post onto the same queue multiple times from a single interrupt. The first call should always pass in pdFALSE. Subsequent calls should pass in the value returned from the previous call. See the file serial .c in the PC port for a good example of this mechanism.

Returns:

pdTRUE if a task was woken by posting onto the queue. This is used by the ISR to determine if a context switch may be required following the ISR.

Example usage for buffered IO (where the ISR can obtain more than one value per call):

```
void vBufferISR( void )
{
portCHAR cIn;
portBASE_TYPE xTaskWokenByPost;

   // We have not woken a task at the start of the ISR.
   xTaskWokenByPost = pdFALSE;

   // Loop until the buffer is empty.
   do
   {
       // Obtain a byte from the buffer.
       cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

       // Post the byte.  The first time round the loop xTaskWokenByPost
       // will be pdFALSE.  If the queue send causes a task to wake we do
       // not want the task to run until we have finished the ISR, so
       // xQueueSendFromISR does not cause a context switch.  Also we
       // don't want subsequent posts to wake any other tasks, so we store
       // the return value back into xTaskWokenByPost so xQueueSendFromISR
       // knows not to wake any task the next iteration of the loop.
       xTaskWokenByPost = xQueueSendFromISR( xRxQueue, &cIn, xTaskWokenByPost );
```

```
    } while( portINPUT_BYTE( BUFFER_COUNT ) );

    // Now the buffer is empty we can switch context if necessary.
    if( xTaskWokenByPost )
    {
        taskYIELD();
    }
}
```

An RTOS for small embedded systems.
Homepage

xQueueReceiveFromISR
[Queue Management]

queue. h

```
portBASE_TYPE xQueueReceiveFromISR(
                                    xQueueHandle pxQueue,
                                    void *pcBuffer,
                                    portBASE_TYPE *pxTaskWoken
                                );
```

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Parameters:

pxQueue	The handle to the queue from which the item is to be received.
pcBuffer	Pointer to the buffer into which the received item will be copied.
pxTaskWoken	A task may be blocked waiting for space to become available on the queue. If xQueueReceiveFromISR causes such a task to unblock *pxTaskWoken will get set to pdTRUE, otherwise *pxTaskWoken will remain unchanged.

Returns:

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Example usage:

```
xQueueHandle xQueue;

// Function to create a queue and post some values.
void vAFunction( void *pvParameters )
{
portCHAR cValueToPost;
const portTickType xBlockTime = ( portTickType )0xff;

   // Create a queue capable of containing 10 characters.
   xQueue = xQueueCreate( 10, sizeof( portCHAR ) );
   if( xQueue == 0 )
   {
       // Failed to create the queue.
   }

   // ...

   // Post some characters that will be used within an ISR.  If the queue
   // is full then this task will block for xBlockTime ticks.
   cValueToPost = 'a';
   xQueueSend( xQueue, ( void * ) &cValueToPost, xBlockTime );
   cValueToPost = 'b';
   xQueueSend( xQueue, ( void * ) &cValueToPost, xBlockTime );

   // ... keep posting characters ... this task may block when the queue
   // becomes full.

   cValueToPost = 'c';
   xQueueSend( xQueue, ( void * ) &cValueToPost, xBlockTime );
}
```

```
// ISR that outputs all the characters received on the queue.
void vISR_Routine( void )
{
portBASE_TYPE xTaskWokenByReceive = pdFALSE;
portCHAR cRxedChar;

   while( xQueueReceiveFromISR( xQueue, ( void * ) &cRxedChar, &xTaskWokenByReceive) )
   {
       // A character was received.  Output the character now.
       vOutputCharacter( cRxedChar );

       // If removing the character from the queue woke the task that was
       // posting onto the queue xTaskWokenByReceive will have been set to
       // pdTRUE.  No matter how many times this loop iterates only one
       // task will be woken.
   }

   if( xTaskWokenByPost != pdFALSE )
   {
       taskYIELD ();
   }
}
```

An RTOS for small embedded systems.
Homepage

Semaphores
[API]

Modules

- vSemaphoreCreateBinary
- xSemaphoreTake
- xSemaphoreGive
- xSemaphoreGiveFromISR

An RTOS for small embedded systems.
Homepage

vSemaphoreCreateBinary
[Semaphores]

semphr. h

```
vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )
```

Macro that implements a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

Parameters:

xSemaphore Handle to the created semaphore. Should be of type xSemaphoreHandle.

Example usage:

```
xSemaphoreHandle xSemaphore;

void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to vSemaphoreCreateBinary ().
   // This is a macro so pass the variable in directly.
   vSemaphoreCreateBinary( xSemaphore );

   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```

An RTOS for small embedded systems.
Homepage

xSemaphoreTake
[Semaphores]

semphr. h

```
xSemaphoreTake(
                  xSemaphoreHandle xSemaphore,
                  portTickType xBlockTime
              )
```

Macro to obtain a semaphore. The semaphore must of been created using vSemaphoreCreateBinary ().

Parameters:

xSemaphore A handle to the semaphore being obtained. This is the handle returned by vSemaphoreCreateBinary ();

xBlockTime The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

Returns:

pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

Example usage:

```
xSemaphoreHandle xSemaphore = NULL;

// A task that creates a semaphore.
void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   vSemaphoreCreateBinary( xSemaphore );
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.

   if( xSemaphore != NULL )
   {
       // See if we can obtain the semaphore.  If the semaphore is not available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE )
       {
           // We were able to obtain the semaphore and can now access the
           // shared resource.

           // ...

           // We have finished accessing the shared resource.  Release the
           // semaphore.
           xSemaphoreGive( xSemaphore );
       }
       else
       {
           // We could not obtain the semaphore and can therefore not access
```

```
            // the shared resource safely.
        }
    }
}
```

An RTOS for small embedded systems.
Homepage

xSemaphoreGive
[Semaphores]

semphr. h

```
xSemaphoreGive( xSemaphoreHandle xSemaphore )
```

Macro to release a semaphore. The semaphore must of been created using vSemaphoreCreateBinary (), and obtained using sSemaphoreTake ().

This must not be used from an ISR. See xSemaphoreGiveFromISR() for an alternative which can be used from an ISR.

Parameters:

xSemaphore A handle to the semaphore being released. This is the handle returned by vSemaphoreCreateBinary ();

Returns:

pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Example usage:

```
xSemaphoreHandle xSemaphore = NULL;

void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.
   vSemaphoreCreateBinary( xSemaphore );

   if( xSemaphore != NULL )
   {
       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
       {
           // We would expect this call to fail because we cannot give
           // a semaphore without first "taking" it!
       }

       // Obtain the semaphore - don't block if the semaphore is not
       // immediately available.
       if( xSemaphoreTake( xSemaphore, ( portTickType ) 0 ) )
       {
           // We now have the semaphore and can access the shared resource.

           // ...

           // We have finished accessing the shared resource so can free the
           // semaphore.
           if( xSemaphoreGive( xSemaphore ) != pdTRUE )
           {
               // We would not expect this call to fail because we must have
               // obtained the semaphore to get here.
           }
       }
   }
}
```

An RTOS for small embedded systems.
Homepage

xSemaphoreGiveFromISR
[Semaphores]

semphr. h

```
xSemaphoreGiveFromISR(
                          xSemaphoreHandle xSemaphore,
                          portBASE_TYPE xTaskPreviouslyWoken
                      )
```

Macro to release a semaphore. The semaphore must of been created using vSemaphoreCreateBinary(), and obtained using xSemaphoreTake().

This macro can be used from an ISR.

Parameters:

xSemaphore	A handle to the semaphore being released. This is the handle returned by vSemaphoreCreateBinary ();
xTaskPreviouslyWoken	This is included so an ISR can make multiple calls to xSemaphoreGiveFromISR() from a single interrupt. The first call should always pass in pdFALSE. Subsequent calls should pass in the value returned from the previous call. See the file serial .c in the PC port for a good example of using xSemaphoreGiveFromISR().

Returns:

pdTRUE if a task was woken by releasing the semaphore. This is used by the ISR to determine if a context switch may be required following the ISR.

Example usage:

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT    10
xSemaphoreHandle xSemaphore = NULL;

// Repetitive task.
void vATask( void * pvParameters )
{
   for( ;; )
   {
       // We want this task to run every 10 ticks or a timer.  The semaphore
       // was created before this task was started

       // Block waiting for the semaphore to become available.
       if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
       {
           // It is time to execute.

           // ...

           // We have finished our task.  Return to the top of the loop where
           // we will block on the semaphore until it is time to execute
           // again.
       }
   }
}

// Timer ISR
```

```
void vTimerISR( void * pvParameters )
{
static unsigned portCHAR ucLocalTickCount = 0;
static portBASE_TYPE xTaskWoken = pdFALSE;

   // A timer tick has occurred.

   // ... Do other time functions.

   // Is it time for vATask () to run?
   ucLocalTickCount++;
   if( ucLocalTickCount >= TICKS_TO_WAIT )
   {
       // Unblock the task by releasing the semaphore.
       xTaskWoken = xSemaphoreGiveFromISR( xSemaphore, xTaskWoken );

       // Reset the count so we release the semaphore again in 10 ticks time.
       ucLocalTickCount = 0;
   }

   // If xTaskWoken was set to true you may want to yield (force a switch)
   // here.
}
```

Free RTOS: AVR, Microchip PIC, x86, ARM7, 8051, ...!

Homepage

FreeRTOS Implementation Modules

Only use this page if your browser does not support frames

If you browser supports frames all this information is contained in the menu frame on the left.

Here is a list of all the pages:

Open source RTOS for small real time systems

Homepage

RTOS Implementation

This section describes part of the FreeRTOS implementation.

The pages will be helpful if you:

- wish to modify the FreeRTOS source code.
- port the real time kernel to another microcontroller or prototyping board.
- are new to using an RTOS and wish to get more information on their operation and implementation.

There are two chapters accessible from the menu frame on the left:

1. *- Fundamentals and RTOS concepts*

   This contains background information on multitasking and basic real time concepts and is intended for beginners.

2. *- RTOS Implementation*

   This explains the real time kernel source code from the bottom up.

   The FreeRTOS real time kernel has been ported to a number of different microcontroller architectures. The Atmel AVR port was chosen for this example due to:

   - the simplicity of the AVR architecture.
   - the free availability of the utilized WinAVR (GCC) development tools.
   - the low cost of the STK500 prototyping board

   The section concludes with a detailed step by step look at one complete context switch.

For further information see the FreeRTOS ColdFire Implementation Report. This was written by the Motorola ColdFire port authors, and details both the ColdFire source code and the development process undertaken in producing the port.

Go directly to www.FreeRTOS.org if you cannot see the menu on the left, alternatively click here if your browser does not support frames.

