Lecture 21: Real-Time Operating Systems

Seyed-Hosein Attarzadeh-Niaki

Some slides due to Ingo Sander and Edward Lee

Review

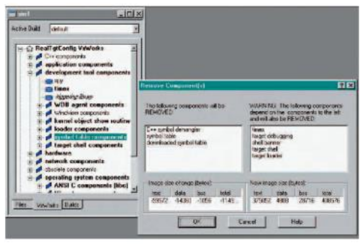
- Mutual exclusion
  - Priority inversion
    - Priority inheritance protocol
  - Deadlock
    - Priority ceiling protocol
- Aperiodic scheduling
  - Polling server
  - Sporadic server
- Multiprocessor scheduling
  - Brittleness
  - Richard's anomalies

Outline

- (Embedded) Real-time Oss
  - Characteristics
  - Microkernels
  - Tasks and scheduling
  - Queues and intercommunication
  - Semaphores and synchronization
  - Other facilities
- RTOS standards

Configurability

- No overhead for unused functions tolerated
- No single OS fits all needs, ☞ configurability needed.
- Example: Configuration of VxWorks



The following components will be REMOVED:

C++ symbol demangler
symbol table
downloaded symbol table

WARNING: The following components depend on the components to the left and will also be REMOVED:

timex
target debugging
shell banner
target shell
target loader

Image size change (bytes):

text	data	bss	total
-89572	-14360	-1056	-1149[illegible]

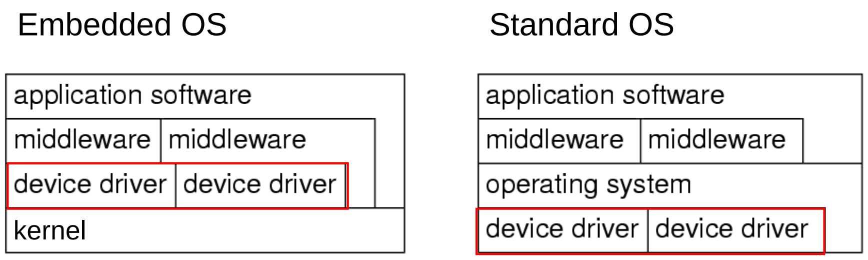
New image size (bytes):

text	data	bss	total
375852	4888	28716	408576

Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

Disk and Network Handled by Tasks

- Effectively no device needs to be supported by all variants of the OS, except maybe the system timer.
- Disk & network handled by tasks instead of integrated drivers



Protection is Optional

- Protection mechanisms not always necessary
  - ES typically designed for a single purpose
  - Untested programs rarely loaded
  - SW considered reliable.
- *Privileged* I/O instructions not necessary and tasks can do their own I/O.

Interrupts not restricted to OS

- Interrupts can be employed by any process
  - For standard OS: serious source of unreliability.
  - For embedded systems: efficient control over a variety of devices is required
- Embedded programs can be considered to be tested
- More efficient than going through OS services

Real-time capability

- The timing behavior of the OS must be **predictable**
  - $\forall$ services of the OS: Upper bound on the execution time!
- OS should manage the timing and scheduling
  - *Internal synchronization*: with one master clock or distributed
  - *External synchronization*: guarantees consistency with actual physical time (GPS, etc.)
- The OS must be *fast*
  - Practically important

Functionality of RTOS-Kernels

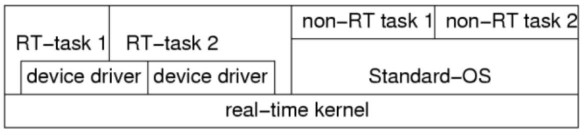
- processor management
- memory management,
- and timer management;

$\left.\vphantom{\begin{matrix}a\\b\\c\end{matrix}}\right\}$ resource management

- task management (resume, wait etc),
- inter-task communication and synchronization.

