Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Especialidad en Sistemas Embebidos



**Design and implementation of a preemptive operative system on Cortex M4 architecture and its usage on a recycling machine**

Tesina para obtener el grado de:

Especialista en Sistemas Embebidos

Presentan: Antonio Rodríguez Soto

Director: Haga clic aquí para escribir texto.

San Pedro Tlaquepaque, Jalisco. 17 de Octubre de 2018.

Esta página es para la dedicatoria (opcional).

Es importante notar que a partir de esta página el documento utiliza un formato distinto. Esto se hace mediante la introducción de un “Salto de sección (Página impar)” de *Microsoft Word*. Para poder observar dónde está insertado el salto de página, es necesario dar un clic en el ícono ¶ que aparece en la parte alta de *Word*.

También es importante notar que la siguiente página (iv) no aparece numerada ya que debe estar en blanco. Al mandar imprimir o al generar el pdf esa página aparecerá en blanco, sin el número iv, aunque corresponda a ese número. El resumen de la tesis comienza en la página v.

Todos los inicios de Capítulos o de secciones principales (como la Introducción, etc.), deben iniciar en una página impar. Esto se hace automáticamente mediante la inserción del salto de sección al final de cada capítulo o de cada sección principal.

# [A](https://rei.iteso.mx/bitstream/handle/11117/5550/Wiener%20filtering%20for%20myoelectric%20signal.pdf?sequence=2)bstract

*México is recycling around 60 % of all PET bottles consumed in its internal marked and is leader in Food-grade recycling PET. México leads this industry above USA and Canada, the contribution is not just economical but obviously it’s a measure to improve our environment, when using recycled PET resin, 87% of emissions against virgin resin are reduced [10].*

# *Despite these positive aspects, recycling PET faces next two big challenges, a) The drop in oil prices is a disincentive for companies, since it is cheaper to acquire a virgin and non-recycled product, b) The process to recover a PET bottle from the dump is plenty of informality, reining freedom in the chain of supplement that´s easily corrupted till become a mafia.*

*The goal of this thesis is to tackle problem b), by designing and implementing an affordable bottle recycle machine. As seen in many other countries, recollection is not just done by scavengers but by the final costumer who pays a small tax by bottled product acquired, this tax is returned once the customer delivers back the bottle into a machine that prints a credit ticket with the returned amount.*

*This complex system is not supported in Mexico but the lecturer could take this work a draft of a proposal for PET waste handling.*

*Just to mention two non-functional requirement, the machine must cost less than $18,000 MXN, and should be smart enough to differentiate among glass and PET bottles. The idea is to move to an improved version but some mechanical, technical and software functional requirements ought to be tackled first.*

# Resumen

*Designing a PET or Glass recollection system similar than those utilized in big stores is a complex task at the level of a transnational company, not to mention the legal frame to make it happen.*

*However, to design and implement an affordable bottle recycle machine is not rocket science, as depicted in Chapter 1, the system hardware elements aren’t complex at all, it´s just matters of applying more time and effort with a team of self-motivated people.*

*In Chapter 2*

*In Chapter 5 is detailed the proposed algorithm that allows the usage of low cost Infra-red sensors devised as four transmitter-receiver pairs connected to an ADC, a self-calibration sequence calculates the necessary offset to compensate the hardware bias, an average is calculated out of 4K samples per pair and used to subtract the noise to each of the four pairs. This operation takes 32ms per channel. The only requirement is to block from the sensor stage from any beam of light.*

*The prototype shown in Chapter 7 was implemented for just one engineer and that’s why there’s room for improvement but it’s good enough to exercise the operative system explained in Chapter 2.*

scope of this project is aimed to design and test a fully preemptive operative System (OS) on a Cortex M4 architecture. While designing a preemptive operative system, enabling-disabling interrupts at the right moment is crucial for right OS behavior.

This work seeks to be a reference for those who want to implement their own embedded operative system and need some ideas.

A basic PET recycling machine is chosen as easy to build HW for testing purposes.

Content

Instituto Tecnológico y de Estudios Superiores de Occidente i

Abstract iii

Objectives iii

OS Requirements: iii

1. Chapter I System Elements. 3

1.1. Priority Buffers and Task Control Block 3

1.1. Task ID 4

1.2. Triggered Tasks, Task activation 4

1.1. Background function 4

1.2. Restore Context 5

1.3. Activate Task(TaskType taskID) 7

1.4. Dispatcher 7

1.5. Task descriptor and dispatcher relationship 10

1.6. Priority buffer 11

1.7. Task private variables (memory allocation) 11

1.8. AC to Digital conversion 12

2. Chapter II System Descrpition 15

2.1. Initialization 15

2.2. Task state 15

2.3. Calls from tasks 16

2.4. Tasks Termination 16

2.5. TASK stack 16

2.6. Stack distribution per Task 18

2.7. SysTick 19

2.8. PLL Initialization steps 20

3. Chapter III Cortex M4 Context change 21

3.1. Complementary information about ISR behavior 22

4. Chapter IV Serial drivers configuration 24

4.1. All serial ports are configured in a single file 24

4.2. All Channels available on MCU can be used 24

5. Measurements 25

5.1. Feeding machine, bottle to Inside 26

5.2. Feeding machine, bottle to Inside 28

5.3. Returning Bottle to outside 28

6. Prototype 29

7. Conclusion 30

8. Bibliografía 36

Introduction

Due that we are designing a RTOS with preemptive kernel in a multitasking environment, a Task control Block structure is allocated in RAM memory, it associates each task properties such as; Task ID, Priority, Context level, Task pointer, Task state, Stack information, Deadline and Its stack buffer size, such as a linked list.

This TCB is used for RTOS in commercial applications because the number of real-time tasks can vary allowing user interaction but mainly because it can be used to facilitate dynamic task prioritization.

A different memory management can include the maintenance of several blocks of memory that are allocated to individual applications as requested.

An alternative to multiple lists involves a single list in which only the status variable in the TCB is modified rather than moving the block. Thus, for example, when a task is moved from suspended to ready state, or from the ready to executing state, only the status word is changed. This approach has the advantage of less list management but slower traversal times, since the entire list must be traversed during each context switch to identify the next highest priority task that is ready to run.

The main drawback of the task-control block model is that when a large number of tasks are created, the overhead of the scheduler can become significant. The operating system manages the TCBs by keeping track of the status or state of each task.

# Recycling machine description

The prototype of the entrance stage for the recycling machine is depicted in Fig. 1‑1, the machine elements are listed on Table 1‑1.

One of the requirements to build the recycling machine is to make it as unexpansive as possible, this is achieved by reducing the cost of the PET sensor stage by using the set of paired IR sensors (3) in Fig. 1‑1 and compensate the bias offset by using (2‑1).

DC Motor Control

MC33931

2

5

1

TM4C123GXL Cortex **M4**

6

8. LCD

4

3

TM4C123GXL Cortex **M4**

Fig. 1‑1. Recycling machine Block Diagram and elements identification

On the left side of Fig. 1‑1, sensor (2) completes the sensor stage. It’s an industrial sensor devised to detect PET material, it was the only sensor considered at the beginning of the project but testing results showed certain tendency to error, data read from IR sensors complement the measurements for accurate results.

|  |  |
| --- | --- |
| **Element** | **Name** |
| 1 | Black housing , Sensor holder |
| 2 | E3ZM-B OMRON PET Bottle sensor |
| 3 | Tx/Rx Infra-red LED |
| 4 | Conveyor, Measurement 6cm x45cm |
| 5 | 12V Motor |
| 6 | TM4C123GXL Cortex M4 DevBoard |
| 7 | NXP H-Bridge for DC motor Control |
| 8 | Nokia 5110 Graphical display |

Table 1‑2. Feed stage of recycle machine, components list.