Next: Section 1 - RTOS fundamentals
... Next: Section 2 - RTOS Implementation Example

Open source RTOS for small embedded systems

1.RTOS Fundamentals

Pages

Detailed Description

This section provides a very brief introduction to real time and multitasking concepts. These must be understood before reading section 2.

Next: RTOS Fundamentals - Multitasking

Open source RTOS for small embedded systems

Homepage

Multitasking

[1.RTOS Fundamentals]

The **kernel** is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.

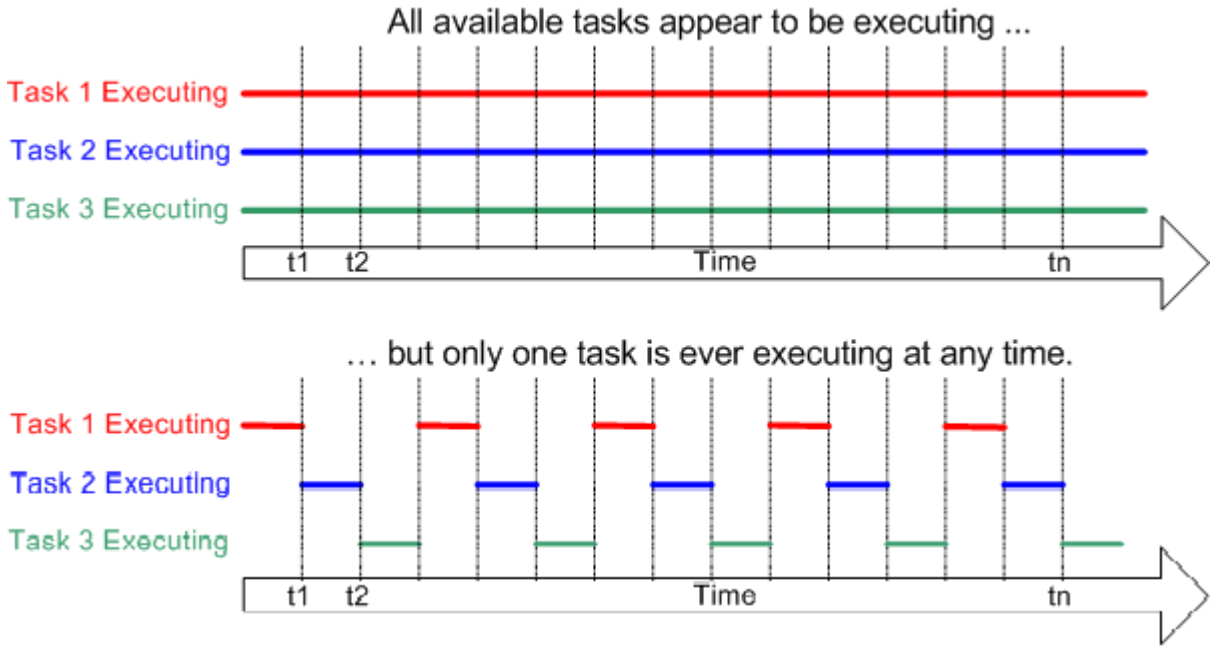
Each executing program is a **task** under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be **multitasking**.

The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

- The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks.
- The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

Multitasking Vs Concurrency

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it **appear** as if each task is executing concurrently. This is depicted by the diagram below which shows the execution pattern of three tasks with respect to time. The task names are color coded and written down the left hand. Time moves from left to right, with the colored lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.



Next: RTOS Fundamentals - Scheduling

Open source RTOS for small embedded systems

Homepage

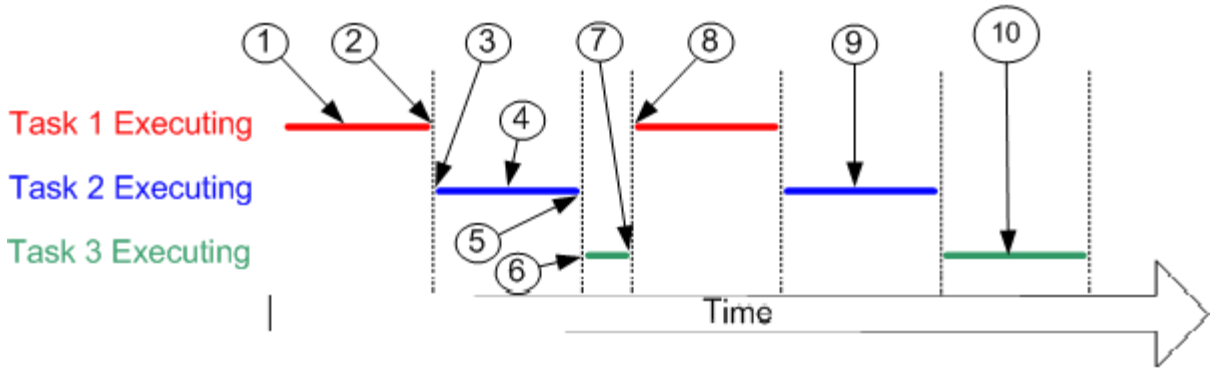
Scheduling

[1.RTOS Fundamentals]

The **scheduler** is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime.

The **scheduling policy** is the algorithm used by the scheduler to decide which task to execute at any point in time. The policy of a (non real time) multi user system will most likely allow each task a "fair" proportion of processor time. The policy used in real time / embedded systems is described later.

In addition to being suspended involuntarily by the RTOS kernel a task can choose to suspend itself. It will do this if it either wants to delay (**sleep**) for a fixed period, or wait (**block**) for a resource to become available (eg a serial port) or an event to occur (eg a key press). A blocked or sleeping task is not able to execute, and will not be allocated any processing time.



Referring to the numbers in the diagram above:

- At (1) task 1 is executing.
- At (2) the kernel suspends task 1 ...
- ... and at (3) resumes task 2.
- While task 2 is executing (4), it locks a processor peripheral for it's own exclusive access.
- At (5) the kernel suspends task 2 ...
- ... and at (6) resumes task 3.
- Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).
- At (8) the kernel resumes task 1.
- Etc.
- The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.
- The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

Next: RTOS Fundamentals - Context Switching

Open source RTOS for small embedded systems

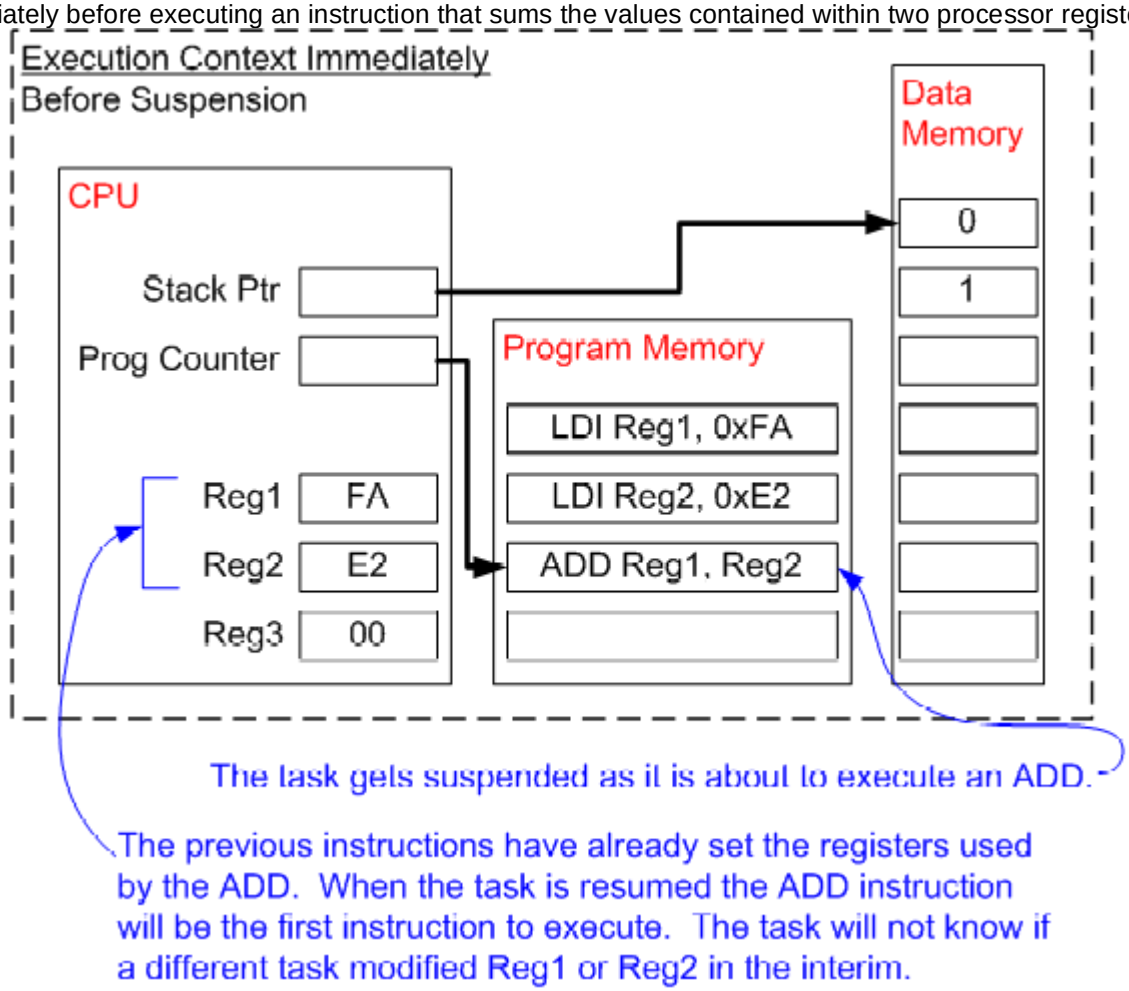
Homepage

Context Switching

[1.RTOS Fundamentals]

As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution **context**.

A task is a sequential piece of code - it does not know when it is going to get suspended or resumed by the kernel and does not even know when this has happened. Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two processor registers.



While the task is suspended other tasks will execute and may modify the processor register values. Upon resumption the task will not know that the processor registers have been altered - if it used the modified values the summation would result in an incorrect value.

To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The operating system kernel is responsible for ensuring this is the case - and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the operating system kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called **context switching**.

Next: RTOS Fundamentals - Real Time Applications

Open source RTOS for small embedded systems

Homepage

Real Time Applications

[1.RTOS Fundamentals]

Real time operating systems (**RTOS**'s) achieve multitasking using these same principals - but their objectives are very different to those of non real time systems. The different objective is reflected in the scheduling policy. Real time / embedded systems are designed to provide a timely response to real world events. Events occurring in the real world can have deadlines before which the real time / embedded system must respond and the RTOS scheduling policy must ensure these deadlines are met.