Classes of RTOSs

- Fast proprietary kernels
  - QNX, PDOS, VCOS, VTRX32, VxWORKS
- RT extensions to standard Oss



- Research RTOSs trying to avoid limitations
  - MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody

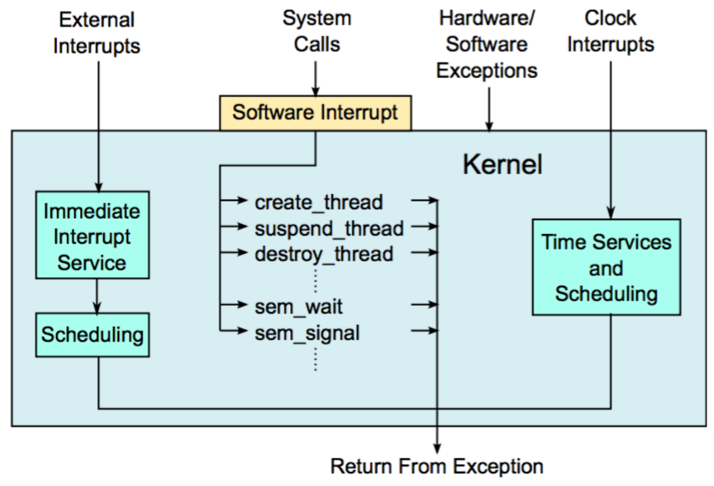
Outline of a Microkernel Scheduler

- Scheduler may be part of a compiler or code generator, part of an OS or microkernel, or both.
- Main Scheduler Thread (Task)
  - set up periodic timer interrupts;
  - create default thread data structures;
  - dispatch a thread (procedure call);
  - execute main thread (idle or power save, for example).
- Thread data structure
  - copy of all state (machine registers)
  - address at which to resume executing the thread
  - status of the thread (e.g. blocked on mutex)
  - priority, WCET (worst case execution time), and other info to assist the scheduler

Outline of a Microkernel Scheduler

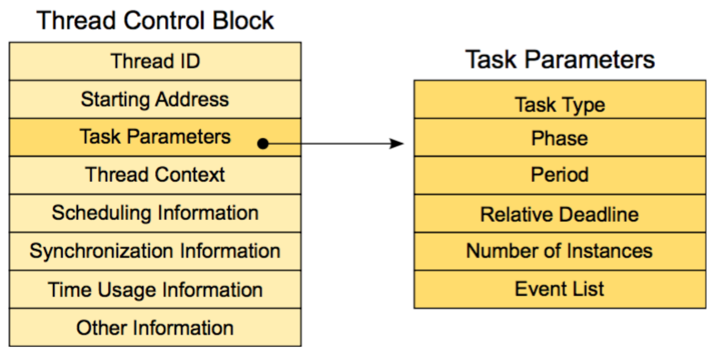
- Timer interrupt service routine:
  - dispatch a thread.
- Upon dispatching a thread
  - *disable interrupts;*
  - determine which thread should execute (scheduling);
  - if the same one, enable interrupts and return;
  - save state (registers) into current thread data structure;
  - save return address from the stack for current thread;
  - copy new thread state into machine registers;
  - replace program counter on the stack for the new thread;
  - *enable interrupts;*
  - return.

Structure of A Microkernel



Thread Control Block (TCB)

The Thread Control Block keeps all information needed to manage and schedule the thread.



Thread Control Block (TCB)

When a thread is

- *executing*, its context is changed continuously.
- *stops execution*, its context is stored in the thread's TCB.
- '*put in a queue*' (ready queue, pending queue for semaphore, . . .), a pointer to the threads TCB is put in the linked list representing the queue.
- *destroyed*, the kernel deletes the TCB and deallocates its memory space.

Task (Implementing) Function

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample variable.  This would not be true if the variable was
declared static – in which case only one copy of the variable would exist
and this copy would be shared by each created instance of the task. */
int iVariableExample = 0;

