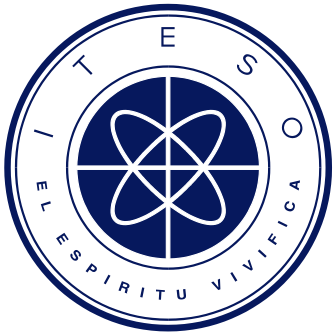
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

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 AN EMBEDDED CONTROL FOR A RECYCLING MACHINE

Tesina para obtener el grado de:

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presentan: Antonio Rodríguez Soto

Director: Dr. Luis Rizo Domínguez

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

ACKNOWLEDGEMENTS

My special appreciations to my parents Rosa y Antonio and my wife Jacqueline who helped me plan this career since the beginning and provided me with enough tools and inspiration along the way.

To MBA. Ricardo García, Ing. Diego Haro, and Ph.D. Joel Chavoya, they paved the way that allowed me to take off in the embedded world.

Many thanks also to my teachers, MSE. Abraham Tezmol and MSE. Luis Puebla, for your compromise and dedication. I would like to continue learning from you.

Thanks to my teachers and friends, Ph.D. Luis Rizo Dominguez and Ph.D. Lorena Michele Brennan Bourdon, who always look forward to selflessly help others to improve.

Abstract

México is recycling around 60 % of all PET bottles consumed in its internal market and is the leader in food-grade recycled PET. México leads this industry above the USA and Canada, and its contribution is not just economical but more so, a measure to improve our environment. The PET recycling industry faces the challenge of the inefficient process to recover a PET bottle from waste. The aim of this work is to facilitate the recovery process of a PET bottle for recycling by designing and implementing an affordable recycling machine. There are two key components devised in this work to accomplish this goal: the infrared sensor stage and the operative system; both components are embedded into the same low power microcontroller from the Cortex M4 family. The sensor stage is based on infrared pairs connected to the microcontroller ADC input, a self-calibration sequence calculates the necessary offset to compensate for the hardware bias. This calibration takes place one time during initialization. Several tests were performed to tune the sensor output so that it could be used as a threshold level to distinguish a PET bottle. A second sensor used is the E3ZM-B OMRON, this device reinforces the sensor stage as the output is connected to the microcontroller input, which determines if a PET bottle is on the conveyor. The second key component of this work is the operative system, a multi task, and fully preemptive system is embedded into the microcontroller, it manages the machine operations by means of tasks processed depending on its priorities, a high priority task can interrupt a lower priority task, this is known as preemption which is achieved by a context switch. In this work, the context switch lead to a not deterministic microcontroller behavior and the PC register was loaded with the wrong return address needed to continue with the program execution. As a result, the periodical tasks were processed correctly but they should last no longer than 1ms, as it is not possible to preempt them. On the other hand, the sensor stage worked as expected with the sole condition of avoiding a beam of light.

.

Resumen

México recicla alrededor del 60% de todas las botellas de PET que se consumen en su mercado interno y es el líder en PET reciclado de grado alimenticio. México encabeza la industria por encima de EE. UU. Y Canadá. La contribución no solo es económica sino que también es una medida para mejorar nuestro medio ambiente. La industria de reciclaje de PET se enfrenta al desafío del ineficiente proceso para recuperar una botella de PET de los residuos. El objetivo de este trabajo es facilitar el proceso de recuperación de una botella de PET para su reciclaje mediante el diseño e implementación de una máquina de reciclaje asequible. Hay dos componentes clave diseñados en este trabajo para lograr este objetivo, estos son la etapa del sensor infrarrojo y el sistema operativo. Ambos componentes están integrados en el mismo microprocesador, se eligió a la familia Cortex M4 por su baja potencia de consumo. La etapa del sensor se basa en pares de infrarrojos conectados a la entrada ADC del microprocesador, para que la secuencia de auto calibración calcule el nivel de compensación necesaria para corregir la desviación del voltaje inherente a los defectos del hardware. Esta calibración tiene lugar solo una vez durante la inicialización. Se realizaron varias pruebas para ajustar la salida del sensor de modo que se pueda utilizar como un nivel de umbral para distinguir una botella de PET; el proceso de exploración lleva 32 ms por cada par de sensores infrarrojos. Un segundo sensor E3ZM-B OMRON es utilizado, este sensor refuerza la etapa del sensado ya que la salida está conectada a la entrada del micro controlador quién determina si hay una botella de PET en la banda transportadora. El segundo componente clave de este trabajo es el sistema operativo, un sistema multitarea y preventivo diseñado para este micro controlador administra las operaciones de la máquina mediante tareas que son procesadas según sus prioridades, una tarea de alta prioridad puede interrumpir una prioridad menor prioridad, esto se logra mediante un cambio de contexto. En este trabajo, el cambio de contexto conduce a un comportamiento no determinista del micro controlador, el registro PC se carga con la dirección de retorno incorrecta para continuar con la ejecución del programa, como resultado, sólo las tareas periódicas se procesan correctamente, pero no deben durar más que 1ms ya que no es posible realizar el cambio de contexto. Por otro lado, la etapa del sensor funciona como se espera con la única condición de evitar al máximo los rayos de luz externos.