To achieve this objective the software engineer must first assign a priority to each task. The scheduling policy of the RTOS is then to simply ensure that the highest priority task that is able to execute is the task given processing time. This may require sharing processing time "fairly" between tasks of equal priority if they are ready to run simultaneously.

Example:

The most basic example of this is a real time system that incorporates a keypad and LCD. A user must get visual feedback of each key press within a reasonable period - if the user cannot see that the key press has been accepted within this period the software product will at best be awkward to use. If the longest acceptable period was 100ms - any response between 0 and 100ms would be acceptable. This functionality could be implemented as an autonomous task with the following structure:

```
void vKeyHandlerTask( void *pvParameters )
{
    // Key handling is a continuous process and as such the task
    // is implemented using an infinite loop (as most real time
    // tasks are).
    for( ;; )
    {
        [Suspend waiting for a key press]

        [Process the key press]
    }
}
```

Now assume the real time system is also performing a control function that relies on a digitally filtered input. The input must be sampled, filtered and the control cycle executed every 2ms. For correct operation of the filter the temporal regularity of the sample must be accurate to 0.5ms. This functionality could be implemented as an autonomous task with the following structure:

```
void vControlTask( void *pvParameters )
{
    for( ;; )
    {
        [Suspend waiting for 2ms since the start of the previous
        cycle]

        [Sample the input]
        [Filter the sampled input]
        [Perform control algorithm]
        [Output result]
    }
}
```

The software engineer must assign the control task the highest priority as:

1. The deadline for the control task is stricter than that of the key handling task.
2. The consequence of a missed deadline is greater for the control task than for the key handler task.

The next page demonstrates how these tasks would be scheduled by a real time operating system.

Next: RTOS Fundamentals - Real Time Scheduling

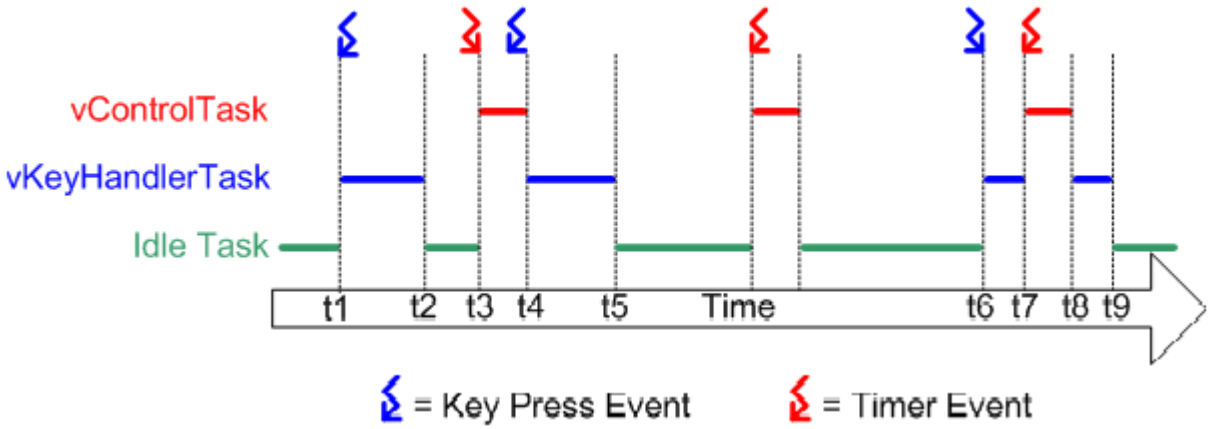
Open source RTOS for small embedded systems

Homepage

Real Time Scheduling

[1.RTOS Fundamentals]

The diagram below demonstrates how the tasks defined on the previous page would be scheduled by a real time operating system. The RTOS has itself created a task - the **idle** task - which will execute only when there are no other tasks able to do so. The RTOS idle task is always in a state where it is able to execute.



Referring to the diagram above:

- At the start neither of our two tasks are able to run - vControlTask is waiting for the correct time to start a new control cycle and vKeyHandlerTask is waiting for a key to be pressed. Processor time is given to the RTOS idle task.
- At time t1, a key press occurs. vKeyHandlerTask is now able to execute - it has a higher priority than the RTOS idle task so is given processor time.
- At time t2 vKeyHandlerTask has completed processing the key and updating the LCD. It cannot continue until another key has been pressed so suspends itself and the RTOS idle task is again resumed.
- At time t3 a timer event indicates that it is time to perform the next control cycle. vControlTask can now execute and as the highest priority task is scheduled processor time immediately.
- Between time t3 and t4, while vControlTask is still executing, a key press occurs. vKeyHandlerTask is now able to execute, but as it has a lower priority than vControlTask it is not scheduled any processor time.
- At t4 vControlTask completes processing the control cycle and cannot restart until the next timer event - it suspends itself. vKeyHandlerTask is now the task with the highest priority that is able to run so is scheduled processor time in order to process the previous key press.
- At t5 the key press has been processed, and vKeyHandlerTask suspends itself to wait for the next key event. Again neither of our tasks are able to execute and the RTOS idle task is scheduled processor time.
- Between t5 and t6 a timer event is processed, but no further key presses occur.
- The next key press occurs at time t6, but before vKeyHandlerTask has completed processing the key a timer event occurs. Now both tasks are able to execute. As vControlTask has the higher priority vKeyHandlerTask is suspended before it has completed processing the key, and vControlTask is scheduled processor time.
- At t8 vControlTask completes processing the control cycle and suspends itself to wait for the next. vKeyHandlerTask is again the highest priority task that is able to run so is scheduled processor time so the key press processing can be completed.

Next: Section 2 - RTOS Implementation

Open source RTOS for small embedded systems

2.RTOS Implementation

Pages

Detailed Description

This section describes the RTOS context switch source code from the bottom up. The FreeRTOS Atmel AVR microcontroller port is used as an example. The section ends with a detailed step by step look at one complete context switch.

Next: RTOS Implementation - Development Tools

A free RTOS for small embedded systems!

Building Blocks

[2.RTOS Implementation]

Pages

Open source RTOS for small embedded systems

Homepage

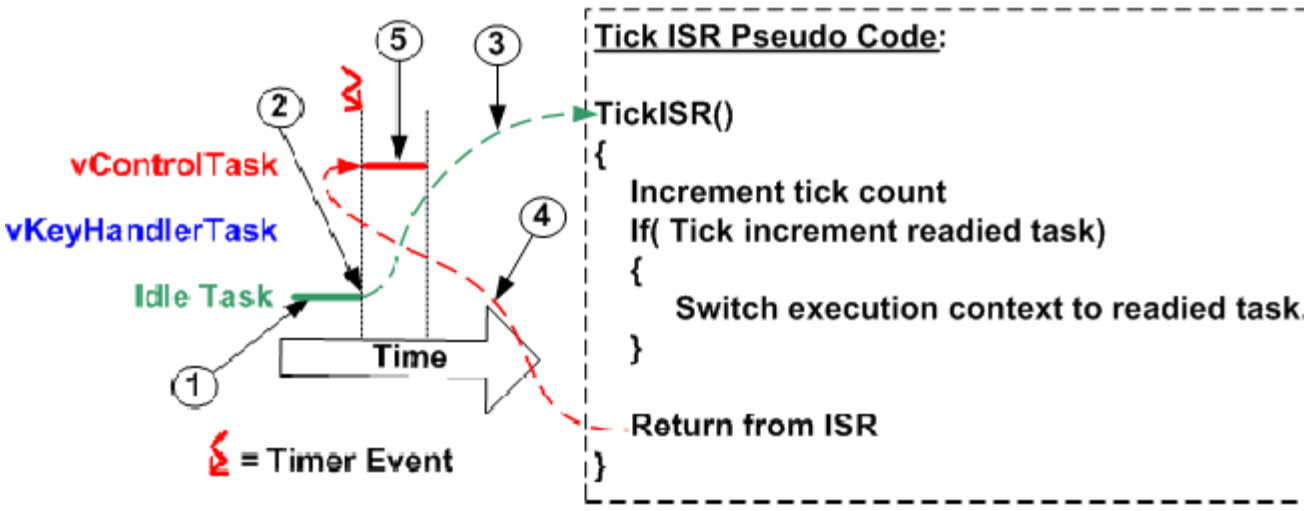
The RTOS Tick

[RTOS Implementation Building Blocks]

When sleeping, a task will specify a time after which it requires 'waking'. When blocking, a task can specify a maximum time it wishes to wait.

The FreeRTOS real time kernel measures time using a **tick** count variable. A timer interrupt (the RTOS **tick interrupt**) increments the tick count with strict temporal accuracy - allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency.

Each time the tick count is incremented the real time kernel must check to see if it is now time to unblock or wake a task. It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR should return to the newly woken/unblocked task - effectively interrupting one task but returning to another. This is depicted below:



Referring to the numbers in the diagram above:

- At (1) the RTOS idle task is executing.
- At (2) the RTOS tick occurs, and control transfers to the tick ISR (3).
- The RTOS tick ISR makes vControlTask ready to run, and as vControlTask has a higher priority than the RTOS idle task, switches the context to that of vControlTask.
- As the execution context is now that of vControlTask, exiting the ISR (4) returns control to vControlTask, which starts executing (5).

A context switch occurring in this way is said to be **Preemptive**, as the interrupted task is preempted without suspending itself voluntarily.

The AVR port of FreeRTOS uses a compare match event on timer 1 to generate the RTOS tick. The following pages describe how the RTOS tick ISR is implemented using the WinAVR development tools.

Next: RTOS Implementation - The GCC Signal Attribute

Open source RTOS for small embedded systems

Homepage

FreeRTOS Tick Code

[RTOS Implementation Building Blocks]

The actual source code used by the FreeRTOS AVR port is slightly different to the examples shown on the previous pages. vPortYieldFromTick() is itself implemented as a 'naked' function, and the context is saved and restored within vPortYieldFromTick(). It is done this way due to the implementation of non-preemptive context switches (where a task blocks itself) - which are not described here.

The FreeRTOS implementation of the RTOS tick is therefore *(see the comments in the source code snippets for further details)*:

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal, naked ) );
void vPortYieldFromTick( void ) __attribute__ ( ( naked ) );

/*-----------------------------------------------------*/

/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A( void )
{
    /* Call the tick function. */
    vPortYieldFromTick();

    /* Return from the interrupt.  If a context
    switch has occurred this will return to a
    different task. */
    asm volatile ( "reti" );
}
/*-----------------------------------------------------*/

void vPortYieldFromTick( void )
{
    /* This is a naked function so the context
    is saved. */
    portSAVE_CONTEXT();

    /* Increment the tick count and check to see
    if the new tick value has caused a delay
    period to expire.  This function call can
    cause a task to become ready to run. */
    vTaskIncrementTick();

    /* See if a context switch is required.
    Switch to the context of a task made ready
    to run by vTaskIncrementTick() if it has a
    priority higher than the interrupted task. */
    vTaskSwitchContext();

    /* Restore the context.  If a context switch
    has occurred this will restore the context of
    the task being resumed. */
    portRESTORE_CONTEXT();

    /* Return from this naked function. */
    asm volatile ( "ret" );
}
/*-----------------------------------------------------*/
```

Next: RTOS Implementation - The AVR Context

Open source RTOS for small embedded systems

Homepage

Restoring the Context

[RTOS Implementation Building Blocks]

portRESTORE_CONTEXT() is the reverse of portSAVE_CONTEXT(). The context of the task being resumed was previously stored in the tasks stack. The real time kernel retrieves the stack pointer for the task then POP's the context back into the correct processor registers.