The system manager is embedded in a Cortex M4 development board TM4C123GXL (6) in Fig. 1‑1. It coordinates the tasks for the elements operation such as, (4) conveyor speed, (7) conveyor direction, (8) printing messages on the LCD screen, and sensor (2, 3) calculations.

The system manager is an operative system suited for the selected development board, it has multitasking and preemptive features and follows the requirements mentioned in chapter 2.

# Operative System Requirements

To control the recycling machine, a preemptive kernel based on switch and restore context is designed under next requirements.

Each task shall be associated with a data structure, called a task control block. This data structure contains at least a PC, register contents, an identification string or number, a task status, and a task priority.

The system stores these TCBs in one or more data structures, such as a linked list.

* Task Stack shall be allocated with Memory Allocation interface
* Each task shall contain its own stack including the Background Task
* Stack size will depend on the project memory model

Project shall support event-driven tasks:

* Button Task 1 -> On Board Button 1 ISR (This will be connected to the knives door lock)
* Button Task 2 -> On Board Button 2 ISR (enabled for future capacities)
* Background Task shall toggle onboard green led
* Increment a counter on the event-driven task Timed Task 1
* All tasks shall set high and low a pin level when entering or terminating a task respectively.
* SaveContext and RestoreContext interfaces shall be provided to support Context Switch mechanism.

# Operative System Elements

## Task ID

The name given to a target service or function, the service status is going to be changed to ready and buffered by using this task ID.

## Triggered Tasks, Task activation

Triggered tasks also call the ActivateTask() function but as they are asynchronous, they don’t need a scheduler to be serviced. The READY to RUNNING state change gets the same treatment than in periodic tasks

## Background Task

As a non-interrupt-driven task, the background processing should include anything that is not time critical. The background process is the process with the lowest priority but is the first function and is proceeded by the following steps:

1. Disable interrupts.

2. Set up interrupt vectors and stacks.

3. Perform system initialization.

## Activate Task(TaskType taskID)

This API receives as input the taskID parameter and switches its state from SUSPENDED to READY if no error occurred, it pushes this task into the priority buffer according to task priority. There are two types of tasks, periodic task and triggered task, the first type takes one additional step to be activated (turned to READY state), it calls to Task\_sch\_activate(), whereas the second type just calls ActivateTask(TaskType taskID).

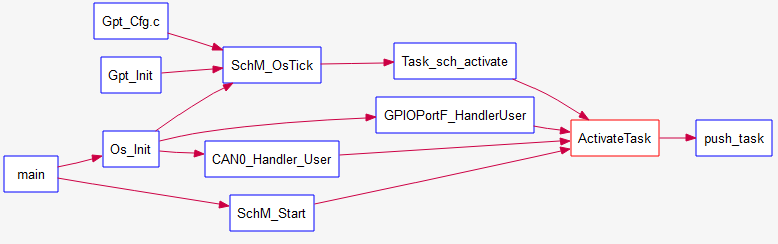


Fig. ‑

## Dispatcher

The dispatcher searches on the priority buffer FIFO and picks up the next task to be executed.

It finds the higher priority and oldest task in READY state to switch its status to RUNNING and returns the pointer to the task to be executed but it doesn’t execute it, that’s the function of restore\_context() explained later in this document.

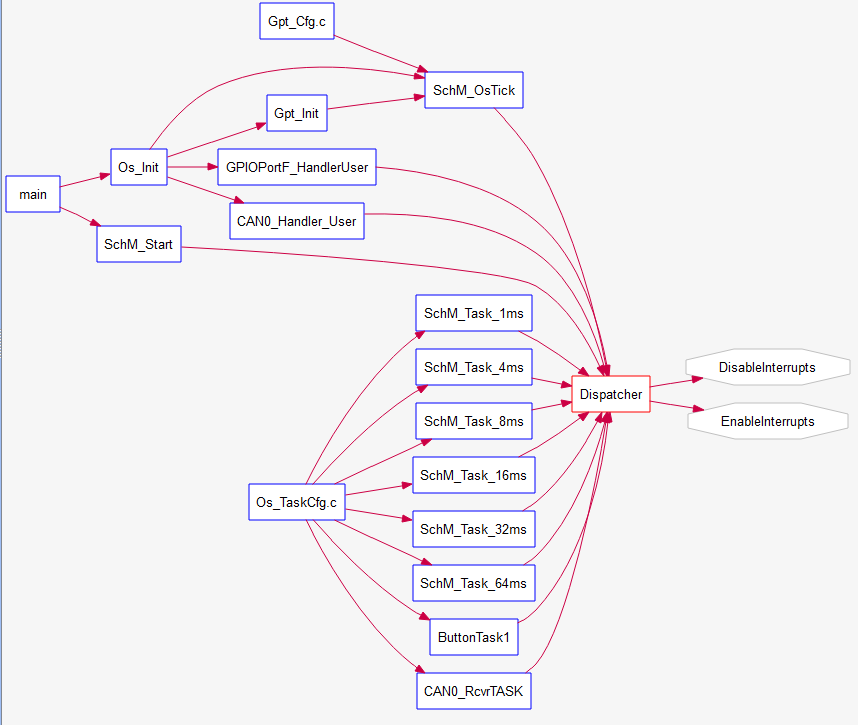


Fig. ‑

Dispatching a task implies its previous activation. Basically, ActivateTask(TASKID) is cyclically running in SchM\_OsTick meaning it shall be possible to interrupt the current task (Task1) to allocate in the execution buffer the next entry.

Y

Status(R)=ready

Pull(R)

Push(R)

Put task on tail

Determine higher priority task

Task2 Priority

> Task1 Priority

Fig. 3‑3 Dispatcher flow diagram, it determines the task to be executed but doesn’t executes it. If Task2 is triggered and has higher priority than the currently running Task1, Task1 is put on tail according its priority and the change context is performed.

## Scheduler Mechanism

In embedded software the processing power is commonly limited, managing the access to CPU resources is crucial. The need of a mechanism based on a counter tick that launches the right task at the right moment in a controlled way is crucial.

|  |  |  |
| --- | --- | --- |
| Os Tick | 500 | us |
| Mask T1 | 63 |  |
| Mask T2 | 15 |  |
| Offset T1 | 0 |  |
| Offset T2 | 3 |  |

Table 3‑1. Example, offset and mask for 32ms and 8ms Task.

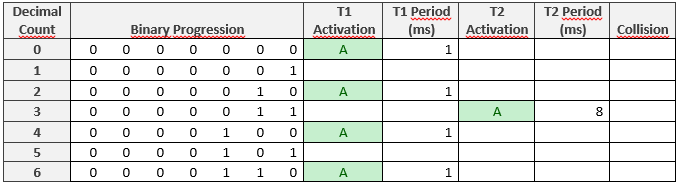


Table 3‑2. Binary progression. ´A´ denotes activation of 32ms and 8ms Task.

In this work, each task control is encoded in a structure called tSchM\_Task\_Descriptor that contains each mask and offset information used in cyclic Binary Progression Scheduler technic to activate the next task as shown in Table 1.2, where Binary Progression column indicates the 7 bit counter that´s incremented by each clock tick. When the binary counter and the mask matches, the task is executed, the rate is calculated by

The ActivateTask() function is called changing to READY the state of the corresponding task, the **Scheduler** sweeps through the priority buffers and picks up the next task to be launched. So this management scheme performs the action that has been planned for this instant. If a task is started, the operating system informs the task of its activation time, which is synchronized within the cluster.