    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Creating a Task in FreeRTOS

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed char * const pcName,
                           unsigned short usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                         );
```

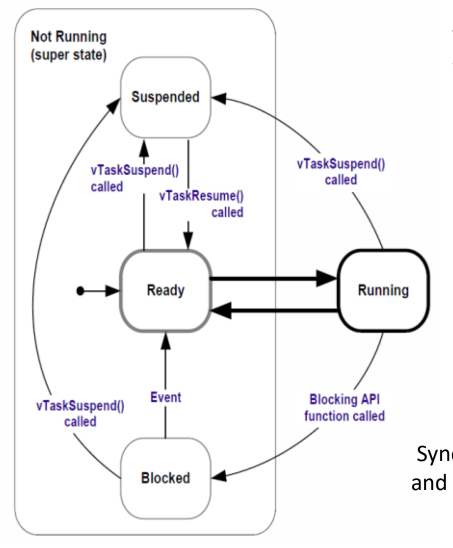
- pvTaskCode – A pointer to the task function
- pcName – Descriptive task name used during profiling
- usStackDepth – Stack depth/size
- pvParameters – Pointer to task parameters
- uxPriority – Task priority
- pxCreatedTask – Returns a task "handle"

➢ Returns success or failure

Major States of A Thread

- **Sleeping:** A thread is in sleeping state after creation or completion. It will stay in this state until it is released. It is not eligible for execution.
- **Ready:** A thread is in this state, when it released or preempted and is not blocked. It is eligible for execution.
- **Running:** A thread is in this state, if it is executing.
- **Suspended (Blocked):** A thread is in this state if it is released but has not yet completed, since it cannot proceed to a lack of a resource or budget.
- **Terminated:** A thread enters this state, if it will not execute again. Threads in this state may be destroyed.

Full Task State Machine of FreeRTOS



Tasks can enter the "Blocked" state to wait for two types of events:

-Temporal Events, e.g. delaying for a fixed amount or until an absolute time

-Synchronization Events, e.g., waiting for data

Synchronization events can come in many forms and can be combined with delay events, e.g. wait for data until a time period has elapsed.

System Calls

- A *system call* is a call to one of the RTOS API functions, like `xSemaphoreTake()` in the case of FreeRTOS.
- When the system call is executed, the kernel saves the context of the calling thread (and switches from user mode to kernel mode).
- The kernel executes then the system call and executes a return from exception(, which means that the system returns to user mode).
- Then the ready thread with the highest priority will be scheduled and starts to execute.

Fixed-Priority Scheduling

- All modern operating systems support fixed-priority scheduling.
- The assigned priority is kept in the TCB.
- If protocols like priority inheritance protocol are used, even current priority has to be part of the TCB.
- Kernel maintains a ready queue for each priority level. When a thread is ready to execute, it is put in the queue that corresponds to its current priority.

Dynamic Priorities

- Most operating systems support even dynamic priorities.
- This means that there is a system call that a thread can use in order to change its own or another threads priority.

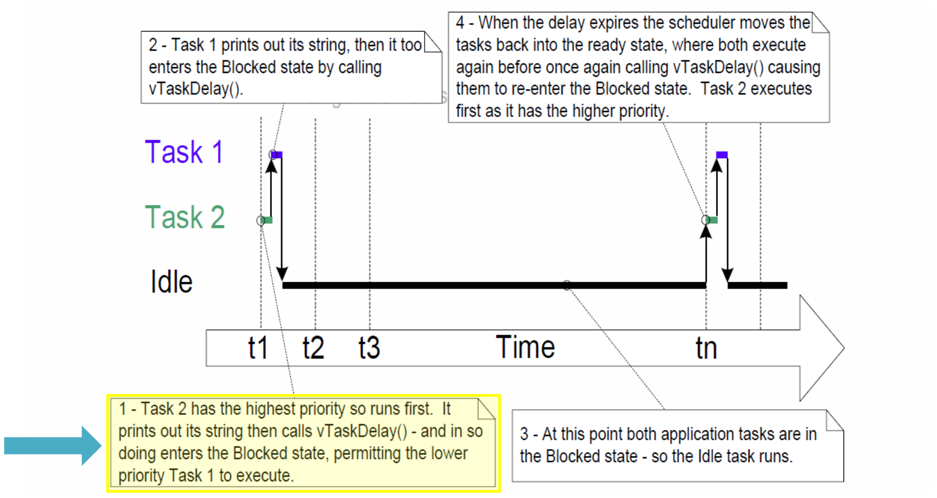
Example: Defining a Printer Task

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period.  This time a call to vTaskDelay() is used which
        places the task into the Blocked state until the delay period has expired.
        The delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Creating the Tasks