```
#define portRESTORE_CONTEXT()        \
asm volatile (
  "lds  r26, pxCurrentTCB      \n\t" \ (1)
  "lds  r27, pxCurrentTCB + 1  \n\t" \ (2)
  "ld   r28, x+                \n\t" \
  "out  __SP_L__, r28          \n\t" \ (3)
  "ld   r29, x+                \n\t" \
  "out  __SP_H__, r29          \n\t" \ (4)
  "pop  r31                    \n\t" \
  "pop  r30                    \n\t" \

    :
    :
    :

  "pop  r1                     \n\t" \
  "pop  r0                     \n\t" \ (5)
  "out  __SREG__, r0           \n\t" \ (6)
  "pop  r0                     \n\t" \ (7)
);
```

Referring to the code above:

- pxCurrentTCB holds the address from where the tasks stack pointer can be retrieved. This is loaded into the X register (1 and 2).
- The stack pointer for the task being resumed is loaded into the AVR stack pointer, first the low byte (3), then the high nibble (4).
- The processor registers are then popped from the stack in reverse numerical order, down to R1.
- The status register stored on the stack between registers R1 and R0, so is restored (6) before R0 (7).

Next: RTOS Implementation - Putting It All Together

Open source RTOS for small embedded systems

Homepage

Saving the Context

[RTOS Implementation Building Blocks]

Each real time task has it's own stack memory area so the context can be saved by simply pushing processor registers onto the task stack. Saving the AVR context is one place where assembly code is unavoidable.

portSAVE_CONTEXT() is implemented as a macro, the source code for which is given below:

```
#define portSAVE_CONTEXT()           \
asm volatile (                         \
  "push  r0                    \n\t" \ (1)
  "in    r0, __SREG__          \n\t" \ (2)
  "cli                         \n\t" \ (3)
  "push  r0                    \n\t" \ (4)
  "push  r1                    \n\t" \ (5)
  "clr   r1                    \n\t" \ (6)
  "push  r2                    \n\t" \ (7)
  "push  r3                    \n\t" \
  "push  r4                    \n\t" \
  "push  r5                    \n\t" \

    :
    :
    :

  "push  r30                   \n\t" \
  "push  r31                   \n\t" \
  "lds   r26, pxCurrentTCB     \n\t" \ (8)
  "lds   r27, pxCurrentTCB + 1 \n\t" \ (9)
  "in    r0, __SP_L__          \n\t" \ (10)
  "st    x+, r0                \n\t" \ (11)
  "in    r0, __SP_H__          \n\t" \ (12)
  "st    x+, r0                \n\t" \ (13)
);
```

Referring to the source code above:

- Processor register R0 is saved first as it is used when the status register is saved, and must be saved with its original value.
- The status register is moved into R0 (2) so it can be saved onto the stack (4).
- Processor interrupts are disabled (3). If portSAVE_CONTEXT() was only called from within an ISR there would be no need to explicitly disable interrupts as the AVR will have already done so. As the portSAVE_CONTEXT() macro is also used outside of interrupt service routines (when a task suspends itself) interrupts must be explicitly cleared as early as possible.
- The code generated by the compiler from the ISR C source code assumes R1 is set to zero. The original value of R1 is saved (5) before R1 is cleared (6).
- Between (7) and (8) all remaining processor registers are saved in numerical order.
- The stack of the task being suspended now contains a copy of the tasks execution context. The kernel stores the tasks stack pointer so the context can be retrieved and restored when the task is resumed. The X processor register is loaded with the address to which the stack pointer is to be saved (8 and 9).
- The stack pointer is saved, first the low byte (10 and 11), then the high nibble (12 and 13).

Next: RTOS Implementation - Restoring The Context

Open source RTOS for small embedded systems

Detailed Example

[2.RTOS Implementation]

Pages

Detailed Description

The final part of section 2 shows how these building blocks and source code modules are used to achieve a context switch on the AVR microcontroller. The example demonstrates in seven steps the process of switching from a lower priority task, called TaskA, to a higher priority task, called TaskB.

The source code is compatible with the WinAVR development tools.

Open source RTOS for small embedded systems

Homepage

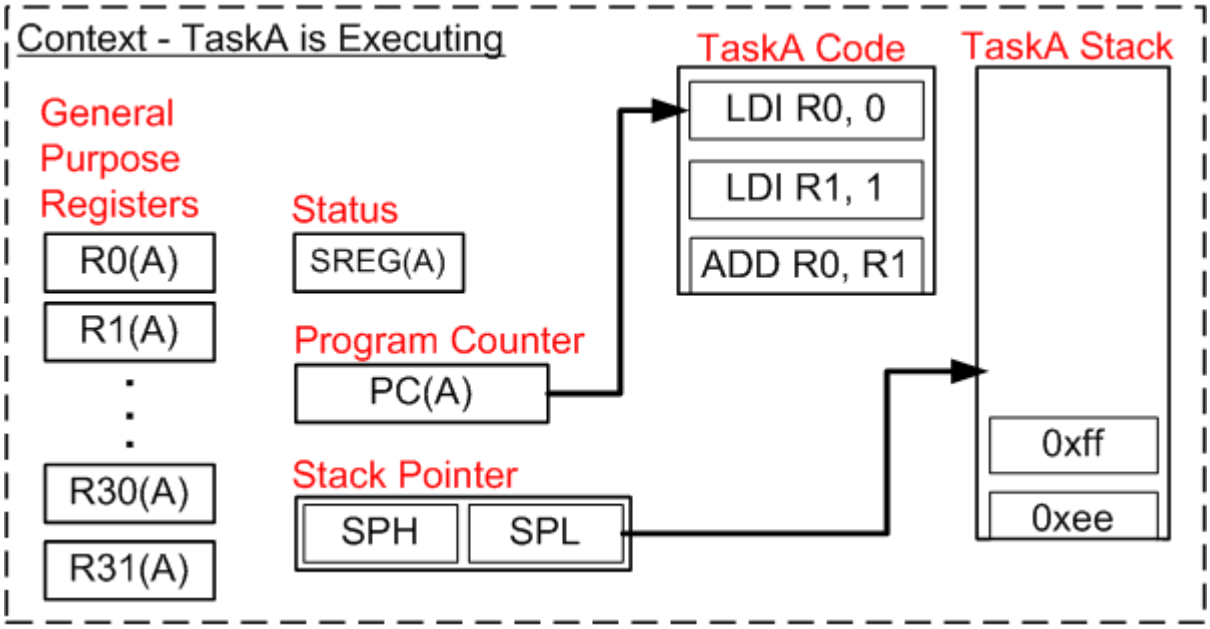
RTOS Context Switch - Step 1

[Detailed Example]

Prior to the RTOS tick interrupt

This example starts with TaskA executing. TaskB has previously been suspended so its context has already been stored on the TaskB stack.

TaskA has the context demonstrated by the diagram below.



The (A) label within each register shows that the register contains the correct value for the context of task A.

Next: RTOS Implementation - Detailed Example Step 2

Open source RTOS for small embedded systems

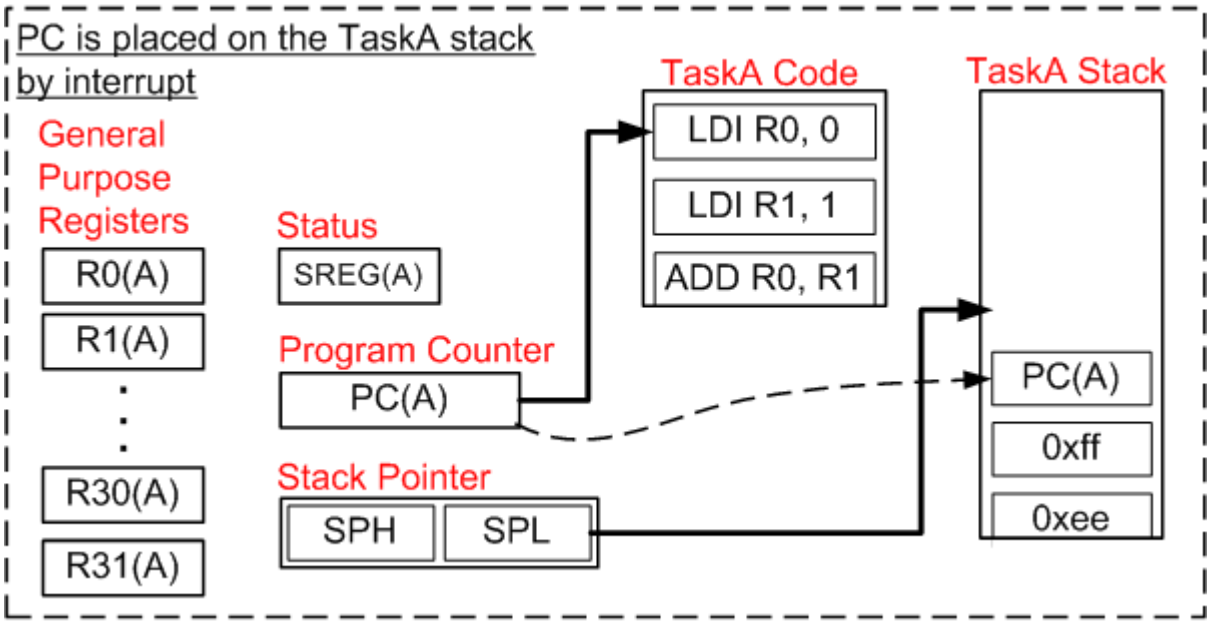
Homepage

RTOS Context Switch - Step 2

[Detailed Example]

The RTOS tick interrupt occurs

The RTOS tick occurs just as TaskA is about to execute an LDI instruction. When the interrupt occurs the AVR microcontroller automatically places the current program counter (PC) onto the stack before jumping to the start of the RTOS tick ISR.



Next: RTOS Implementation - Detailed Example Step 3

Open source RTOS for small embedded systems

Homepage

RTOS Context Switch - Step 3

[Detailed Example]

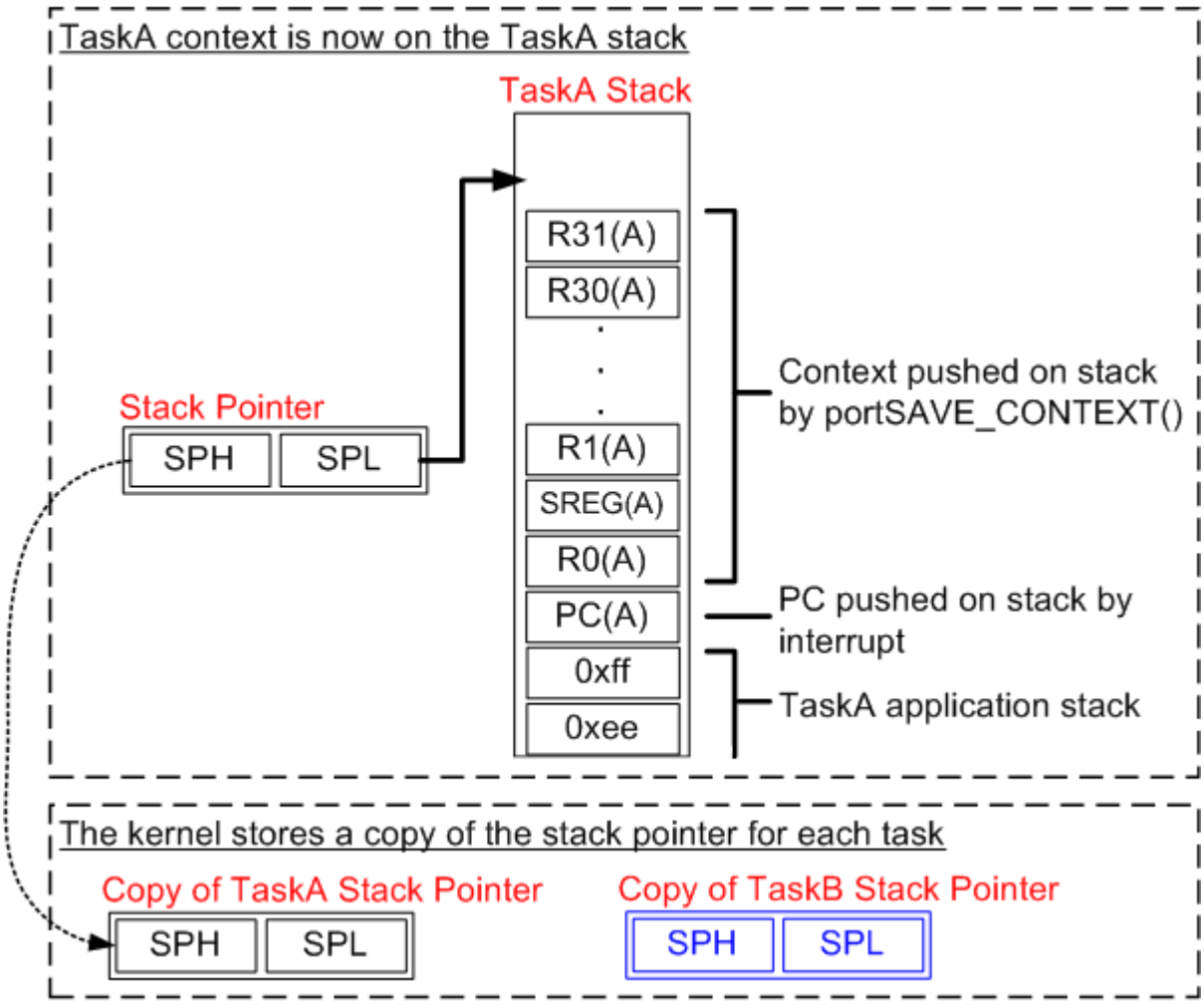
The RTOS tick interrupt executes

The ISR source code is given below. The comments have been removed to ease reading, but can be viewed on a previous page.