This schedule considers the required precedence and mutual relationships among the tasks such that an explicit coordination of the tasks by the operating system at runtime is not necessary. tSchM\_Task\_Descriptor structure must be allocated in flash memory with the command *const* as declared in file Os\_task\_cfg.c.

## Priority buffer

Buffer arrange contains as many FIFO queues as the number of tasks priorities.

Task Control Block

TaskId;

priority;

ptrTask;

READY;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

READY;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

READY;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

READY;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

READY;

StackInfo;

deadline;

Priority buffers

Low priority

High priority

Scheduler does a Sweep on priority buffer

0

1

N

.

Fig. 3‑4. Graphical representation of scheduler job on priority buffer.

ActivateTask(TaskType taskID) changes the identified task state to READY and pushes it into the corresponding priority buffer. Then, all tasks within priority buffer are in READY state and are waiting for Dispatcher which selects the next task to update its status to RUNNING.

The context level variable on Task control block informs the level of the interrupts context.

typedef struct

{

enum tSchedulerTasks\_ID TaskId;

UINT8 priority;

UINT8 ContextLevel;

tPtr\_to\_function ptrTask;

enum tTaskStates enTaskState;

tStackInformation StackInfo;

tDeadline deadline;

UINT8 StackBuffer[100];

}tTaskControlBlock;

*Table 3‑3. Task Control Block Elements.*

## Priority Buffers and Task Control Block

The priority buffer only saves the TaskID of the tasks in READY state. The priority buffer operates as FIFO such as the first task entering the buffer is the first to be executed.

Task Control Block, Tasks Set to SUSPENDED after initialization

TaskId;

priority;

ptrTask;

SUSPENDED;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

SUSPENDED;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

SUSPENDED;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

SUSPENDED;

StackInfo;

deadline;

TaskId;

priority;

ptrTask;

SUSPENDED;

StackInfo;

deadline;

Low priority

High priority

0

1

After Activation, each Tasks is set to READY and queued in a priority buffer

N

Priority buffers

Fig. 3‑5. Representation of Task Control Block.

Figure1. Shows the structure of the buffers interrupt and TaskControl Block structure, which are located in RAM and dynamically allocated,

## Task private variables (memory allocation)

Use of memory allocation functions makes Task variables private to other modules, Allocates memory for internal status and control structures. Function Os\_Init () reserve memory for ISR and status:

(3‑1)

## Restore Context

As described in chapter 3.3, the dispatcher selects which is the following task by loading the static pointer with the address of the next function to be executed and updates its status to RUNNING, restore\_context() function shifts the stack for the recently updated RUNNING task and POPS data to run it.

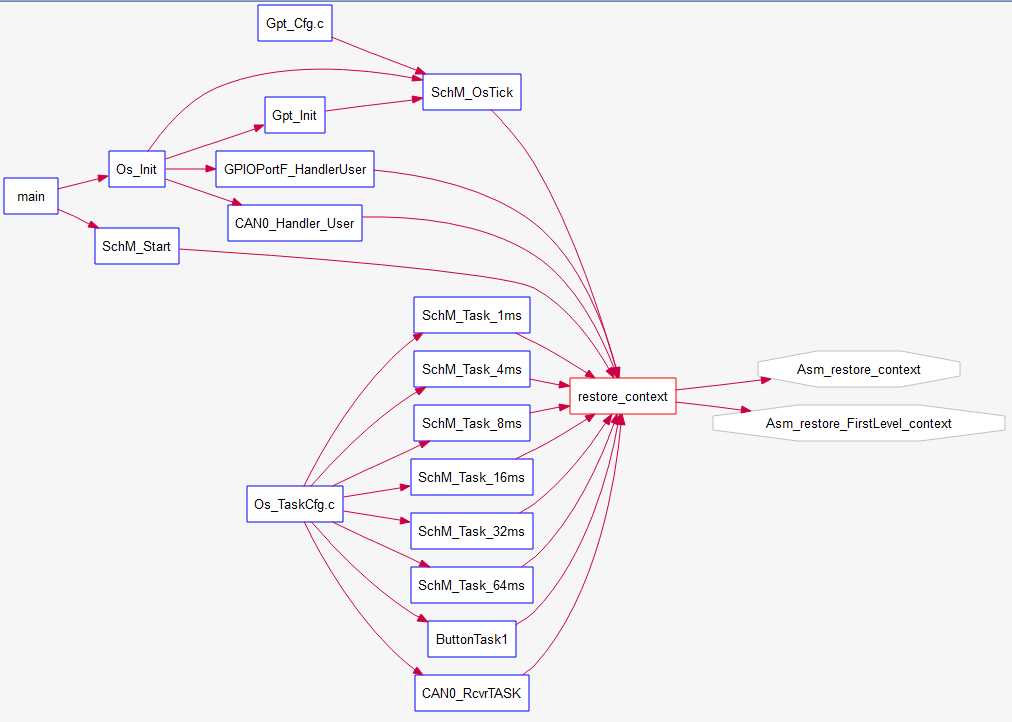


Fig. 3‑6. Arrows indicate periodic tasks and OsTick are calling restore\_context.

Relationship between periodically activated tasks and restore\_context function is depicted in Fig. 3‑7.

Asm\_restore\_context

CPSID I

LDR R2,=newSP

LDR sp, [R2] ; new thread SP = newSP

POP {R4-R11} ; restore regs r4-11

POP {R0-R3} ; restore regs r0-3

LDR R2,=EndSP

LDR sp, [R2] ; new thread SP = newSP

POP {R12}

POP {LR} ; discard LR from initial stack

CPSIE I ; Enable interrupts at processor level

BX LR

Table 3‑4 Code segment to restore context

The code shown in Table 1.3 loads the pointed task addressed by Dispatcher and updates the register values that correspond to it. There’s where context switch occurs.

## Task state

A task typically can be in any one of the four following states:

1. Running

2. Ready

3. Suspended (or blocked)

In a single-processing system, there can be only one task running. A task can enter the running state when it is created (if no other tasks are ready), or from the ready state (if it is eligible to run based on its priority or its position in the scheduler list). When a task is completed it returns to the suspended state. Tasks in the ready state are those that are ready to run but are not running. A running task enters the ready state if it was executing and its time slice runs out, or it was preempted. If it was in the suspended state, then it can enter the ready state if an event that initiates it occurs.

There are two types of tasks, periodic task and triggered task, the task type is defined in a variable within task descriptor structure explained below. The main difference between them is that the firs type takes one additional step to be activated (turned to READY state), it calls to Task\_sch\_activate(), whereas the second type just calls ActivateTask(TaskType taskID).

In the case of periodic tasks, the SUSPENDED to READY state switch is managed by the Binary Progression Scheduler within Task\_sch\_activate(). A mechanism based on a counter tick, a task mask and offset to manage the access to the CPU resources by launching the right task at the right moment in a controlled manner.

## Calls from tasks

In OSEK, a task can only terminate itself, each tasks shall terminate itself at the end of its code. Then, three functions must be called at the end of overall tasks, these functions are described below.

1. statusOS = TerminateTask() ;

2. Dispatcher();

3. restore\_context();

## Tasks Termination

Both types of tasks terminate themself by calling TerminateTask(). This function terminates the task transferring it from RUNNING to SUSPENDED state and the operating system makes the results of the task available to other tasks and informs whether the function terminated normally “E\_OK” or wrongly mean “E\_OS\_LIMIT. Both results will be saved in the result variable statusOS (statusOS = TerminateTask()).