```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

int main( void )
{
    /* Create the first task at priority 1.  The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 240, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2. */
    xTaskCreate( vTaskFunction, "Task 2", 240, (void*)pcTextForTask2, 2, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well we will never reach here as the scheduler will now be
    running.  If we do reach here then it is likely that there was insufficient
    heap available for the Idle task to be created. */
    for( ;; );
}
```

Task Execution


2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().
4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.
Task 1
Task 2
Idle
t1
t2
t3
Time
tn
1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.
3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

Queue Access

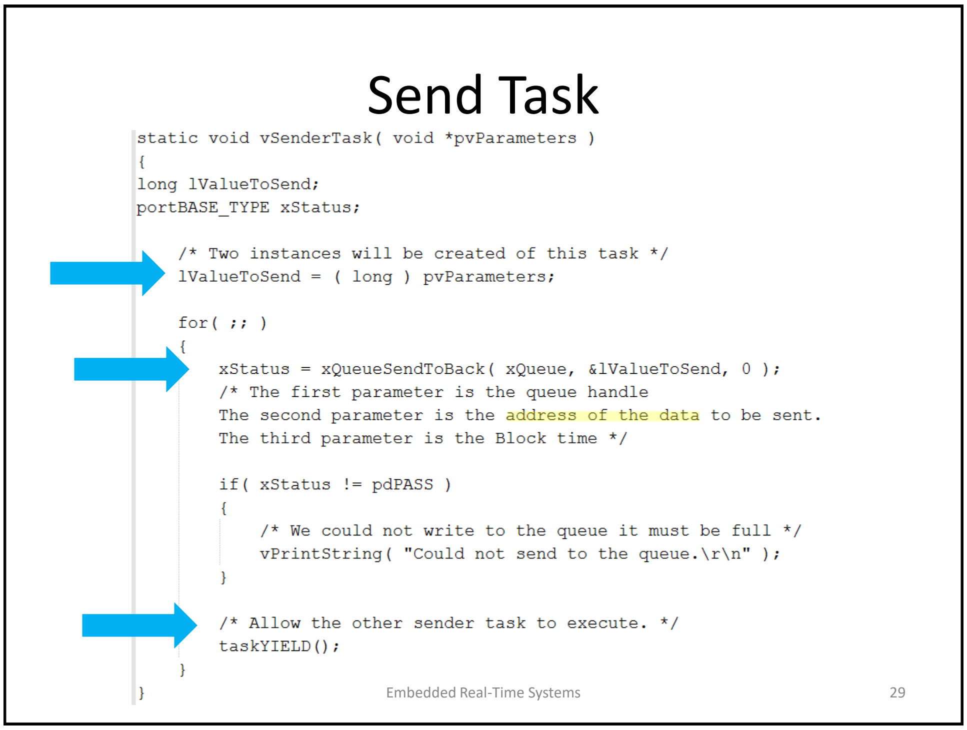
- Accessed through *a handle* that is returned when the queue is created
- Operates as a **FIFO**, but can write to front
- Can also “peek”, i.e., receive a copy of data, or “query” the number of items in the queue
- Access can include a *maximum blocking time* to wait if the queue is full or empty (in ticks)
- Special API functions must be used to access from an Interrupt Service Routine (ISR)

Example using Queues

- Single queue – multiple send, single read task
- Send tasks are low priority and don't block
- Receive task is higher priority and does have a specified blocking time
- Thus, as soon as an item is written the read unblocks and empties queue
- As a result, queue should never have more than one item

main()

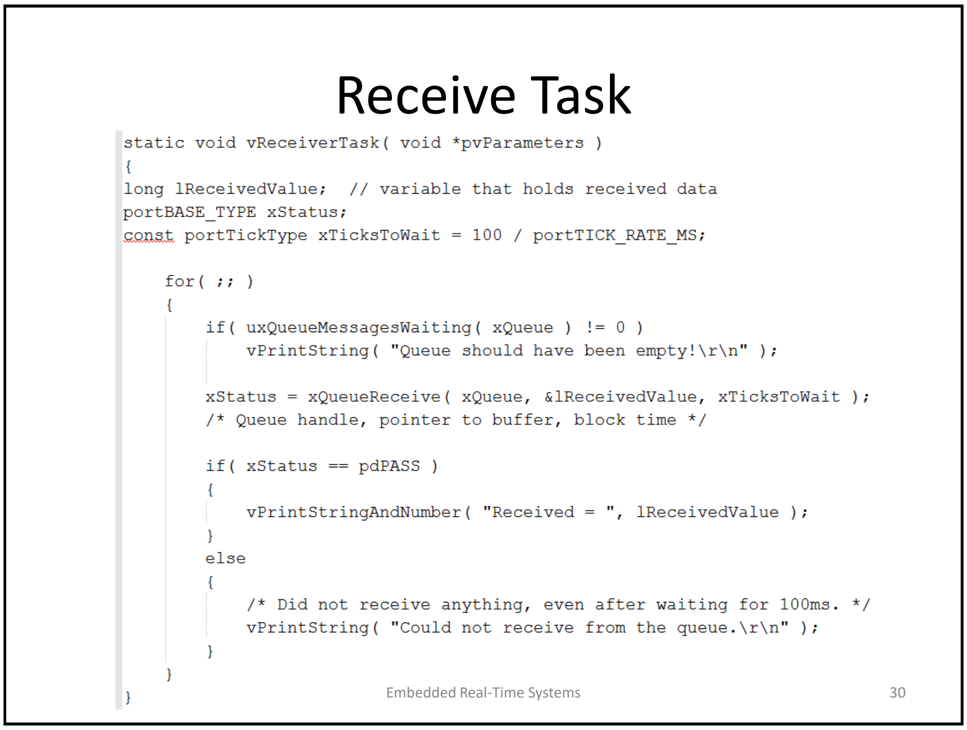
```
xQueueHandle xQueue; // Global queue handle used by all tasks