```
/* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A( void )
{
    vPortYieldFromTick();
    asm volatile ( "reti" );
}
/*-----------------------------------------------------*/

void vPortYieldFromTick( void )
{
    portSAVE_CONTEXT();

    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();

    asm volatile ( "ret" );
}
/*-----------------------------------------------------*/
```

SIG_OUTPUT_COMPARE1A() is a naked function, so the first instruction is a call to vPortYieldFromTick(). vPortYieldFromTick() is also a naked function so the AVR execution context is saved explicitly by a call to portSAVE_CONTEXT().

portSAVE_CONTEXT() pushes the entire AVR execution context onto the stack of TaskA, resulting in the stack illustrated below. The stack pointer for TaskA now points to the top of it's own context. portSAVE_CONTEXT() completes by storing a copy of the stack pointer. The real time kernel already has copy of the TaskB stack pointer - taken the last time TaskB was suspended.



Next: RTOS Implementation - Detailed Example Step 4

Open source RTOS for small embedded systems

Homepage

RTOS Context Switch - Step 4

[Detailed Example]

Incrementing the Tick Count

vTaskIncrementTick() executes after the TaskA context has been saved. For the purposes of this example assume that incrementing the tick count has caused TaskB to become ready to run. TaskB has a higher priority than TaskA so vTaskSwitchContext() selects TaskB as the task to be given processing time when the ISR completes.

Next: RTOS Implementation - Detailed Example Step 5

Open source RTOS for small embedded systems

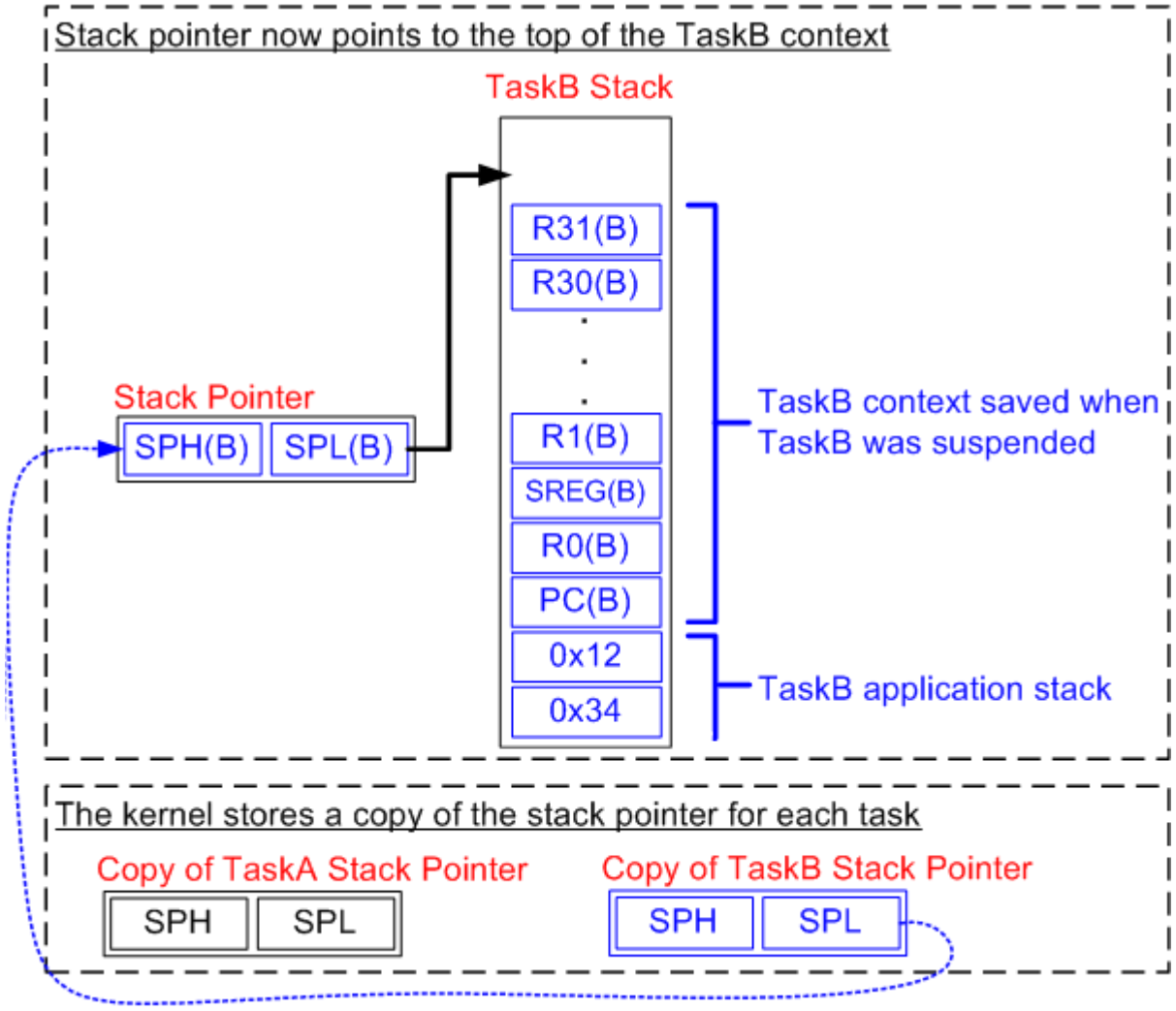
Homepage

RTOS Context Switch - Step 5

[Detailed Example]

The TaskB stack pointer is retrieved

The TaskB context must be restored. The first thing portRESTORE_CONTEXT does is retrieve the TaskB stack pointer from the copy taken when TaskB was suspended. The TaskB stack pointer is loaded into the processor stack pointer, so now the AVR stack points to the top of the TaskB context.



Next: RTOS Implementation - Detailed Example Step 6

Open source RTOS for small embedded systems

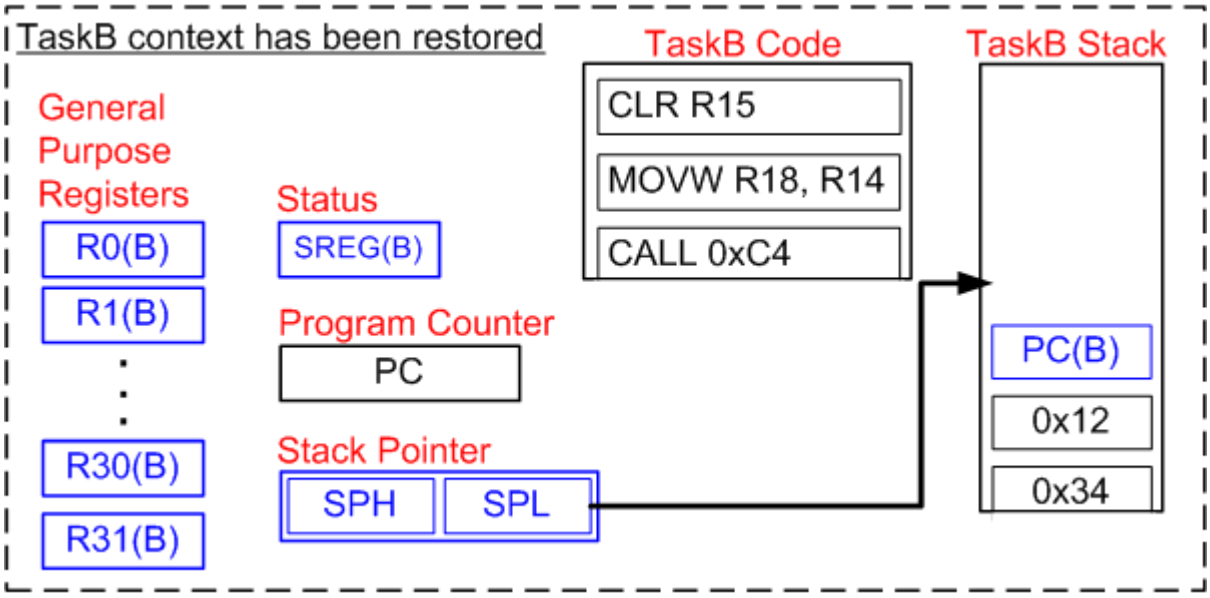
Homepage

RTOS Context Switch - Step 6

[Detailed Example]

Restore the TaskB context

portRESTORE_CONTEXT() completes by restoring the TaskB context from its stack into the appropriate processor registers.



Only the program counter remains on the stack.

Next: RTOS Implementation - Detailed Example Step 7

Open source RTOS for small embedded systems

Homepage

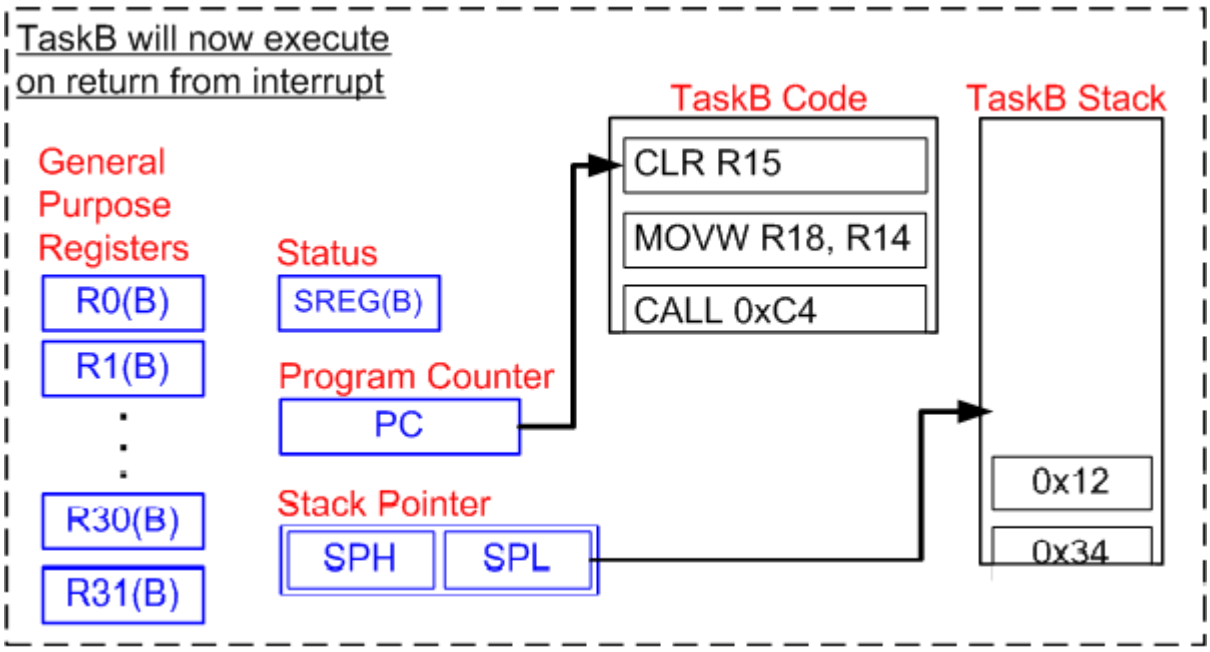
RTOS Context Switch - Step 7

[Detailed Example]

The RTOS tick exits

vPortYieldFromTick() returns to SIG_OUTPUT_COMPARE1A() where the final instruction is a return from interrupt (RETI). A RETI instruction assumes the next value on the stack is a return address placed onto the stack when the interrupt occurred.

When the RTOS tick interrupt started the AVR automatically placed the TaskA return address onto the stack - the address of the next instruction to execute in **TaskA**. The ISR altered the stack pointer so it now points to the **TaskB** stack. Therefore the return address POP'ed from the stack by the RETI instruction is actually the address of the instruction **TaskB** was going to execute immediately before it was suspended.



The RTOS tick interrupt interrupted **TaskA**, but is returning to **TaskB** - the context switch is complete!

If you would like more information, take a look at the FreeRTOS ColdFire Implementation Report. This was written by the Motorola ColdFire port authors, and details both the ColdFire source code and the development process undertaken in producing the port.

Homepage

Real Time Application Design
Using FreeRTOS in small embedded systems

<<< | >>>

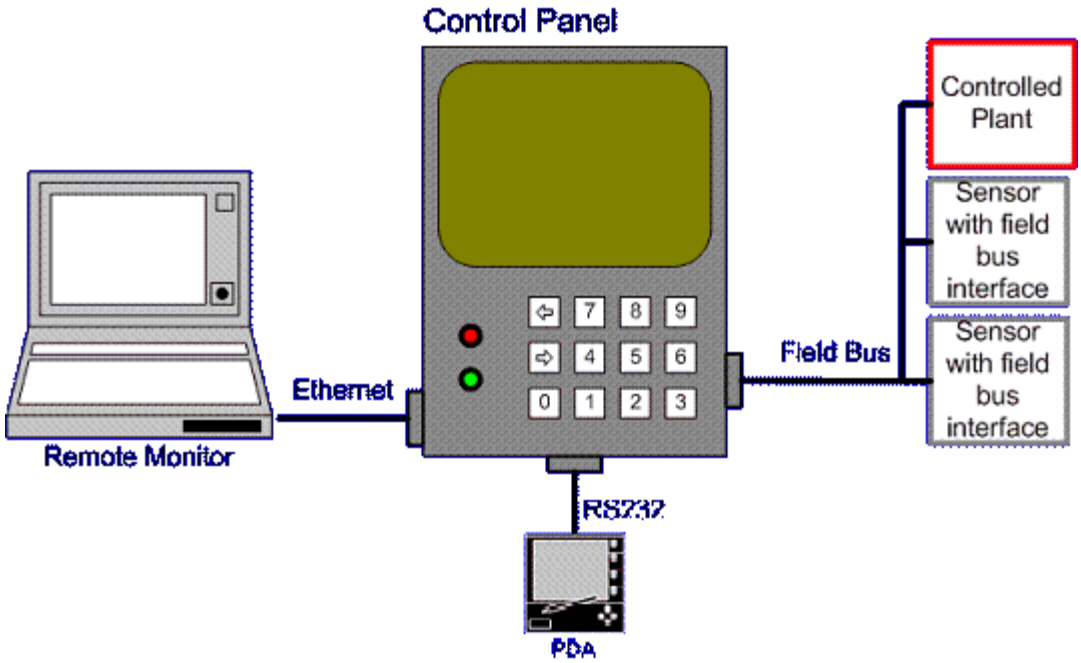
HINT: Use the <<< and >>> arrows to navigate this section.

Introduction

This section presents four contrasting design solutions to a hypothetical embedded real time application. The suitability of each solution is judged for embedded computers with varying RAM, ROM and processing capabilities. In addition the simplicity and corresponding maintainability of each design is assessed.

This is not intended to present an exhaustive list of possible designs, but a guide to the ways in which the FreeRTOS.org[tm] real time kernel can be used.

The [hypothetical] Application



System context [not to scale].

The application will execute on an embedded single board computer that must control a plant while maintaining both local and remote user interfaces.

Depicted above, the system consists of:

1. An embedded computer within a control terminal.
2. Two fieldbus networked sensors.
3. The plant being controlled (could be anything, motor, heater, etc.). This is connected on the same fieldbus network.
4. A matrix keypad that is scanned using general purpose IO.
5. Two LED indicators.
6. An LCD display.
7. An embedded WEB server to which a remote monitoring computer can attach.
8. An RS232 interface to a configuration utility that runs on a PDA.

Top Level Software Requirements

Here we are interested in the sequencing and timing requirements, rather than the exact functional requirements.

Plant Control

Each control cycle shall perform the following sequence:

1. Transmit a frame on the fieldbus to request data from the networked sensors.
2. Wait to receive data from both sensors.
