G-RTOS Documentation

Implementing a simple real time operating system that works on ARM Cortex-M4 Processors.

(TM4C123GXL Tiva C Development Board)

Project made by: Youssef Galal

Project Link on GitHub:
https://github.com/Y-Galal/G-RTOS

Table of Contents

1-Introduction:

Real Time Operating Systems are a segment or a part of the whole program that decides the next task, task priority, handles the task messages and coordinates all of the tasks. A Real time operating system handles some tasks or routines to be run. The kernel of the operating system assigns CPU attention to a particular task for a period of time. It also checks the task priority, arranges the massages from tasks and schedules.

There are many types of schedulers. In this project I implemented Preemptive Scheduling algorithm that enables the user to assign task priorities. Also, the scheduler uses time slicing to switch between tasks in a fixed time which the user can choose.

There are many specifications that RTOS developers implement in their system to enable the user to synchronize between tasks and communicate between them. In this project I implemented Semaphores for task synchronization and Queues for inter-task communication. Also, there are some task utilities that help the user to delay, block and resume a task.

The system uses the internal Systick timer in the Cortex-M4 processor to apply time slicing and allow context switching after some time to schedule between different tasks.

This documentation provides a brief explanation for every function in the system and some important notes about it.

2-The System Files

- **kernel.c:** A source file that contains kernel functions.
- **kernel.h:** A header file that contains kernel functions declarations.
- **oskernel.asm:** An assembly file that contains the systick interrupt handler.
- **macros.h:** A header file that contains useful macros.
- **queue.h:** A header file that contains the function declarations for the queues.
- **queue.c:** A source file that contains queues' routines.
- **queue_structs.h:** A header file that has the important structures for the queues.
- **RTOS_Config.h:** A header file that contains important definitions for the system.
- **semaphore.h:** A header file that contains the functions declaration.
- **semaphore.c:** A source file that has the routines needed for the semaphores.
- **systick_registers.h:** A header file that contains the definitions for the systick registers in Cortex-M4 Processors.

Note: You should config the system by defining the macros in the RTOS_Config.h header file.

Note: Everything created in the system is created statically. No dynamic allocation used in the OS.

3-Task and Scheduler API

Kernel files are the files which have the entire OS functions that provide the implementation of the scheduler and also the task utilities. You should include kernel.h to use these functions.

3.1 Functions Description:

3.1.1 osKernelInit():

void osKernelInit(void);

This function must be called before you start initializing the system to avoid false interrupts and to ensure that the OS will be initialized successfully.

Parameters: Void

Return Values: Void

3.1.2 TaskCreate();

*Void TaskCreate(void(*task)(), uint8_t priority , uint32_t *TaskHandle);*

This function creates a task. Each task requires RAM that is used to hold the task state (the task control block, or TCB), and used by the task as its stack. Newly created tasks are initially placed in the Ready state.

Important Note: due to system defects, You should create the higher priority tasks first.

Parameters:

- void(*task)() : A pointer to the task function that's required to be the task code. Simply, this is the function name.
- uint8_t priority: The priority given for this task from 1 to NUM_OF_TASKS.
- uint32_t *TaskHandle: This is the task handle that will be used for task control in functions like TaskDelay.

Return Values: Void

3.1.3 osStartSystem():

void osStartSystem(void);

This function starts the scheduler by initializing the systick timer and switching to the highest priority task. If this function is working properly, it can't return to the main code.

Parameters: Void

Return Values: Void

3.1.4 TaskDelay():

void TaskDelay(uint32_t TaskHandle,uint32_t delayInTicks);

This function is responsible for delaying a task for systick ticks time by changing the task status to the (DELAYED) state. You can convert from time in milliseconds to systick ticks by using the macro MS_TO_TICKS() that is provided in macros.h file.

Parameters:

- uint32_t TaskHandle: The task handle for a function to be delayed
- uint32_t delayInTicks: The ticks required to delay this task.

Return Values: Void

3.1.5 TaskBlock():

void TaskBlock(uint32_t TaskHandle);

This function blocks a task forever as it changes the task status to the (BLOCKED) state. You can unblock this task by calling TaskResume.

Parameters: uint32_t TaskHandle: The task handle for a function to be blocked.

Return Values: Void

3.1.6 TaskResume();

void TaskResume(uint32_t TaskHandle);

This function changes the task status to be in the (READY) state. When you call this function and give it a handle of a task that's blocked, it will unblock it. If the task wasn't blocked, it will have no effect.