## TASK stack

The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

Any time an interrupt runs in, we need to perform a context switch to store all information running during foreground, complete background and save its results and return to foreground thread recovering the information previously stored.

Managing the stack during context switch could result a complex task, especially when the microcontroller automatically saves 8 Registers. The Integrator should design a strategy to keep safe each piece of task information while microcontroller handles the registers in the designated stack.

Each task has its own stack area on memory such as shown on Fig. 3‑9. These stacks have enough space no save N time the context of their own context plus local variables and interrupts of class 1.

Stack Task 1

SP1->

Stack Task 2

SP2->

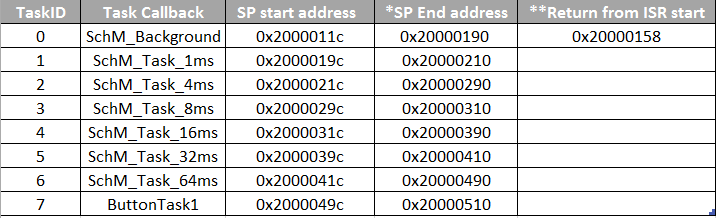
Stack Task N

SPN->

*Fig. 3‑9 Stack pointer model, after initialization all SP are pointing to the lower address of the* *task assigned memory known as “thumb bit”.*

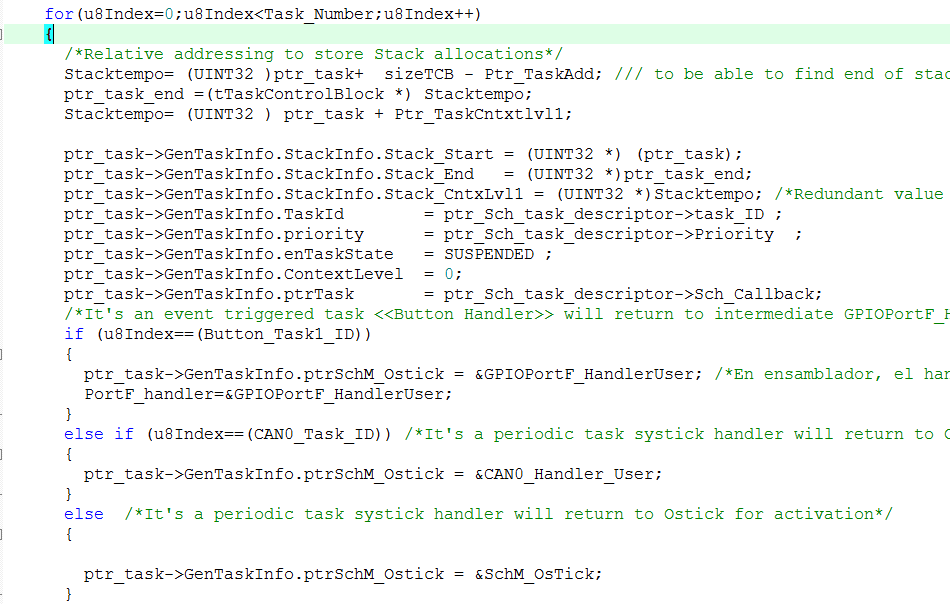
The task control block of each task stores the last address of SP inside Stack info, so SP is configured with Stack info by using Os\_Init function.

## Stack distribution per Task



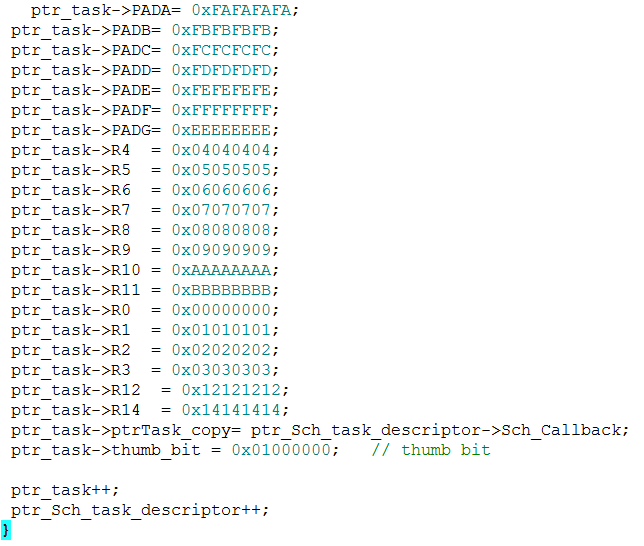
*Table2. Stack Distribution per TASK*

Below, one of the key parts in Os\_Init (), this function is called once in main program after power on.



Task information as declared at:

tTaskControlBlock->tTaskInfo



Padding to guard some memory between

TaskInfo and Registers information.

In the end, Stack pointer points to the last element of this array “thumb bit” for each task.

Motivation for this padding. Acts like memory guard; In case that more information is added in tTaskInfo, the current size of the stack is not increased by Heap allocation so that performance is not compromised. It also promotes a more readable stack.

I´ll describe the strategy followed to return from an interrupt successfully.

## Initialization

Initialization is the stage where the buffer size is defined, Os\_init() is the function where Start and End address task by task are declared and is done by the next couple of commands.

Interrupts are enabled after initialization and once background task was started.

Initialization is actually the first part of the background process. It is important to disable interrupts because many systems startup with interrupts enabled while time is still needed to set things up. This setup consists of initializing the appropriate interrupt vector addresses, setting up stacks if it is a multiple-level interrupt system, and initializing any data, counters, arrays, and so on.

## Context switch

Once “SP End address” is calculated, this value is loaded into SP during restore\_context() and before running into the ISR handler.

Below is an example for stack pointer in RAM.

While running a task, SP points to the lower address of the assigned task memory or “thumb bit” as shown in Fig. 3‑9this memory is assigned during initialization time. When an interrupt is asserted, Cortex M architecture PUSHES {PSR, PC, LR, R12, R3, R2, R1, and R0} automatically by hardware while the entering interrupt handler PUSHES {R4-R11} while pushing into the SP, its address is decrement automatically.