List of figures

List of Tables

List of acronyms and abbreviations

API	Application program interface
BPS	Binary Progression Scheduler
FIFO	First input First output buffer
TT	Trigger task.
TADL	Task descriptor list.
RTOS	Real Time Operative System
OS	Operative System
Hz	Hertz
KHz	Kilo Hertz
MHz	Mega Hertz
XTAL	Crystal
SP	Stack Pointer
LR	Link Register
PSR	Program Status Register
PC	Program Counter
ISR	Interrupt Service Register
PLL	Phase Locked Loop
RCC2	Run Mode Clock Configuration
TCB	Task Control Block
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles

Table of contents

Introduction

Recycling PET is a measure to improve our environment. In 2014, México recycled 2 million bottles, equivalent to 6.6 million tons and thus, less CO2 expelled into the environment [10]. However, this recycling rate represents just the 58% of the local consumption [13], meaning that Mexico is still a the early stages on this topic, the main problem in the PET recycling process is the inefficient means to recover a PET bottle from the waste [13]. The purpose of this work is to design and implement an affordable PET bottle recycling machine which could be installed in a collection center. In Chapter 1 is shown the prototype of the machine feeding stage.

The challenge comes with Chapter 2 and 3 the larger chapters in this work since intermediate and advanced programming skills ought to be applied for a scalable and real life solution. This machine will shred plastic by means of iron scissors spinning at a high speed and safety is priority. Furthermore, the machine operations must be controlled by a preemptive and deterministic control, a multi-task fully preemptive system is suited to be running on a low cost/low power *microcontroller* using free KEIL compiler tools for its design [9].

Chapters 4 and 5 detail the proposed algorithm that allows the use 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. A second sensor for industrial applications is used to accurately distinguish between glass and PET bottle. The sensor output is 5V when a PET bottle is detected or 0V for other different materials. It is the E3ZM-B OMRON PET Bottle sensor [11], the only requirement is to block the sensor stage from any beam of light.

The final prototype shown in Figure 6-1, was implemented by one engineer and the goal to implement an affordable bottle recycle machine was tackled in this thesis.

1. Recycling machine description

A recycling machine is a system build around a conveyor devised to identify if the introduced bottle can be recycled. A user deposits the PET bottle on this conveyor, which transports the bottle in front of a bar code reader or scanner. The conveyor spins the bottle until the barcode printed on its label is read by a scanner, then the data from the bottle label is processed and allocated in a database. The recycling machine performs the lookup task and takes the decision whether the bottle is in the database or not, this sort of test determines if the bottle is approved to be recycled. If the test is passed, the bottle is sent to a post-processing stage where the bottle is transformed in PET flakes. This work is focused on developing a prototype of the entrance stage of the recycling machine, this includes a PET sensor and infrared scanner controlled by a multitask OS. The recycling machine prototype is depicted in Figure 1-1, and the machine elements are listed in Table 1-1.

