Outline

Introduction

Preliminaries: FreeRTOS Start Up

Task creation and Management

FreeRTOS Scheduling
Outline

Introduction

Preliminaries: FreeRTOS Start Up

Task creation and Management

FreeRTOS Scheduling
Timing parameters of a job $J_j$

- Arrival time ($a_j$) or release time ($r_j$) is the time at which the job becomes ready for execution.
- Computation (execution) time ($C_j$) is the time necessary to the processor for executing the job without interruption.
- Absolute deadline ($d_j$) is the time at which the job should be completed.
- Relative deadline ($D_j$) is the time length between the arrival time and the absolute deadline.
- Start time ($s_j$) is the time at which the job starts its execution.
- Finishing time ($f_j$) is the time at which the job finishes its execution.
- Response time ($R_j$) is the time length at which the job finishes its execution after its arrival, which is $f_j - a_j$. 

\[ C_j \]
\[ a_j \quad s_j \quad f_j \quad d_j \]
\[ R_j \quad D_j \]

\text{time}
State of Jobs

• A job is either running or not running:

• All the jobs that are ready for executions are put into a ready queue

• A “scheduler” decides which job in the ready queue obtains the privilege of the computing resources to run
Schedules for a set of jobs \( \{ J_1, J_2, \ldots, J_N \} \)

- A schedule is an assignment of jobs to the processor, such that each job is executed until completion.
- A schedule can be defined as an integer step function \( \sigma : \mathbb{R} \rightarrow \mathbb{N} \), where \( \sigma(t) = j \) denotes job \( J_j \) is executed at time \( t \), and \( \sigma(t) = 0 \) denotes the system is idle at time \( t \).
- If \( \sigma(t) \) changes its value at some time \( t \), then the processor performs a context switch at time \( t \).
- Non-preemptive scheduling: there is only one interval with \( \sigma(t) = j \) for every \( J_j \), where \( t \) is covered by the interval.
- Preemptive scheduling: there could be more than one interval with \( \sigma(t) = j \).
Scheduling Concept: Preemptive

**Schedule**: \( \sigma : \mathbb{R} \rightarrow \mathbb{N} \) function of processor time to jobs

\[ \sigma(t) \]

![Diagram showing scheduling concept with context switches](image)
Recurrent Tasks

- When jobs (usually with the same computation requirement) are released recurrently, these jobs can be modeled by a recurrent task.

- **Periodic Task** $\tau_i$:
  - A job is released exactly and periodically by a period $T_i$.
  - A phase $\phi_i$ indicates when the first job is released.
  - A relative deadline $D_i$ for each job from task $\tau_i$.
  - $(\phi_i, C_i, T_i, D_i)$ is the specification of periodic task $\tau_i$, where $C_i$ is the worst-case execution time.

- **Sporadic Task** $\tau_i$:
  - $T_i$ is the minimal time between any two consecutive job releases.
  - A relative deadline $D_i$ for each job from task $\tau_i$.
  - $(C_i, T_i, D_i)$ is the specification of sporadic task $\tau_i$, where $C_i$ is the worst-case execution time.
Multi-Tasking

• The execution entities (tasks, processes, threads, etc.) are competing from each other for shared resources
  • In FreeRTOS, we will only use “tasks” for the rest of the whole course.
• Scheduling decision policy is needed
  • When to schedule an entity?
  • Which entity to schedule?
  • How to schedule entities?
Multi-Tasking

All available tasks appear to be executing ...

... but only one task is ever executing at any time.
Outline

Introduction

Preliminaries: FreeRTOS Start Up

Task creation and Management

FreeRTOS Scheduling
Task States

- Ready: A task that is able to execute is not currently running due to its lower priority
- Running: When a task is actually executing
- Blocked: Waiting for either a temporal or external event (Always has a timeout)
- Suspended: No timeout period allowed

[http://www.freertos.org]
## Data Types in FreeRTOS

<table>
<thead>
<tr>
<th>Macro or typedef</th>
<th>Actual type</th>
</tr>
</thead>
<tbody>
<tr>
<td>portCHAR</td>
<td>char</td>
</tr>
<tr>
<td>portSHORT</td>
<td>short</td>
</tr>
<tr>
<td>portLONG</td>
<td>long</td>
</tr>
<tr>
<td>portTickType</td>
<td>Used to store the tick count and specify block times</td>
</tr>
<tr>
<td>portBASE_TYPE</td>
<td>Used to represent the most efficient data type for the architecture</td>
</tr>
</tbody>
</table>