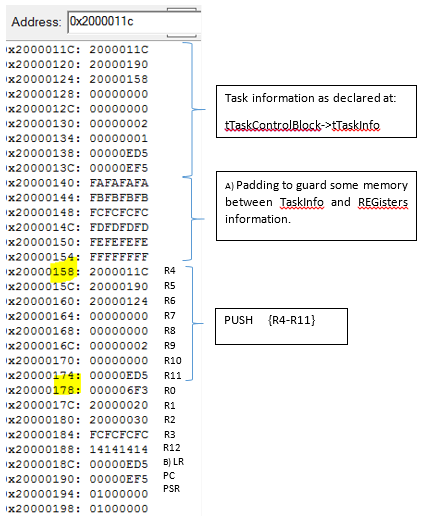
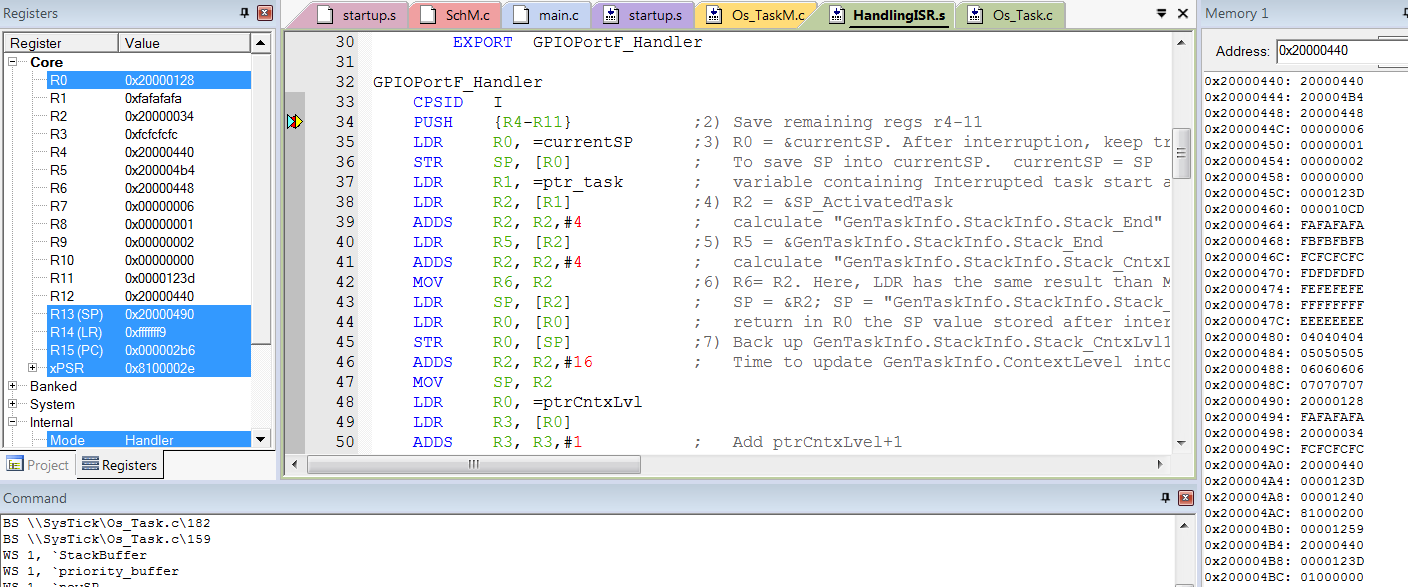


Fig. 3‑11

Secondly, during ISR handler the microprocessor pushes next registers automatically:

PSR, PC, LR, R12, R3, R2, R1, and R0, e.g. Next registers are automatically saved during PortF interruption.

By using “SP End address” as initial base address and decreasing “SP address - 4” before storing next register; finally “Return from ISR start” value is known after doing PUSH {R4-R11}.



R0

R1

R2

R3

R12

B) LR

PC

PSR

In this way the RAM area between “Return from ISR start” and “SP End address” enclosed all information needed to return from interrupt and perform “ISR\_User” callout.

It means that before executing BX LR we shall set SP in ”Return from ISR start” where all the information to execute “ISR\_User” shall be complete and organized, some overhead is added in each ISR\_Handler to make it possible. Extra information about this overhead is provided in B).

1. LR. PC and PSR values are updated within ISR, these values are copied from TASKINFO-> Sch\_Callback structure, originally defined within Os\_Init().

## Complementary information about ISR behavior

When an interrupt is processed the current instruction is finished; registers R0-R3, R12, LR, PC, and PSR are pushed automatically; LR is set to 0xFFFFFFF9; IPSR is set to the interrupt number being processed; PC is set with interrupt vector address. The last three steps can occur in any order.

LR is set to 0xFFFFFFF9 this pattern indicates we´re running an ISR meaning the return to main program points an allocation loaded in the 8 registers previously stored when ISR was called. This pattern avoid jumping nowhere once the handler is finished (To POP 8 Registers including the PC once BX LR is called at the end of ISR).

R4-R11 are not pushed on the stack automatically, in theory when the programmer writes an ISR he will not use R4-R11, so it´s not needed to backup them.

Five conditions must be true for an interrupt to be generated:

1) Device arm.

2) NVIC enable.

3) Global enable.

4) Interrupt priority level must be higher than current level executing.

5) Hardware event trigger.

An interrupt causes the following sequence of five events.

1) Current instruction is finished.

2) Suspend, makes a record of where we were. Eight registers are automatically pushed on the stack, the SP points to the top of stack.

3) LR is set to 0xFFFFFFF9.

4) IPSR is set to the interrupt number.

5) PC is loaded with the interrupt vector.

These five steps, called a context switch, occur automatically in hardware as the context is switched from a foreground thread to a background thread. We can also have a context switch from a lower priority ISR to a higher priority ISR. Next, the software executes the ISR.

Clearing a trigger flag is called acknowledgment, which occurs only by specific software action.

Each trigger flag has a specific action software must perform to clear that flag. We will pay special attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgment. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The third aspect that the software controls is the interrupt enable bit. Specifically, bit 0 of the special register PRIMASK is the interrupt mask bit, I. If this bit is 1 most interrupts and exceptions are not allowed, which we will define as disabled. If the bit is 0, then interrupts are allowed, which we will define as enabled.

The design of the interrupt service routine requires careful consideration of many factors. Except for the SysTick interrupt, the ISR software must explicitly clear the trigger flag that caused the interrupt (acknowledge). After the ISR provides the necessary service, it will execute BX LR. Because LR contains a special value (e.g., 0xFFFFFFF9), this instruction pops the 8 registers from the stack, which returns control to the main program. If the LR is 0xFFFFFFE9, then 26 registers (R0-R3, R12, LR, PC, PSW, and 18 floating point registers) will be popped by BX LR.

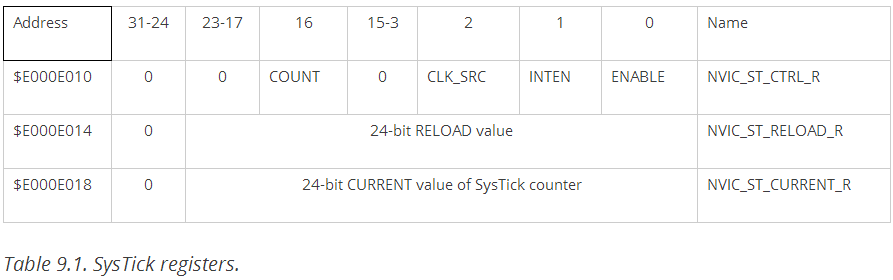
An alternative to multiple lists involves a single list in which only the status variable in the TCB is modified rather than moving the block. Thus, for example, when a task is moved from suspended to ready state, or from the ready to executing state, only the status word is changed.

This approach has the advantage of less list management but slower traversal times, since the entire list must be traversed during each context switch to identify the next highest priority task that is ready to run.

The main drawback of the task-control block model is that when a large number of tasks are created, the overhead of the scheduler can become significant. The operating system manages the TCBs by keeping track of the status or state of each task.