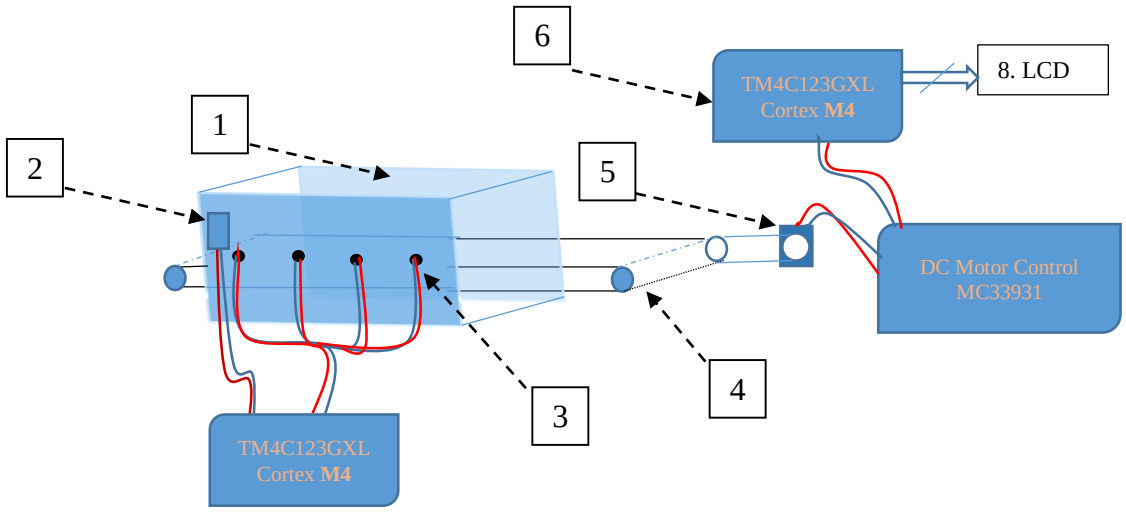


Figure 1-1. Recycling machine Block Diagram and element identification.

One of the requirements to build the recycling machine is to make it as inexpensive as possible, this is achieved by reducing the cost of the PET sensor stage by using the set of paired IR sensors [3] shown in Figure 1-1 and compensate the bias offset by using *(1-1).*

On the left side of Figure 1-1, the sensor [2] completes the sensor stage. It is an industrial sensor devised to detect PET material. It was the only sensor considered at the beginning of the project but testing results showed a certain tendency to error, and the data read from IR sensors complement the measurements for accurate results.

Element	Name
1	Black housing , the 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 1-1. The list of components from the feeding stage of the recycling machine.

The system manager is embedded in a Cortex M4 development board TM4C123GXL element [6] shown in Figure 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.

1.1. Operative System Requirements

To control the recycling machine, a preemptive kernel based on a switch and restore context was designed with the following requirements [5].

Each task shall be associated with a data structure, called a Task Control Block *TCB*. This data structure contains at least a PC, register contents, an identification string or number, a task status, and a task priority [2].

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

- Task Stack shall be allocated with Memory Allocation interface [3].
- Each task shall contain its own stack including the Background Task
- Stack size will depend on the project memory model

The 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 a high and low pin level when entering or terminating a task respectively.
- SaveContext and RestoreContext interfaces shall be provided to support Context Switch mechanism.

2. Operative System Elements

2.1. Task ID

It is the given name to a target service or function. The task is going to be identified with this name along the OS system. In other words, there is no other task with the same name.

2.2. Triggered Tasks, (Task activation)

Triggered tasks, also called the ActivateTask() function, however, these are asynchronous, they do not need a scheduler to be serviced. The READY to RUNNING state change receives the same treatment than in periodic tasks

2.1. Background Task

The background task is the process with the lowest priority. Nonetheless, it is the first process to run in the OS as a non-interrupt-driven task [12]. The background processing should include anything that is not time critical and is proceeded by the following steps:

1. Disable interrupts.

2. Set up interrupt vectors and stacks.

3. Perform system initialization.

2.2. 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 its task priority. There are two types of tasks, periodic task and triggered task, Figure 2-1 shows how the first type takes one additional step in order to be activated (turned to READY state), it calls to Task_sch_activate(), whereas, the second type just calls ActivateTask(TaskType taskID).