3. Execute the control algorithm.
4. Transmit a command to the plant.

The control function of the embedded computer shall transmit a request every 10ms exactly, and the resultant command shall be transmitted within 5ms of this request. The control algorithm is reliant on accurate timing, it is therefore paramount that these timing requirements are met.

Local Operator Interface **[keypad and LCD]**

The keypad and LCD can be used by the operator to select, view and modify system data. The operator interface shall function while the plant is being controlled.

To ensure no key presses are missed the keypad shall be scanned at least every 15ms. The LCD shall update within 50ms of a key being pressed.

LED

The LED shall be used to indicate the system status. A flashing green LED shall indicate that the system is running as expected. A flashing red LED shall indicate a fault condition.

The correct LED shall flash on and off once ever second. This flash rate shall be maintained to within 50ms.

RS232 PDA Interface

The PDA RS232 interface shall be capable of viewing and accessing the same data as the local operator interface, and the same timing constraints apply - discounting any data transmission times.

TCP/IP Interface

The embedded WEB server shall service HTTP requests within one second.

Application components

The timing requirements of the hypothetical system can be split into three categories:

1. **Strict timing** - the plant control

   The control function has a very strict timing requirement as it must execute every 10ms.

2. **Flexible timing** - the LED

   While the LED outputs have both maximum and minimum time constraints, there is a large timing band within which they can function.

3. **Deadline only timing** - the human interfaces

   This includes the keypad, LCD, RS232 and TCP/IP Ethernet communications.

   The human interface functions have a different type of timing requirement as only a maximum limit is specified. For example, the keypad must be scanned at least every 10ms, but any rate up to 10ms is acceptable.

NEXT >>> Solution #1: Why use an RTOS kernel?

Homepage

Solution #1
Why Use an RTOS Kernel?

<<< | >>>

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.

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 LED's 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

Homepage

Solution #2
A Fully Preemptive System

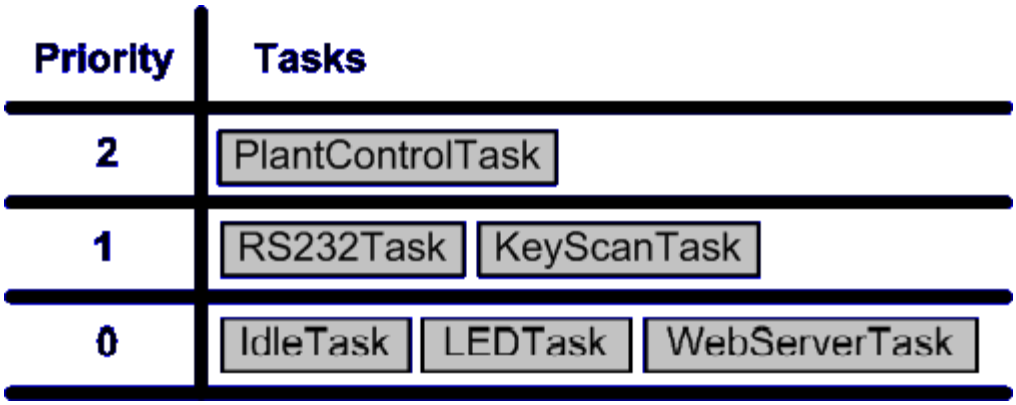
<<< | >>>

Synopsis

This is a traditional preemptive multitasking solution. It makes full use of the RTOS services with no regard to the resultant memory and processor overhead. There is a simplistic partitioning of the required functionality to a number of autonomous tasks.

Implementation

A separate task is created for each part of the system that can be identified as being able to exist in isolation, or as having a particular timing requirement.



Solution #2 functions tasks and priorities

Tasks will block until an event indicates that processing is required. Events can either be external (such as a key being pressed), or internal (such as a timer expiring).

Priorities are allocated to tasks in accordance to their timing requirements. The stricter the timing requirement the higher the priority.

Concept of Operation

The highest priority task that is able to execute (is not blocked) is the task guaranteed by the RTOS to get processor time. The kernel will immediately suspend an executing task should a higher priority task become available.

This scheduling occurs automatically, with no explicit knowledge, structuring or commands within the application source code. It is however the responsibility of the application designers to ensure that tasks are allocated an appropriate priority.

When no task is able to execute the idle task will execute. The idle task has the option of placing the processor into power save mode.

Scheduler Configuration

The scheduler is configured for preemptive operation. The kernel tick frequency should be set at the slowest value that provides the required time granularity.

Evaluation


Simple, segmented, flexible, maintainable design with few interdependencies.


Processor utilisation is automatically switched from task to task on a most urgent need basis with no explicit action required within the application source code.


Power consumption can be reduced if the idle task places the processor into power save (sleep) mode, but may also be wasted as the tick interrupt will sometimes wake the processor unnecessarily.


The kernel functionality will use processing resources. The extent of this will depend on the chosen kernel tick frequency.


This solution requires a lot of tasks, each of which require their own stack, and many of which require a queue on which events can be received. This solution therefore uses a lot of RAM.


Frequent context switching between tasks of the same priority will waste processor cycles.

Conclusion

This can be a good solution provided the RAM and processing capacity is available. The partitioning of the application into tasks and the priority assigned to each task requires careful consideration.

Example

This example is a partial implementation of the hypothetical application introduced previously. The FreeRTOS.org API is used.

Plant Control Task

This task implements all the control functionality. It has critical timing requirements and is therefore given the highest priority within the system:

```
#define CYCLE_RATE_MS       10
#define MAX_COMMS_DELAY     2

void PlantControlTask( void *pvParameters )
{
portTickType xLastWakeTime;
DataType Data1, Data2;

    InitialiseTheQueue();

    // A
    xLastWakeTime = xTaskGetTickCount();

    // B
    for( ;; )
    {
```

```
        // C
        vTaskDelayUntil( &xLastWakeTime, CYCLE_RATE_MS );

        // Request data from the sensors.
        TransmitRequest();

        // D
        if( xQueueReceive( xFieldBusQueue, &Data1, MAX_COMMS_DELAY ) )
        {
            // E
            if( xQueueReceive( xFieldBusQueue, &Data2, MAX_COMMS_DELAY ) )
            {
                PerformControlAlgorithm();
                TransmitResults();
            }
        }
    }

    // Will never get here!
}
```

Referring to the labels within the code fragment above:

A. `xLastWakeTime` is initialised. This variable is used with the `vTaskDelayUntil()` API function to control the frequency at which the control function executes.

B. This function executes as an autonomous task so must never exit.

C. `vTaskDelayUntil()` tells the kernel that this task should start executing exactly 10ms after the time stored in `xLastWakeTime`. Until this time is reached the control task will block. As this is the highest priority task within the system it is guaranteed to start executing again at exactly the correct time. It will pre-empt any lower priority task that happens to be running.

D. There is a finite time between data being requested from the networked sensors and that data being received. Data arriving on the field bus is placed in the `xFieldBusQueue` by an interrupt service routine, the control task can therefore make a blocking call on the queue to wait for data to be available. As before, because it is the highest priority task in the system it is guaranteed to continue executing immediately data is available.

E. As 'D', waiting for data from the second sensor.

A return value of 0 from xQueueReceive() indicates that no data arrived within the specified block period. This is an error condition the task must handle. This and other error handling functionality has been omitted for simplicity.

Embedded WEB Server Task

The embedded WEB server task can be represented by the following pseudo code. This only utilises processor time when data is available but will take a variable and relatively long time to complete. It is therefore given a low priority to prevent it adversely effecting the timing of the plant control, RS232 or keypad scanning tasks.

```
void WebServerTask( void *pvParameters )
{
DataTypeA Data;

    for( ;; )
    {
        // Block until data arrives.  xEthernetQueue is filled by the
        // Ethernet interrupt service routine.
        if( xQueueReceive( xEthernetQueue, &Data, MAX_DELAY ) )
        {
            ProcessHTTPData( Data );
        }
```

```
    }
}
```

RS232 Interface

This is very similar in structure to the embedded WEB server task. It is given a medium priority to ensure it does not adversely effect the timing of the plant control task.

```
void RS232Task( void *pvParameters )
{
DataTypeB Data;

    for( ;; )
    {
        // Block until data arrives.  xRS232Queue is filled by the
        // RS232 interrupt service routine.
        if( xQueueReceive( xRS232Queue, &Data, MAX_DELAY ) )
        {
            ProcessSerialCharacters( Data );
        }
    }
}
```

Keypad Scanning Task

This is a simple cyclical task. It is given a medium priority as it's timing requirements are similar to the RS232 task.

The cycle time is set much faster than the specified limit. This is to account for the fact that it may not get processor time immediately upon request - and once executing may get pre-empted by the plant control task.

```
#define DELAY_PERIOD 4

void KeyScanTask( void *pvParmeters )
{
char Key;
portTickType xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, DELAY_PERIOD );

        // Scan the keyboard.
        if( KeyPressed( &Key ) )
        {
            UpdateDisplay( Key );
        }
    }
}
```

If the overall system timing were such that this could be made the lowest priority task then the call to vTaskDelayUntil() could be removed altogether. The key scan function would then execute continuously whenever all the higher priority tasks were blocked - effectively taking the place of the idle task.

LED Task

This is the simplest of all the tasks.

```
#define DELAY_PERIOD 1000
```

```
void LEDTask( void *pvParmeters )
{
portTickType xLastWakeTime;

    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, DELAY_PERIOD );

        // Flash the appropriate LED.
        if( SystemIsHealthy() )
        {
            FlashLED( GREEN );
        }
        else
        {
            FlashLED( RED );
        }
    }
}
```

NEXT >>> Solution #3: Reducing RAM utilisation

Homepage

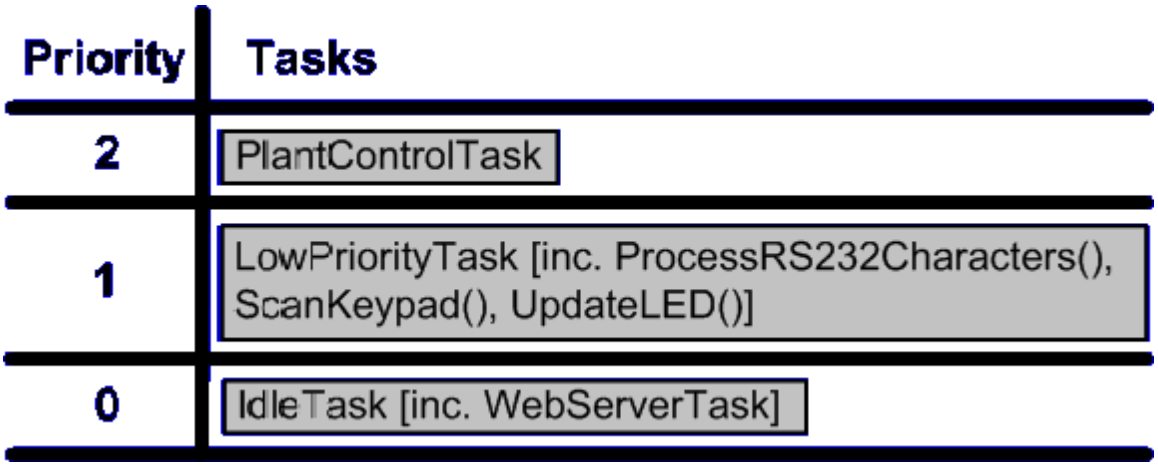
Solution #3
Reducing RAM Utilisation

<<< | >>>

Synopsis

Solution #2 makes full use of the RTOS. This results in a clean design, but one that can only be used on embedded computers with ample RAM and processing resource. Solution #3 attempts to reduce the RAM usage by changing the partitioning of functionality into tasks.

Implementation



Solution #3 functions tasks and priorities

We have previously seen how the timing requirements of our hypothetical application can be split into three categories:

1. **Strict timing** - the plant control

   As before, a high priority task is created to service the critical control functionality.

2. **Deadline only timing** - the human interfaces

   Solution #3 groups the RS232, keyscan and LED functionality into a single medium priority task.

   For reasons previously stated it is desirable for the embedded WEB server task to operate at a lower priority. Rather than creating a task specifically for the WEB server an idle task hook is implemented to add the WEB server functionality to the idle task. The WEB server must be written to ensure it never blocks!

3. **Flexible timing** - the LED

   The LED functionality is too simple to warrant it's own task if RAM is at a premium. For reasons of demonstration this example includes the LED functionality in the single medium priority task. It could of coarse be implemented in a number of ways (from a peripheral timer for example).

Tasks other than the idle task will block until an event indicates that processing is required. Events can either be external (such as a key being pressed), or internal (such as a timer expiring).

Concept of Operation

The grouping of functionality into the medium priority task has three important advantages over the infinite loop implementation presented in solution #1:

1. The use of a queue allows the medium priority task to block until an event causes data to be available - and then immediately jump to the relevant function to handle the event. This prevents wasted processor cycles - in contrast to the infinite loop implementation whereby an event will only be processed once the loop cycles to the appropriate handler.

2. The use of the real time kernel removes the requirement to explicitly consider the scheduling of the time critical task within the application source code.

3. The removal of the embedded WEB server function from the loop has made the execution time more predictable.

In addition, the functionality that has been grouped into a single task is taken from several tasks that previously shared the same priority anyway (barr the LED function). The frequency at which code at this priority executes will not alter whether in a single or multiple tasks.

The plant control task, as the highest priority task, is guaranteed to be allocated processing time whenever it requires. It will pre-empt the low and medium priority tasks if necessary. The idle task will execute whenever both application tasks are blocked. The idle task has the option of placing the processor into power save mode.

Scheduler Configuration

The scheduler is configured for preemptive operation. The kernel tick frequency should be set at the slowest value that provides the required time granularity.

Evaluation


Creates only two application tasks so therefore uses much less RAM than solution #2.


Processor utilisation is automatically switched from task to task on a most urgent need basis.


Utilising the idle task effectively creates three application task priorities with the overhead of only two.


The design is still simple but the execution time of the functions within the medium priority task could introduce timing issues. The separation of the embedded WEB server task reduces this risk and in any case any such issues would not effect the plant control task.


Power consumption can be reduced if the idle task places the CPU into power save (sleep) mode, but may also be wasted as the tick interrupt will sometimes wake the CPU unnecessarily.


The RTOS functionality will use processing resources. The extent of this will depend on the chosen kernel tick frequency.


The design might not scale if the application grows too large.

Conclusion

This can be a good solution for systems with limited RAM but it is still processor intensive. Spare capacity within the system should be checked to allow for future expansion.

Example

This example is a partial implementation of the hypothetical application introduced previously. The FreeRTOS.org API is used.

Plant Control Task

The plant control task is identical to that described in solution #2.

The Embedded WEB Server

This is simply a function that is called from the idle task and runs to completion.

The medium Priority Task

The medium priority task can be represented by the following pseudo code.