Parameters: uint32_t TaskHandle: The task handle for a function to be resumed.

Return Values: Void

4-Queue API

Queues are important in real time operating systems as they hold the data from a task to another task. This provides the inter thread communication. In this system you can create only one queue with uint32_t size and its length must be defined in the RTOS_Config.h as everything is created statically in the compile time.

4.1 Functions Description:

4.1.1 QueueInit():

*void QueueInit(QueueType_t *QueueHandle);*

This function initializes your queue. You should create an array of the length of the queue you need in your code and pass the pointer to this array to this function to initialize the queue.

Parameters: QueueType_t QueueHandle: The handle of the queue.

Return Values: Void

4.1.2 QueueSend():

void QueueSend(uint32_t TaskHandle, QueueType_t QueueHandle, uint32_t data,*

uint32_t delayTime);

This function is responsible for sending data to the queue. If the queue is full, it will block for the time passed to this function.

Parameters:

- uint32_t TaskHandle: The current task you're calling the function from.
- QueueType_t QueueHandle: The handle of the queue.
- uint32_t data: The data you want to send in the queue.
- uint32_t delayTime: The delay time in ticks if the queue is full.

Return Values: Void

4.1.3 QueueReceive():

void QueueReceive(uint32_t TaskHandle,QueueType_t QueueHandle, uint32_t*data, uint32_t delayTime);*

This function is responsible for receiving data to the queue. If the queue is empty, it will block for the time passed to this function.

Parameters:

- uint32_t TaskHandle: The current task you're calling the function from.
- QueueType_t QueueHandle: The handle of the queue.
- uint32_t *data: A pointer to a uint32_t variable to receive data in it.
- uint32_t delayTime: The delay time in ticks if the queue is full.

Return Values: Void

5-Semaphore API

Semaphore are counting variables used for synchronizing between tasks and also to protect shared resources to avoid the shared data corruption problem.

5.1 Functions Description:

5.1.1 SemaphoreCreate():

*void SemaphoreCreate(int32_t startValue,int32_t *semaphoreHandle);*

This function creates a semaphore and initializes its start value. If the start value is 0, the semaphore is initialized as taken.

Parameters:

- int32_t startValue: The start value of the semaphore. If the start value is 0, the semaphore is initialized as taken.
- int32_t *semaphoreHandle: a pointer to the semaphore handle that is created in the application code.

Return Values: Void

5.1.2 SemaphoreTake():

*void SemaphoreTake(int32_t *semaphoreHandle,uint32_t TaskHandle, uint32_t delayInTicks);*

This function is responsible for taking the semaphore. If the semaphore is already taken, the task will block for a time passed to this function.

Parameters:

- int32_t *semaphoreHandle: a pointer to the semaphore handle that is created in the application code.
- uint32_t TaskHandle: The current task you're calling the function from.
- uint32_t delayInTicks: The delay time in ticks if the semaphore is taken.

Return Values: Void

5.1.2 SemaphoreGive():

*void SemaphoreGive(int32_t *semaphoreHandle);*

This function is responsible for giving the semaphore.

Parameters:

- int32_t *semaphoreHandle: a pointer to the semaphore handle that is created in the application code.

Return Values: Void

6-RTOS_Config.h File

This file is very important that the user should create to define some macros to make the system work properly.

6.1 Macros description:

6.1.1 Target Frequency:

#define TARGET_FREQUENCY

This macro is responsible for defining the frequency of the development board you use. The frequency here is in Hertz.

6.1.2 The Stack Size:

#define STACK_SIZE

As the system allocates the memory statically, you should define the stack size used for the tasks. If the stack size macro is defined to be 100, the system allocates 400 bytes for every task.

6.1.3 Number of tasks:

#define NUM_OF_TASKS

The number of tasks is important to statically create the TCBs for the tasks. This macro only holds the number of the application tasks but the actual number of tasks is NUM_OF_TASKS + 1 as the system automatically creates additional idle task with the lowest priority in the system.

6.1.4 The Tick Rate:

#define TICK_RATE_MS

The tick rate is the time in milliseconds that the system will fire an interrupt to switch between tasks.

6.1.5 The length of the queue:

#define QUEUE_LENGTH

The length of the queue used in the system. If you're not using queues, this has not any effect.