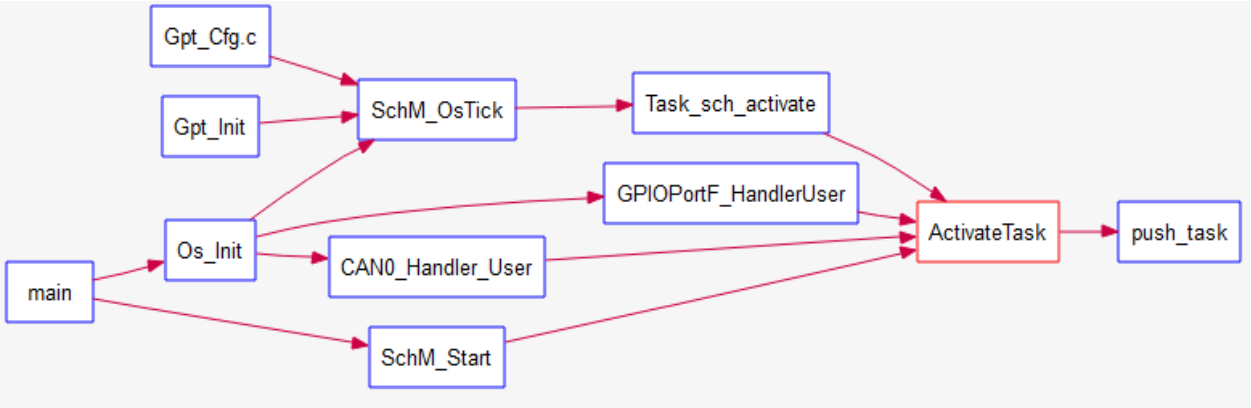


Figure 2-1, Static analysis shows which functions call Activate Task process.

2.3. Dispatcher

The dispatcher searches on the priority buffer FIFO and picks up the next task to be executed. It finds the highest priority and oldest task in a READY state to switch its status to RUNNING and returns the pointer to the next task to be executed but it does not execute it, that is the function of restore_context() explained later in this document, Figure 2-2 shows the dispatcher being called from each OS task (event driven, periodical and interrupt task) before they terminate themselves as detailed in chapter 2.12.