```
#define DELAY_PERIOD 4
#define FLASH_RATE 1000

void MediumPriorityTask( void *pvParameters )
{
xQueueItem Data;
portTickType FlashTime;

    InitialiseQueue();
    FlashTime = xTaskGetTickCount();

    for( ;; )
    {
        do
        {
            // A
            if( xQueueReceive( xCommsQueue, &Data, DELAY_PERIOD ) )
            {
                ProcessRS232Characters( Data.Value );
            }

          // B
        } while ( uxQueueMessagesWaiting( xCommsQueue ) );

        // C
        if( ScanKeypad() )
        {
            UpdateLCD();
        }

        // D
        if( ( xTaskGetTickCount() - FlashTime ) >= FLASH_RATE )
        {
            FlashTime = xTaskGetTickCount();
            UpdateLED();
        }
    }
```

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

Referring to the labels within the code fragment above:

A. The task first blocks waiting for a communications event. The block time is relatively short.

B. The do-while loop executes until no data remains in the queue. This implementation would have to be modified if data arrives too quickly for the queue to ever be completely empty.

C. Either the queue has been emptied of all data, or no data arrived within the specified blocking period. The maximum time that can be spent blocked waiting for data is short enough to ensure the keypad is scanned frequently enough to meet the specified timing constraints.

D. Check to see if it is time to flash the LED. There will be some jitter in the frequency at which this line executes, but the LED timing requirements are flexible enough to be met by this implementation.

NEXT >>> Solution #4: Reducing the processor overhead

Homepage

Solution #4
Reducing the Processor Overhead

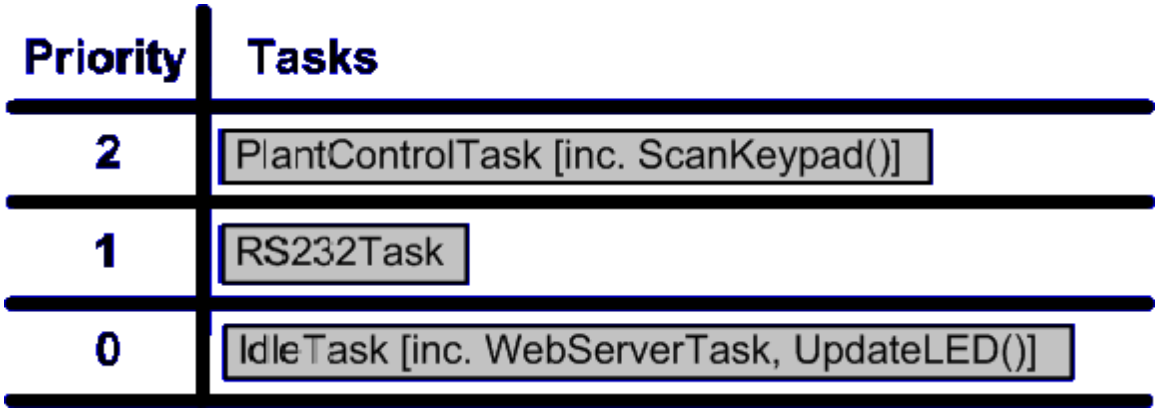
<<< | >>>

Synopsis

Solution #2 showed how a clean application can be produced by fully utilising the RTOS functionality. Solution #3 showed how this can be adapted for embedded computers with limited RAM resource. Solution #4 makes further modifications with the objective of a reduction in the RTOS processing overhead.

A hybrid scheduling algorithm (neither fully preemptive or fully cooperative) is created by configuring the kernel for cooperative scheduling, then performing context switching from within event interrupt service routines.

Implementation



Solution #4 functions tasks and priorities

The critical plant control functionality is once again implemented by a high priority task but the use of the cooperative scheduler necessitates a change to its implementation. Previously the timing was maintained using the `vTaskDelayUntil()` API function. When the preemptive scheduler was used, assigning the control task the highest priority ensured it started executing at exactly the specified time. Now the cooperative scheduler is being used - therefore a task switch will only occur when explicitly requested from the application source code so the guaranteed timing is lost.

Solution #4 uses an interrupt from a peripheral timer to ensure a context switch is requested at the exact frequency required by the control task. The scheduler ensures that each requested context switch results in a switch to the highest priority task that is able to run.

The keypad scanning function also requires regular processor time so it too is executed within the task triggered by the timer interrupt. The timing of this task can be easily evaluated; The worst case processing time of the control function is given by the error case - when no data is forthcoming from the networked sensors causing the control function to time out. The execution time of the keypad scanning function is basically fixed. We can therefore be certain that chaining their functionality in this manner will never result in jitter in the control cycle frequency - or worse still a missed control cycle.

The RS232 task will be scheduled by the RS232 interrupt service routine.

The flexible timing requirements of the LED functionality means it can probably join the embedded WEB server task within the idle task hook. If this is not adequate then it too can be moved up to the high priority task.

Concept of Operation

The cooperative scheduler will only perform a context switch when one is explicitly requested. This greatly reduces the processor overhead imposed by the RTOS. The idle task, including the embedded WEB server functionality, will execute without any unnecessary interruptions from the kernel.

An interrupt originating from either the RS232 or timer peripheral will result in a context switch exactly and only when one is necessary. This way the RS232 task will still pre-empt the idle task, and can still itself be pre-empted by the plant control task - maintaining the prioritised system functionality.

Scheduler Configuration

The scheduler is configured for cooperative operation. The kernel tick is used to maintain the real time tick value only.

Evaluation


Creates only two application tasks so therefore uses much less RAM than solution #2.


The RTOS processing overhead is reduced to a minimum.


Only a subset of the RTOS features are used. This necessitates a greater consideration of the timing and execution environment at the application source code level, but still allows for a greatly simplified design (when compared to solution #1).


Reliance on processor peripherals. Non portable.


The problems of analysis and interdependencies between modules as were identified with solution #1 are starting to become a consideration again - although to a much lesser extent.


The design might not scale if the application grows too large

Conclusion

Features of the RTOS kernel can be used with very little overhead, enabling a simplified design even on systems where processor and memory constraints prevent a fully preemptive solution.

Example

This example is a partial implementation of the hypothetical application introduced previously. The FreeRTOS.org API is used.

High Priority Task

The high priority task is triggered by a semaphore 'given' by a periodic interrupt service routine:

```
void vTimerInterrupt( void )
{
    // 'Give' the semaphore.  This will wake the high priority task.
    xSemaphoreGiveFromISR( xTimingSemaphore );

    // The high priority task will now be able to execute but as
    // the cooperative scheduler is being used it will not start
    // to execute until we explicitly cause a context switch.
    taskYIELD();
}
```

Note that the syntax used to force a context switch from within an ISR is different for different ports. Do not copy this example directly but instead check the documentation for the port you are using.

The high priority task contains both the plant control and keypad functionality. `PlantControlCycle()` is called first to ensure consistency in its timing.

```
void HighPriorityTaskTask( void *pvParameters )
{
    // Start by obtaining the semaphore.
    xSemaphoreTake( xSemaphore, DONT_BLOCK );

    for( ;; )
    {
        // Another call to take the semaphore will now fail until
        // the timer interrupt has called xSemaphoreGiveFromISR().
        // We use a very long block time as the timing is controlled
        // by the frequency of the timer.
        if( xSemaphoreTake( xSemaphore, VERY_LONG_TIME ) == pdTRUE )
        {
            // We unblocked because the semaphore became available.
            // It must be time to execute the control algorithm.
            PlantControlCycle();

            // Followed by the keyscan.
            if( KeyPressed( &Key ) )
            {
                UpdateDisplay( Key );
            }
        }

        // Now we go back and block again until the next timer interrupt.
    }
}
```

RS232 Task

The RS232 task simply blocks on a queue waiting for data to arrive. The RS232 interrupt service routine must post the data onto the queue - making the task ready to run - then force a context switch. This mechanism is as per the timer interrupt pseudo code given above.

The RS232 task can therefore be represented by the following pseudo code:

```
void vRS232Task( void *pvParameters )
{
DataType Data;

    for( ;; )
    {
       if( cQueueReceive( xRS232Queue, &Data, MAX_DELAY ) )
        {
            ProcessRS232Data( Data );
        }
    }
```

```
}
```

The Embedded WEB Server and LED Functionality

The remaining system functionality is placed within the idle task hook. This is simply a function that is called by each cycle of the idle task.

```
void IdleTaskHook( void )
{
static portTickType LastFlashTime = 0;

    ProcessHTTPRequests();

    // Check the tick count value to see if it is time to flash the LED
    // again.
    if( ( xTaskGetTickCount() - LastFlashTime ) > FLASH_RATE )
    {
        UpdateLED();

        // Remember the time now so we know when the next flash is due.
        LastFlashTime = xTaskGetTickCount();
    }
}
```
