

RTOS:

Non-preemptive scheduler manual

**Guillermo Francisco Ramirez Vásquez**

**Óscar Francisco Miranda García**

Automotive Entry Program 2015

Coach: Carlos Calvillo

**Continental México**

Campus: Santa Anita

Date: November 16th, 2015

Content

[Introduction 3](#_Toc435519524)

[Features 3](#_Toc435519525)

[Fundamentals 3](#_Toc435519526)

[Clock Tick 3](#_Toc435519527)

[Tasks 4](#_Toc435519528)

[Use of the scheduler 5](#_Toc435519529)

# Introduction

This scheduler is a non-preemptive task delay depending. It’s been developed for simple projects and soft real systems. It was implemented in a Freescale MPC5606B development board. It allows the user to add as much tasks as needed and configure the delays and clock tick, etc.

# Features

This manual includes and provides:

* The general scheduler code
* The capability to add several tasks,
* Modify the task’s period and offset,
* And establish the clock tick.

# Fundamentals

## Clock Tick

The clock tick is set in the timer module (STM) of the hardware that runs at 16 MHz (this value can be modified) and an ISR function is used.

To establish the clock tick use the CMP register of the STM module using the following formula:

Were *Clock Tick Hex* is a hexadecimal value, *Clock Speed Hex* is the hexadecimal magnitude of the clock speed, and *Desired Clock Tick Time* is the value in seconds of the Clock Time.

To keep the control of the clock tick a header, OS\_Init.h, is used. It has a variable used to mark the deadline of the tick and the interrupt function **Tick\_Flag()** that clears the **ub\_TickFlag**, the clock interrupt flag of the STM, and reset the CNT register. This function is shown in the OS\_Init.c file of the next page.

OS\_Init.h

**T\_UBYTE ub\_TickFlag;**

**void** **Tick\_Flag**(**void**);

OS\_Init.c

**void** **Tick\_Flag**(**void**)

{ /\* -------------------------------------------------------------------

\* Name : Tick\_Flag

\* Description : Check if the channel 0 of the STM as

reached 10ms and raise a flag when

reached

\* Parameters : void

\* Return : void

\* ----------------------------------------------------------------

\*/

**if** (STM.CH[0].CIR.B.CIF)

{

ub\_TickFlag = 1; /\* Clear tick clock flag \*/

STM.CH[0].CIR.B.CIF = 1; /\* Clear interrupt flag \*/

STM.CNT.R = 0; /\*Reset counter\*/

}

}

## Tasks

To add a task:

In the TASK\_Init.h add as much enums as necessary inside **E\_TASK** enum, the most important constant is **E\_TASK\_NUM** that tells the number of the total tasks of the scheduler, this one must not be modified or moved as its used in other modules of the program.

Each task is defined by the following:

A structure S\_TASK has three members:

* A handler called **rp\_Tasks** to call the task,
* A 32 bit **rul\_Period**, which is used to define the period of such task,
* A 32 bit **rul\_Offset**, which is used to define the task’s offset.

The tasks are declared and defined in both, the TASK\_Init.h and Task\_Init.c files:

TASK\_Init.h

**typedef** **struct** {

**void**(\* rp\_Tasks)(**void**); /\*Pointer that'll call every task\*/

T\_ULONG rul\_Period; /\*Period of task\*/

T\_ULONG rul\_Offset; /\*Offset of task\*/

}S\_TASK;

**typedef** **enum**{

*E\_TASK1*,

*E\_TASK2*,

…/\* NEW TASK MUST BE ADDED HERE \*/ …

*E\_TASK\_NUM*

}E\_TASK;

In the TASK\_Init.c, an array of **S\_TASK** variables must be used to define the tasks pointer, and to define the periods and offsets the format will be the following:

TASK\_Init.c

**#define** taskPeriod1 XXXX

… …

**#define** taskPeriodN XXXX

**#define** taskOffset1 XXXX

… …

**#define** taskOffsetN XXXX

**const** S\_TASK taskName[*E\_TASK\_NUM*] = {

{taskName1, taskPeriod1, taskOffset1 },

{taskName2, taskPeriod2, taskOffset2 },

{taskName3, taskPeriod3, taskOffset3 },

… … … …

{taskNameN, taskPeriodN, taskOffsetN }

};

Note that **E\_TASK\_NUM** represents the number of tasks, in this case, four tasks, and that the period and the offset are numeric values.

## Use of the scheduler

The scheduler controls the tasks with delays that are defined by the clock tick. To have a good performance of each task, a task must be small enough to fit within the clock tick. If a task is larger than the clock tick, it would be pause and will be dispatched in the next clock tick if other tasks are waiting. A task offset is necessary to avoid executing tasks at the same time, but this only reduces the possibility of such case, it doesn’t eliminate it. The next pictures shows the good practice of using an offset:

Task 1

Task 2

Offset

Task 3

Offset

Task 4

Fig. 1. Using offsets

All the tasks request to be dispatched at the same time. This is a critical problem.

Task 1

Task 2

Task 3

Task 4

ct 2ct 3ct 4ct 5ct 6ct

Fig. 1. Without offset

The scheduler is contained inside a function called kernel, this function is in TASK\_Init.c. This function starts by doing an indexing of the task, assigning the offset. This process occurs only once in each execution of the code.

Then an infinite loop starts, the tick of the scheduler is defined by a flag called ub\_TickFlag, which was configured to raise every millisecond. Once it was acquired it is restarted to zero.

Now, inside the if there is a for cycle which who through all the task accessing to their offset with and if, whenever the offset is over zero it will decrease, for every task. When one of the offset, of whichever task, reach zero, the else of the if will be active and will stablish the period of the task and will call it using the pointer previously defined.