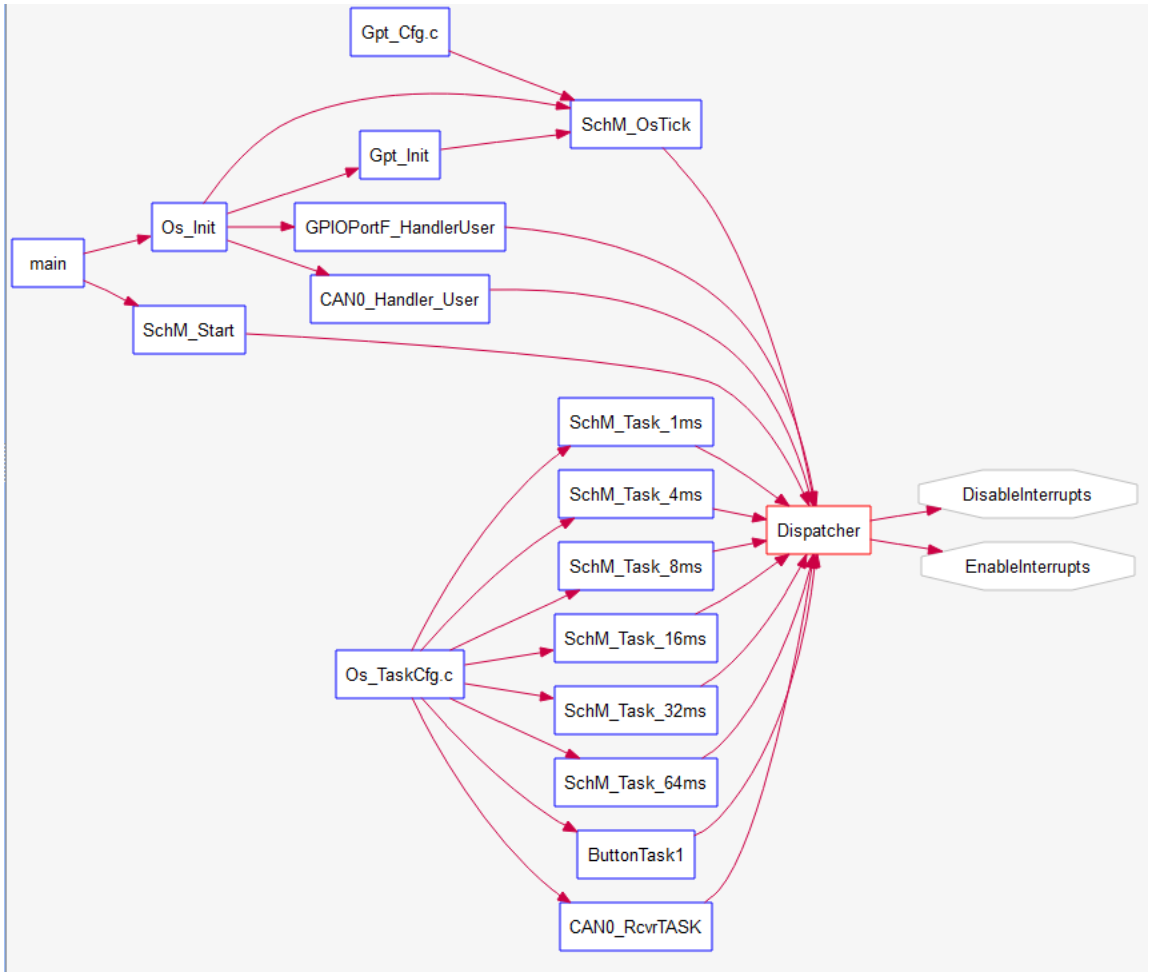


Figure 2-2.All OS tasks shall call the Disptacher function before terminating themselves.

Dispatching a task implies its previous activation. As shown in Figure 2-1, ActivateTask(TASKID) is cyclically running inside SchM_OsTick and later it calls the Dispatcher, meaning it shall be possible to interrupt the current task (Task1) to allocate the next entry in the execution buffer, Figure 2-3 shows the flow diagram devised to carry out the operation.

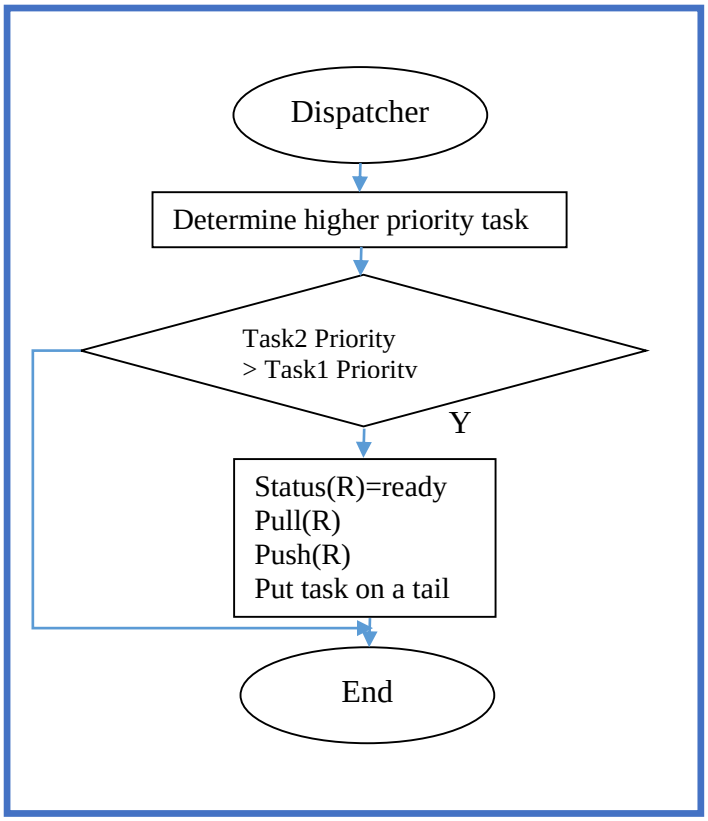


Figure 2-3, Dispatcher flow diagram, it determines the task to be executed but it does not execute it. If Task2 is triggered and it has a higher priority than the currently running Task 1, Task1 is placed on tail according to its priority and the context change is performed.