- Signed or unsigned `char` types should be always specified
- Plain `int` types should never be used
Variables and Functions Naming

- **Variables: prefixes**
  - 'c': for char
  - 's': for short
  - 'l': for long
  - 'x': for portBASE_TYPE and any others
  - 'u': for unsigned
  - 'p': pointer
  - combinations are possible

- **Functions: prefixes**
  - by the returning data type
  - 'v': for void

- **Common macros:**
  - pdTRUE is 1, pdFALSE is 0
  - pdPASS is 1, pdFAIL is 0
## Macro prefixes

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>port (e.g., portMAX_DELAY)</td>
<td>portable.h</td>
</tr>
<tr>
<td>task (e.g., taskENTER_CRITICAL())</td>
<td>task.h</td>
</tr>
<tr>
<td>pd (e.g., pdTRUE)</td>
<td>projdefs.h</td>
</tr>
<tr>
<td>config (e.g., configUSE_PREEMPTION)</td>
<td>FreeRTOSConfig.h</td>
</tr>
<tr>
<td>err (e.g., errQUEUE_FULL)</td>
<td>projdefs.h</td>
</tr>
</tbody>
</table>
Configurations: FreeRTOSConfig.h

Some important fields/features:

- **configUSE_PREEMPTION**: This is set to 1 if the preemptive kernel is desired.
- **configUSE_IDLE_HOOK**: An idle task hook will execute a function during each cycle of the idle task.
- **configUSE_TICK_HOOK**: A tick hook function will execute on each RTOS tick interrupt if this value is set to 1.
- **configTICK_RATE_HZ**: This is the frequency at which the RTOS tick will operate.
- **configMAX_PRIORITIES**: The total number of priority levels that can be assigned when prioritizing a task.
- **configUSE_COUNTING_SEMAPHORES**: This is set to 1 if counting semaphores are required.
- **configUSE_MUTEXES**: This is set to 1 if mutexes are needed. Priority inheritance will then be enforced.
- etc...
Outline

Introduction

Preliminaries: FreeRTOS Start Up

Task creation and Management

FreeRTOS Scheduling
Structure of the Task

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

/* Tasks must not attempt to return from their implementing function or otherwise exit. If it is necessary for a task to exit then have the task call vTaskDelete( NULL ) to ensure its exit is clean. */
Create A Task

1 portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
2 const signed portCHAR * const pcName,
3 unsigned portSHORT usStackDepth,
4 void *pvParameters,
5 unsigned portBASE_TYPE uxPriority,
6 xTaskHandle *pxCreatedTask
7 );

- **calls** xTaskGenericCreate( ( pvTaskCode ), ( pcName ), ( usStackDepth ), ( pvParameters ), ( uxPriority ), ( pxCreatedTask ), ( NULL ), ( NULL ) )

- **pvTaskCode**: A function that performs the computation of the task
- **pcName**: Name of the task used for debugging
- **usStackDepth**: Stack size of the process (task)
- **pvParameters**: Parameters passed to the process (task)
- **uxPriority**: Priority level indexed from 0 (lowest) to configMAX_PRIORITIES-1(highest)
- **pxCreatedTask**: Handler when creating a task
Create A Task (cont.)

- return value of xTaskCreate()
  - pdTRUE: the task is created successfully
  - errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY: insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack

- All the related information of a task is stored in a task control block (TCB) so that the operating systems can use it for operating on a task.

<table>
<thead>
<tr>
<th>Top of Stack</th>
<th>pointer to the last item place on the stack for the task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task State</td>
<td>list item used to place the TCB in ready and blocked queues</td>
</tr>
<tr>
<td>Event List</td>
<td>list item used to place the TCB in the event lists</td>
</tr>
<tr>
<td>Priority</td>
<td>task priority (0=lowest)</td>
</tr>
<tr>
<td>Stack Start</td>
<td>pointer to the start of the process stack</td>
</tr>
<tr>
<td>Others</td>
<td>other information</td>
</tr>
</tbody>
</table>
Example - Task to be Created

```c
/* Task to be created. */
void vTaskCode( void * pvParameters )
{
    /* The parameter value is expected to be 1 as 1 is passed in the pvParameters value in the call to vTaskCreate() below. */
    configASSERT( ( ( uint32_t ) pvParameters ) == 1 );
    for( ;; )
    {
        /* Task code goes here. */
    }
}
```
Example - Function to Create a Task

1 /* Function that creates a task. */
2 void vOtherFunction( void )
3 {
4 portBASE_TYPE xReturned;
5 xTaskHandle xHandle = NULL;
6 /* Create the task, storing the handle. */
7 xReturned = xTaskCreate(
8 vTaskCode,     /* Function that implements the task. */
9     "NAME",     /* Text name for the task. */
10 1000,         /* Stack size in words, not bytes. */
11     ( void * ) 1, /* Parameter passed into the task. (if not, pass NULL) */
12     1,           /* Priority at which the task is created. */
13     &xHandle);  /* Used to pass out the created handle of the task. (if not, pass NULL) */
Example - Function to Create a Task (cont.)