int main( void )
{

    vSetupEnvironment(); // Configure HW

    /* The queue is created to hold a maximum of 5 long values. */
    xQueue = xQueueCreate( 5, sizeof( long ) );

    if( xQueue != NULL ) // Successfully created
    {
        /* Two send tasks, priority 1. Each write 100 or 200 to queue */
        xTaskCreate( vSenderTask, ( signed char * ) "Sender1", 240, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, ( signed char * ) "Sender2", 240, ( void * ) 200, 1, NULL );

        /* Read task, priority 2 */
        xTaskCreate( vReceiverTask, ( signed char * ) "Receiver", 240, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {       /* The queue could not be created. */    }

    /* If all is well we will never reach here  */
    for( ;; );
    return 0;
}
```



Send Task

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
portBASE_TYPE xStatus;

    /* Two instances will be created of this task */
    lValueToSend = ( long ) pvParameters;

    for( ;; )
    {
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        /* The first parameter is the queue handle
        The second parameter is the address of the data to be sent.
        The third parameter is the Block time */

        if( xStatus != pdPASS )
        {
            /* We could not write to the queue it must be full */
            vPrintString( "Could not send to the queue.\r\n" );
        }

        /* Allow the other sender task to execute. */
        taskYIELD();
    }
}
```

Embedded Real-Time Systems 29

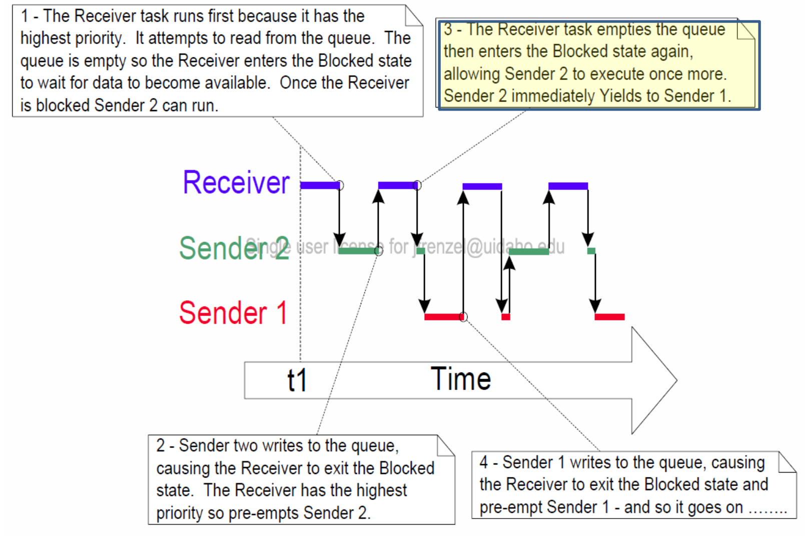


Receive Task

```
static void vReceiverTask( void *pvParameters )
{
long lReceivedValue;  // variable that holds received data
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    for( ;; )
    {
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
            vPrintString( "Queue should have been empty!\r\n" );

        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
        /* Queue handle, pointer to buffer, block time */

        if( xStatus == pdPASS )
        {
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* Did not receive anything, even after waiting for 100ms. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

Embedded Real-Time Systems 30

Task Execution


1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Once the Receiver is blocked Sender 2 can run.
3 - The Receiver task empties the queue then enters the Blocked state again, allowing Sender 2 to execute once more. Sender 2 immediately Yields to Sender 1.
Receiver
Sender 2
Sender 1
t1
Time
2 - Sender two writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.