2.4. Scheduler Mechanism

In an embedded software the processing power is commonly limited, so managing the access to CPU resources is crucial [1]. 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.

In this work, each task control was encoded in a structure called tSchM_Task_Descriptor that contains each mask and offset information used in a cyclic Binary Progression Scheduler technique to activate the next task as shown in Table 2-1. The Binary Progression column indicates the 7 bit counter that is incremented by each clock tick

Decimal Count	Binary Progression	T1 Activation	T1 Period (ms)	T2 Activation	T2 Period (ms)	Collision
0	0 0 0 0 0 0 0	A	1			
1	0 0 0 0 0 0 1					
2	0 0 0 0 0 1 0	A	1			
3	0 0 0 0 0 1 1			A	8	
4	0 0 0 0 1 0 0	A	1			
5	0 0 0 0 1 0 1					
6	0 0 0 0 1 1 0	A	1			

Table 2-1. Binary progression. ´A´ denotes activation of 32ms and 8ms Task.

When the binary counter and the mask matches, the task is executed, the rate is calculated by $task\ rate = OS\ tick * (mask + 1)$ *(2-1)*

$$task\ rate = OS\ tick * (mask + 1) \qquad (2\text{-}1)$$

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 as shown in Figure 2-4. 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.

The tSchM_Task_Descriptor structure must be allocated in the flash memory with the command *const* as declared in file Os_task_cfg.c.

2.5. Priority buffer

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

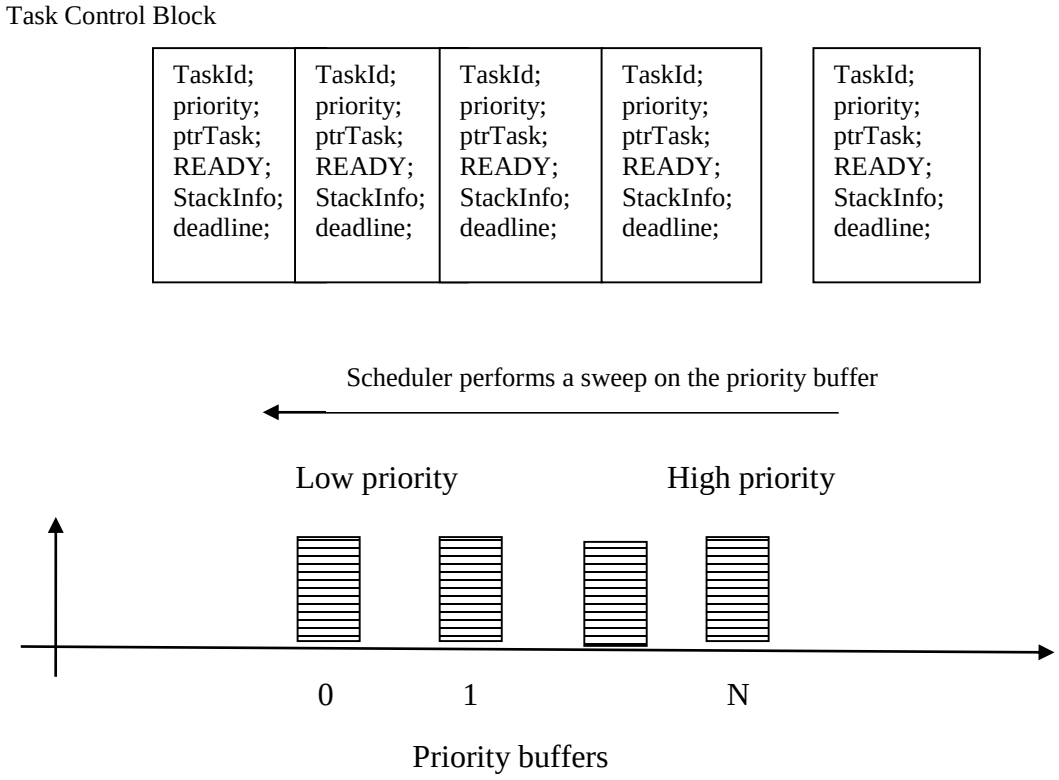


Figure 2-4, Graphical representation of how the scheduler works on the priority buffers.