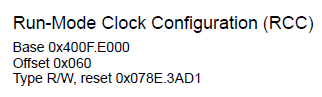
## SysTick

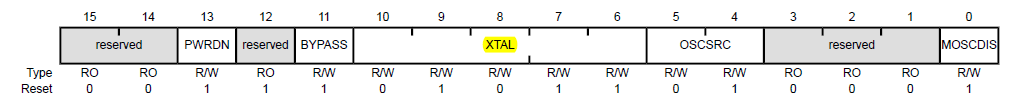
SysTick is a simple counter that we can use to create time delays and generate periodic interrupts. It exists on all Cortex-M microcontrollers, so using SysTick means the system will be easy to port to other microcontrollers. Table 9.1 shows the register definitions



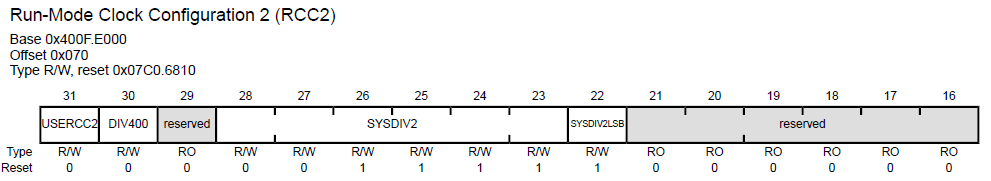
The PLL can take advantage of the external oscillator to speed up or slow down the clock, speeding up is going to make the clock faster, doing more work but consuming energy faster. If we require longer battery life we may need to reduce the CPU speed slowing down the clock.

Write XTAL= 10101 or 0x15 in RCC register to configure 16Mz Xtal frequency as shown in TM4C123GH6PM.PDF table 24-14.





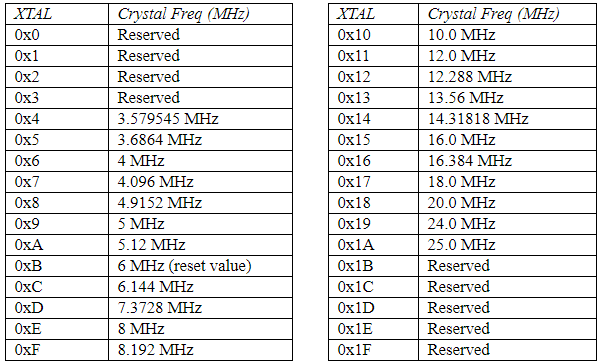
Write ´n´ number in SYSDIV2 of RCC2 register to get n+1 divisor



Finally, we will get PLL nominal frequency (400MHz) divided by n+1.

## PLL Initialization steps

This initialization is done at the very beginning during main function [3], in this project it’s wrongly coded into *Mcu* function but corrected later to *PLL* function.



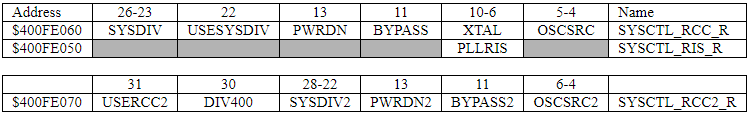


Table 3‑5 RCC2 (PLL registers) in TM4C microcontroller.

Steps to configure the PLL to operate at 16MHz Xtal Oscillator at 80MHz [7].

0) Use RCC2 register for this configuration.

1) To set BYPASS2, after this step the PLL is avoided.

2) To set the crystal frequency in the four (10-6) bits in Table 3‑6 using XTAL table for desired configuration. To clear OSCSRC2 bits, this action selects the main oscillator as clock source.

3) To clear PWRDN2, this activates the PLL.

4) To configure and enable the clock divider. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place n = 4 into the SYSDIV2 field because this value divides the clock by n+1.

5) To wait SYS\_R to become high, this is controlled by the processor and indicates the PLL is stabilized.

6) To rollback step 1) by re-connecting the PLL clearing the BYPASS2 bit.

# Serial drivers configuration

SSI (Nokia Display) driver is developed by following next requirements:

* Driver shall take a unique configuration and configure all channels as per their corresponding static configuration
* Require, upon initialization, dynamically allocated RAM for their TX and RX buffers
* Require, upon initialization, dynamically allocated RAM for their internal status structures

## Serial ports configuration file

Application configure all channels on a single file Protocol\_cnf.c, number of channels and configuration per channel

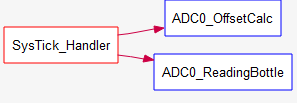
Serial configuration is based on pointer to their registers, this allow to remove switch-case sentences on the Serial init

tProtocol\_Port\_ptr  Protocol\_Port**[]** **=**  **{**    Protocol0\_BASE**,**  
                                  Protocol1\_BASE**,**  
                                  Protocol2\_BASE**,**  
                                  Protocol3\_BASE**,**  
                                  Protocol4\_BASE**,**  
                                  Protocol5\_BASE**,**  
                                  Protocol6\_BASE**,**  
                                  Protocol7\_BASE **};**

# Photoelectric Sensors

## Infra-red pairs

Designing a low cost solution is crucial to produce an affordable machine, part of the solution is to implement Infra-red sensors stage devised as four transmitter-receiver pairs element connected to an ADC as shown in figure 1. The ADC module is configured to work. It´s attended by SysTick\_Handler interruption every 25ms. Just once after energizing the prototype an offset calculation takes place for calibration purposes; ADC\_ReadSensorOffset (return p\_medicion.u16Offset\_Buffer) is called to calculate the implied error in ADC lecture as Infra-Red led is analogous and prone to noise, an array of size MEAN\_BUFFER\_SIZE (16 elements) containing consecutive measures per Channel, calculates the amount of noise in Volts along 4 Channels sweep.



The function ADC\_ReadSensorOffset reads the instant value and calculated in (5‑1)

(5‑1)

In the end, the noise amount converted in Volts is subtracted from the mean voltage calculated later in ADC0\_ReadingBottle, where another structure pmedicion.u16Mean calculates the Mean voltage after scanning the bottle cylindrical part in Formula 2:

Formula 2

The usage of Offset calculation comes later in Formula 3:

Formula 3

It takes 23 ms to be fully executed, this time involves waiting till ADC Hardware triggers its end of conversion flag as follows:

// 2) wait for conversion done

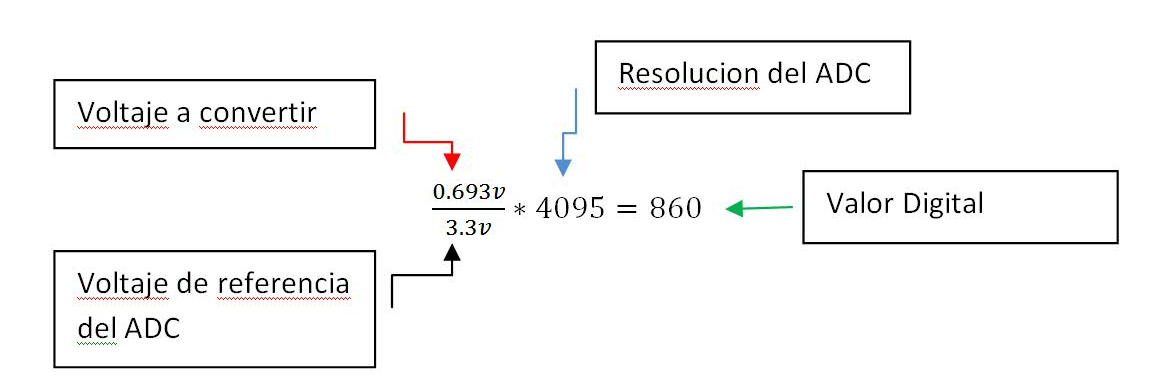
The challenge is to integrate this change into the operative system, where Systick\_Handler was replaced by a TimerA0 interrupt and the IR mean voltage calculation was enabled just once a bottle was over the feeding band.

Below is the math to convert analog voltage into digital value:

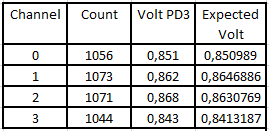
ADC Reference Voltage

Voltage to convert

Digital Value



ADC Resolution



**Table 1.8.1 Comparison real Voltage (VoltPD3) vs. Expected Voltage (Volt)**

## OMRON sensor.

The sensor is allocated at the end of the conveyor as shown in figure X the part number is E3ZM-B PET Bottle Detection.