4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ……..

Tips

- For large amounts of data, send a pointer to the data (i.e., shared memory buffer)
- Care must be taken to ensure that sender doesn’t modify RAM until it has been read
- If memory allocated dynamically, only one task should free the memory
- Beware sharing data on a task stack via a pointer, as the stack frame might change (Use global?)

Binary Semaphores

- Can be used for synchronization between tasks
- Can also be used for connecting an ISR to a task
  - The "handler" task is written to block, waiting for a "binary semaphore" (i.e., signal)
  - The handler is trying to "take" a semaphore, and the ISR is written to "give" the semaphore
- A binary semaphore is analogous to a one-element queue, i.e., full or empty
- But rather than return to the preempted task, control is transferred to the handler …

Example: Periodic Task

```
static void vPeriodicTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* This task is just used to 'simulate' an interrupt.  This is done by
        periodically generating a software interrupt. */
        vTaskDelay( 500 / portTICK_RATE_MS );

        /* Generate the interrupt, printing a message both beforehand and
        afterwards so the sequence of execution is evident from the output. */
        vPrintString( "Periodic task - About to generate an interrupt.\n" );
        mainTRIGGER_INTERRUPT();
        vPrintString( "Periodic task - Interrupt generated.\n\n" );
    }
}
```

Example: Periodic Task ISR

```
void vSW1_ISR_Handler( void )
{
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    /* 'Give' the semaphore to unblock the task. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* Clear the software interrupt flag. */
    mainCLEAR_INTERRUPT();

    /* Then clear the interrupt in the interrupt controller. */
    IFS0CLR = mainSW1_INT_BIT;

    /* Giving the semaphore may have unblocked a task - if it did and the
    unblocked task has a priority equal to or above the currently executing
    task then xHigherPriorityTaskWoken will have been set to pdTRUE and
    portEND_SWITCHING_ISR() will force a context switch to the newly unblocked
    higher priority task.

    NOTE: The syntax for forcing a context switch within an ISR varies between
    FreeRTOS ports.  The portEND_SWITCHING_ISR() macro is provided as part of
    the PIC32 port layer for this purpose.  taskYIELD() must never be called
    from an ISR! */
    portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
}
```

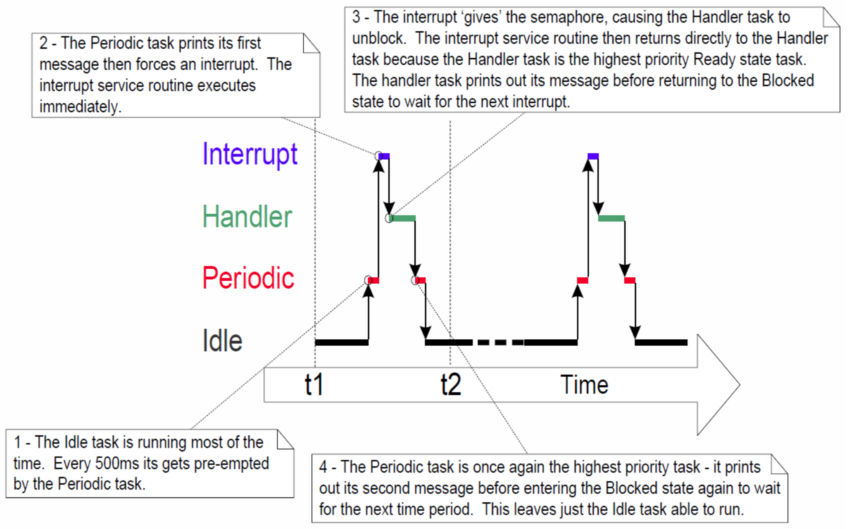
Example: Periodic Task Task Handler