1 if( xReturned == pdPASS )
2 {
3    /* The task was created. Use the handle of the task to delete the task. */
4    vTaskDelete( xHandle );
5  }
6 }
Examples of Recurrent Task Models

**Periodic task:** \((\phi_i, C_i, T_i, D_i) = (2, 2, 6, 6)\)

\[\uparrow \quad \downarrow\]

**Sporadic task:** \((C_i, T_i, D_i) = (2, 6, 6)\)
States of A Task

- Create Task
- Suspended
- Ready
- Running
- Blocked
- Deleted

xTaskCreate
Example: Sporadic Control System

Pseudo-code for this system

```
while (true)
    • start := get the system tick;
    • perform analog-to-digital conversion to get \( y \);
    • compute control output \( u \);
    • output \( u \) and do digital-to-analog conversion;
    • end := get the system tick;
    • \( \text{sleepTime} := T - (\text{end} - \text{start}) \);
    • sleep \( \text{sleepTime} \);
end while
```
void vTaskDelay(portTickType xTicksToDelay)

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.

- **xTicksToDelay**: The amount of time, in tick periods, that the calling task should block.
- **INCLUDE_vTaskDelay** must be defined as 1 for this function to be available.
- **vTaskDelay()** specifies a time at which the task wishes to unblock relative to the time at which vTaskDelay() is called.
Example: Periodic Control System

Pseudo-code for this system

set timer to interrupt periodically with period \( T \);

at each timer interrupt

do

- perform analog-to-digital conversion to get \( y \);
- compute control output \( u \);
- output \( u \) and do digital-to-analog conversion;

od
Delay a task until a specified time. This function can be used by periodic tasks to ensure a constant execution frequency (period).

- **INCLUDE** `vTaskDelayUntil` must be defined as 1 for this function to be available.
- It differs from `vTaskDelay`:
  - `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.
An Example for vTaskDelayUntil

```c
void vTaskFunction( void * pvParameters )
{
    portTickType xLastWakeTime;
    const portTickType xFrequency = 10;
    xLastWakeTime = xTaskGetTickCount();
    for( ;; )
    {
        //do something
        vTaskDelayUntil( &xLastWakeTime, xFrequency );
    }
}
```
States of A Task

- Create Task
- Suspended
- Ready
- Running
- Blocked
- Deleted

Transition Functions:
- `xTaskCreate`
- `vTaskDelay`
- `vTaskDelayUntil`
Outline

Introduction

Preliminaries: FreeRTOS Start Up

Task creation and Management

FreeRTOS Scheduling
vTaskStartScheduler()

```c
int main(void)
{
    xTaskCreate(hello, "Task 1", 1000, NULL, 1, NULL);
    vTaskStartScheduler();
    for(;;);
}
```
vTaskStartScheduler()

- Starts the real-time kernel tick processing.
- After calling, the kernel has control over which tasks are executed and when.
- This function does not return until an executing task calls vTaskEndScheduler().
- At least one task should be created via a call to xTaskCreate() before calling vTaskStartScheduler().
- An idle task is created automatically when the function is called.
  - It has a priority of zero (tskIDLE_PRIORITY).
Idle Task

• The processor requires something to execute, unless it is halt
• An idle task is a dummy procedure
• We can add a hook function `vApplicationIdleHook()` to an idle task
  • e.g., to measure the amount of processing capacity
  • e.g., to place the processor to the low-power mode
  • `vApplicationIdleHook()` must not, under any circumstances, call a function that might block. This will result in a scenario that no task is available to enter the Running state
• The idle task is also responsible for cleaning up the kernel resources used by a deleted task. It should be done in a reasonable amount of time.
• The idle task can be configured to be preempted
  • only at ticks (`#define configIDLE_SHOULD_YIELD 0`) or
  • at any time (`#define configIDLE_SHOULD_YIELD 1`)