E3ZM B

Reflector

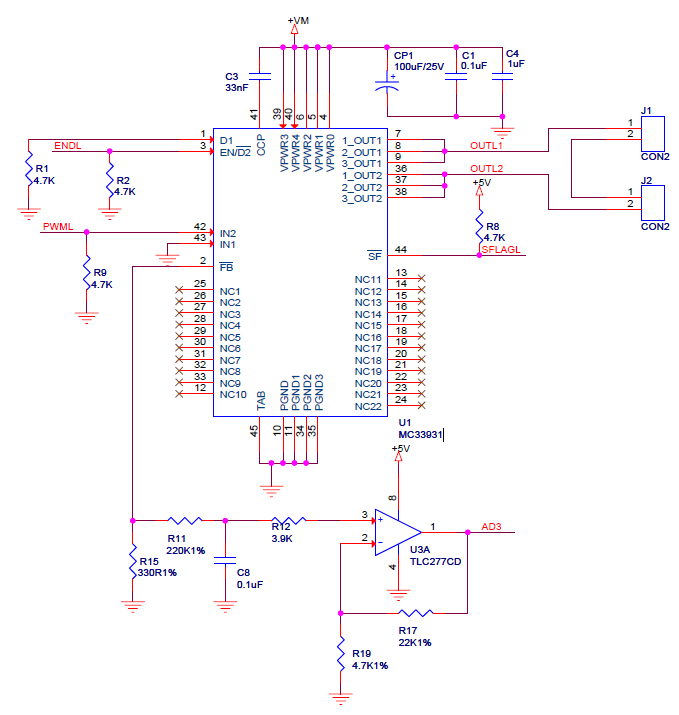
Receptor

**Figure 1.8.2 Omron sensor-reflector polarizes with PET bottle.**

As shown in figure 1.8.2, It requires a reflective surface mounted on the frontal surface, this surface is a polarization filter that is tuned only with the PET bottle refraction reducing loss. The sensor output is 5V when a PET bottle is detected or 0V for other different materials.

# Measurements

The conveyor is the first contact to feed the machine, a DC motor shall push inside the sensor box (piece 1 in fig XX ) First two rows in truth table shown in Appendix A to control EN/! D2, then OUT1/2 states will be defined by controlling IN1 and IN2 states.

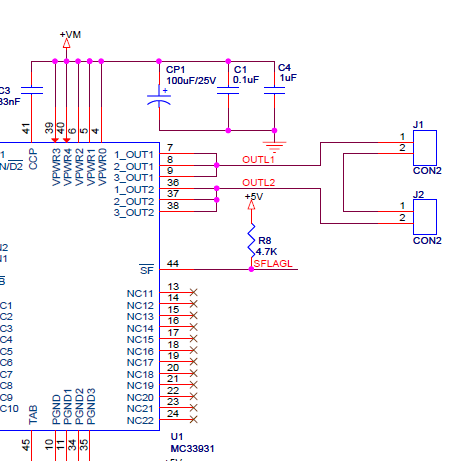


If you look at the schematic, you can see that IN1 is directly connected to GND. In order to enable breaking you need to lift pin 43, solder a wire to it and control it properly. [6]

Below is shown the board connection for MC33931:

U1 connected to Motor

Power Source J4 pin 1 (+ 9V), pin 2 ( - ) , current consumption = 360mA



+M

-M

## Feeding machine, bottle to Inside

Terminal ENDL (JP2 pin1) +5V connected,

Signal connected to PWML (JP2 pin 15)

T = 1ms

Duty cycle = 100us



Returning Bottle to outside

Terminal ENDL (JP2 pin1) GND connected,

Signal connected to PWML (JP2 pin 15)

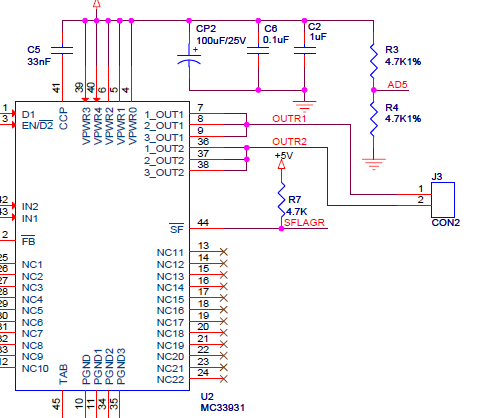
T = 1ms

Dutycycle = 900us

****

U2 Connected to Motor

Source power J4 pin 1 (+ 9V), pin 2 ( - ) , current consumption = 360mA

****

-M

+M

## Feeding machine, bottle to Inside

Terminal ENDR (JP2 pin2) +5V connected,

Signal connected to PWML (JP2 pin 16)

T = 1ms

Dutycycle = 100us

****

## Returning Bottle to outside

Terminal ENDR (JP2 pin2) +5V connected,

Signal connected to PWML (JP2 pin 16)

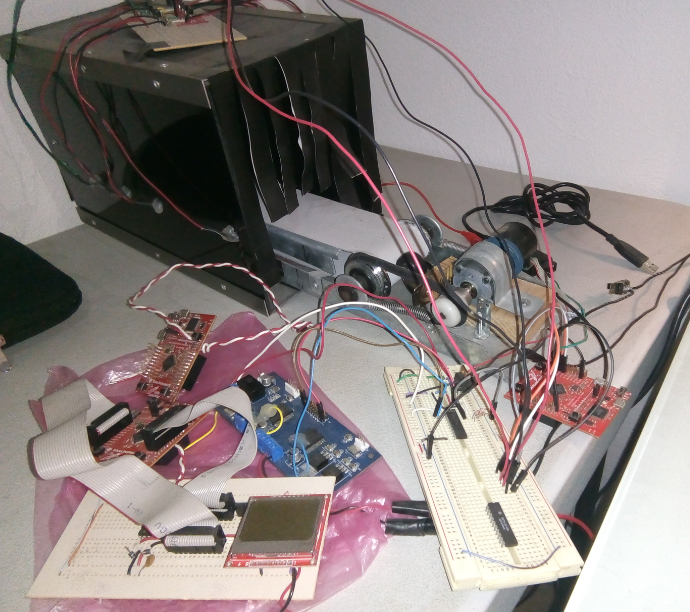
T = 1ms

Dutycycle = 900us

****

# Prototype

The prototype was assembled in parallel the SW development.



# Conclusion

The execution of the main program is called the foreground thread, and the executions of the various

Interrupt service routines are called background threads.

This operative system is crashing when an ISR is called even if former ISR is not finished yet, this is during the context switch where the stack pointer is automatically addressing to the calculated address plus (+1 or +2) randomly which made impossible to predict where the program counter has to continue executing the program.

This is mitigated in SW version “RecyPET\_V1p6\_with\_CANTx\_V1p20asc1p2” by applying the following technic:

Task for Button1 is implemented, enabling asynchronous interruptions that could be triggered the micro does again this when automatically saves an extra data when entering the synch interrupt, I think it is related to the change when creating the new function. To solve the effect of that extra data, you have to cycle for the next pseudo code. It is necessary to shift or shift the data in the stack, taking as reference the address in SysTick\_Handler (step 3. R0 = & currentSP after interruption).