```
static void vHandlerTask( void *pvParameters )
{
    /* As per most tasks, this task is implemented within an infinite loop.

    Take the semaphore once to start with so the semaphore is empty before the
    infinite loop is entered.  The semaphore was created before the scheduler
    was started so before this task ran for the first time.*/
    xSemaphoreTake( xBinarySemaphore, 0 );

    for( ;; )
    {
        /* Use the semaphore to wait for the event.  The task blocks
        indefinitely meaning this function call will only return once the
        semaphore has been successfully obtained - so there is no need to check
        the returned value. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* To get here the event must have occurred.  Process the event (in this
        case we just print out a message). */
        vPrintString( "Handler task - Processing event.\n" );
    }
}
```

Example: Periodic Task main ()

```
int main( void )
{
    /* Configure both the hardware and the debug interface. */
    vSetupEnvironment();

    /* Before a semaphore is used it must be explicitly created.  In this example
    a binary semaphore is created. */
    vSemaphoreCreateBinary( xBinarySemaphore );

    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {
        /* Create the 'handler' task.  This is the task that will be synchronized
        with the interrupt.  The handler task is created with a high priority to
        ensure it runs immediately after the interrupt exits.  In this case a
        priority of 3 is chosen. */
        xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );

        /* Create the task that will periodically generate a software interrupt.
        This is created with a priority below the handler task to ensure it will
        get preempted each time the handler task exits the Blocked state. */
        xTaskCreate( vPeriodicTask, "Periodic", 240, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }

    /* If all is well we will never reach here as the scheduler will now be
    running the tasks.  If we do reach here then it is likely that there was
    insufficient heap memory available for a resource to be created. */
    for( ;; );
}
```

Example: Periodic Task Task Execution



2 - The Periodic task prints its first message then forces an interrupt. The interrupt service routine executes immediately.

3 - The interrupt 'gives' the semaphore, causing the Handler task to unblock. The interrupt service routine then returns directly to the Handler task because the Handler task is the highest priority Ready state task. The handler task prints out its message before returning to the Blocked state to wait for the next interrupt.

1 - The Idle task is running most of the time. Every 500ms its gets pre-empted by the Periodic task.

4 - The Periodic task is once again the highest priority task - it prints out its second message before entering the Blocked state again to wait for the next time period. This leaves just the Idle task able to run.

Some other FreeRTOS Facilities

- Counting semaphores
- Mutexes
- Software timers
- Event groups
- (Heap) memory management
- Dynamic priority management
- Kernel configuration

RTOS Standards

- There exist a number of standards for real-time operating systems.
- At present the standards specify portability at the source code level, but the code needs to be recompiled for each operating system.
- At present there are the following main operating system standards
  - RT-POSIX - general-purpose operating system with real-time extensions
  - OSEK - automotive industry
  - APEX - avionic systems
  - µITRON - embedded systems

RT-POSIX

- The real-time extension of POSIX, which is adopted by many operating systems.
- Specifies a large set of system calls, for advanced RTOS functions
  - priority inheritance
  - sporadic server scheduling
  - execution time budgeting
  - virtual memory management
- Four real-time profiles are defined by POSIX.13:
  - Minimal real-time system profile (PSE51) - small embedded systems with a memory footprint with tens of kilobyte.
  - Real-time controller profile (PSE52) - complete file system added
  - Dedicated real-time system profile (PSE53) - large embedded systems, includes memory protection
  - Multi-purpose real-time system profile (PSE54) - general purpose OSs

Examples:
- QNX
- RTEMS
- VxWorks