Description of variables in pseudo code

o1; content in & currentSP, the first time takes the value in the address after interruption, then takes the value the address in turn.

o2; content in & currentSP, same as o1 for address in turn +1.

x1; Generic name for Register1 that is free and that will be used

x2; same as x1 for a second record.

Increases; refers to value of current\_SP + 4.

save o1 in x1>

Increment moves to o2 to open macro

save o2 in x2

paste x1 on o2>

Increment moves to o3

save o3 in x1 \_\_\_\_ 1st cycle, save x1

paste x2 on o3 paste x2

Increment moves to o4

save o4 in x2 ++++ 2nd cycle, save x2

paste x1 in o4 paste x1

Increment moves to o5

save o5 in x1 \_\_\_\_ 1st cycle

paste x2 on o5

Increment moves to o6

save o6 in x2 ++++ 2nd cycle

paste x1 on o6

Increment moves to o7

save o7 in x1 \_\_\_\_1rst cycle

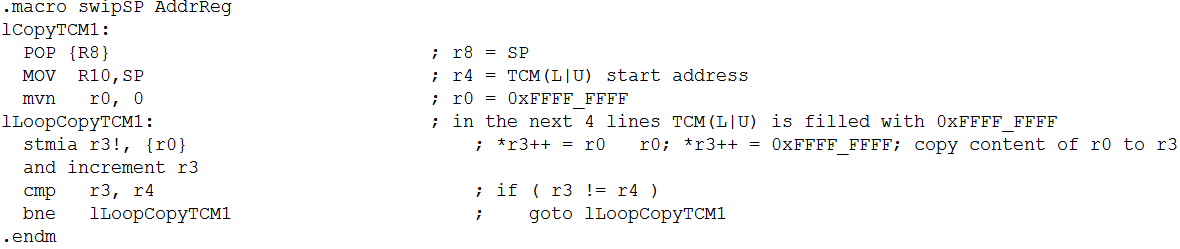
paste x2 on o7

Increment moves to o8

save o8 in x2 ++++ 2nd cycle

paste x1 on o8

... and so on, down to the lower value of the task memory (or thumb bit 0x10000000)



It would be to implement a macro for the shift of data offset when the BX LR instruction is executed, bits 31-1 of register LR are put back into the PC, and bit 0 of LR goes into the T bit. On the ARM Cortex-M processor, the T bit should always be 1, meaning the processor is always in the Thumb state. Normally, the proper value of bit 0 is assigned automatically.

+ I do not have WDOG\_feed.

*Errors solved during development:*

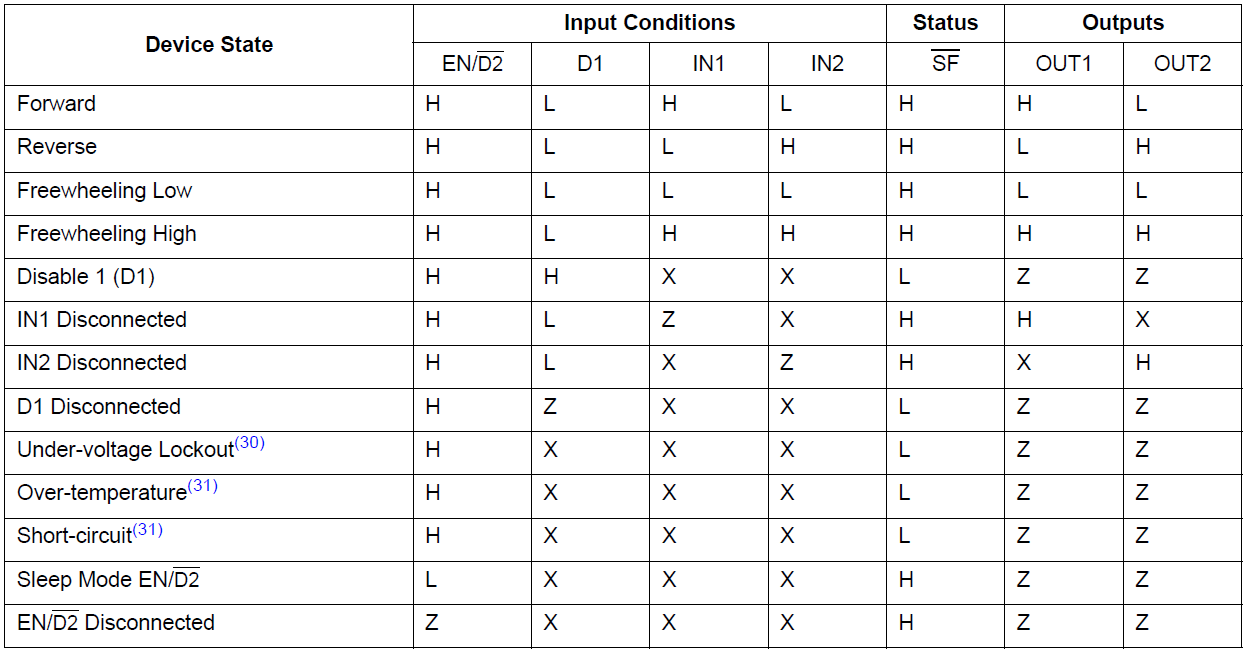
The system will crash if the interrupt service routine doesn’t either acknowledge or disarm the device requesting the interrupt. The ISR software should not disable interrupts at the beginning nor should it re-enable interrupts at the end. Which interrupts are allowed to run is automatically controlled by the priority set in the NVIC.

This board was selected by its price and its power, this board allows floating point operations required in (2‑1).

Appendix

A. Conveyor, H Bridge Control

The tri-state conditions and the status flag are reset using D1 or EN/! D2.



The EN/!D2 pin performs the same function as D1 pin when it goes to a logic LOW the outputs are immediately tri-stated.

# Bibliografía

[1] Phillip A. Laplante, *Real Time Systems Design and Analysis*, IEEE Press pp. 73-93, Sept. 2003.

[2] Hermann Kopetz, *Real Time Systems Design Principles for Distributed Embedded Applications”,* Springers pp. 183-186, January 2011

[3] J.W. Valvano, [*Embedded Systems: Introduction to ARM Cortex M Microcontrollers*](http://users.ece.utexas.edu/~valvano/arm/outline1.htm),   5th Ed. Vol1, pp.258-260, pp.280-282, pp. 326-329, Nov. 2003.

[4] J.W. Valvano, [*Embedded Systems: Real-Time Interfacing to ARM Cortex M Microcontrollers*](http://users.ece.utexas.edu/~valvano/arm/outline.htm), 2nd

Ed. Vol3, pp.119-126, Jan. 2014.

[5] ARM Technical Staff, *ARM Compiler Toolchain Assembler Reference*, Ver. 5.0

[6] NXP Technical Staff, *MOTOR\_DRIVE\_REVA*. Aug.2011. [Online]. Available: <https://community.freescale.com/docs/DOC-1019>.

[7] Texas Instrument Technical Staff, *TM4C123GH6PM Microcontroller.* July 2013. [Online]. Available: http://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf

[8] Texas Instrument Technical Staff, *TM4C123GH6PM Microcontroller.* July 2013. [Online]. Available: <http://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf>

[9] “Cortex M3 application with CAN driver”, Notes for UT.6.10x Embedded Systems, University of Texas at Austin, Jun. 2017.

# [10] El Financiero “México, líder en reciclaje de PET”, Rogelio Varela. Sept. 2014. [Online]. Available:

# http://www.elfinanciero.com.mx/opinion/rogelio-varela/mexico-lider-en-reciclaje-de-pet