ActivateTask(TaskType taskID) changes the identified task state to READY and pushes it into the corresponding priority buffer. Figure 2-4 shows all tasks in the priority buffer are in READY state and are waiting for the Dispatcher, which selects the next task to update its status to RUNNING. Then, the dispatcher updates the function pointer with the address of the next function to execute but it does not executes it.

2.6. Task Control Block

A Task Control Block *TCB*, is a data structure that contains all the information related to the operative system tasks. As shown in Table 2-2, it contains the task ID or number, a task status, and a task priority. This structure is a sort of template utilized by all tasks at the time it is created. The context level variable on the Task control block informs the level of the interrupt context [5].

```
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 2-2, Task Control Block Elements.

The advantage of having a TCB is the flexibility and order it brings to the OS, the TCB serves as the information template during the declaration of new tasks.

2.7. Priority Buffers and Task Control Block

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

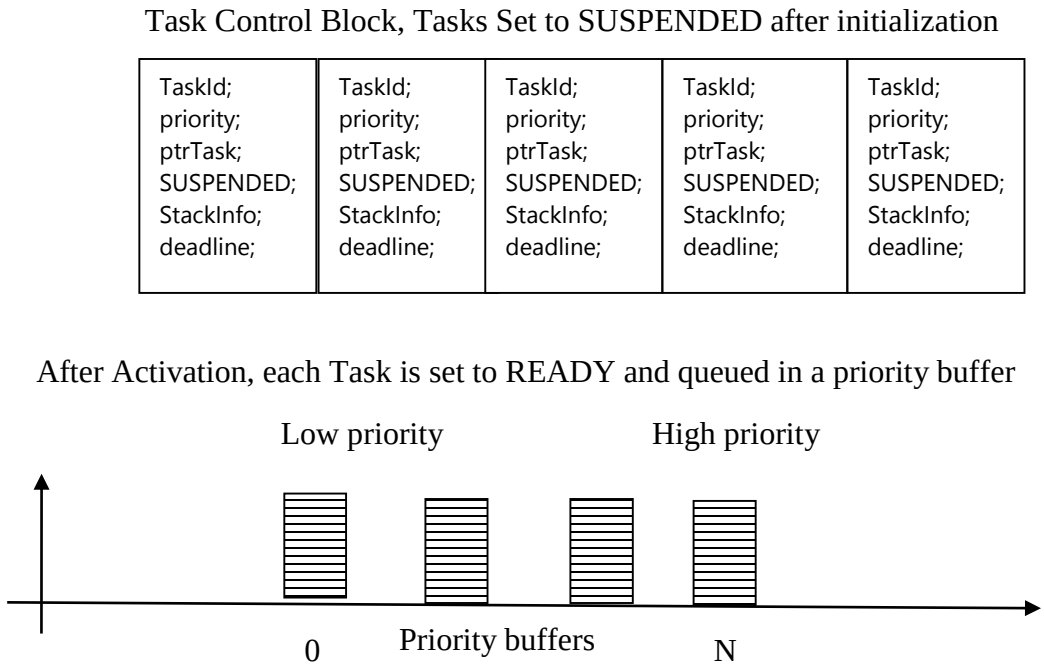


Figure 2-5, Task Control Block representation during task management.

Figure 2-5, shows the structure of the buffers interrupt and the Task Control Block structure, which are located in RAM.

2.8. Task private variables (memory allocation)

The operative system uses memory allocation functions to make Task variables private to other modules, memory allocation allocates the amount of memory for internal status and control structures, the amount of memory is calculated by (2-2). Function Os_Init () reserve memory for ISR and status.

$$ptr\_task\_ctrl\_block = (tTaskControlBlock *) Heap\_Malloc(sizeTCB\_TskNum); \quad (2\text{-}3)$$

2.9. Restore Context

As described in chapter 2.3, 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 [5].

The relationship between periodically activated tasks and restore_context function is depicted in Figure 2-6.

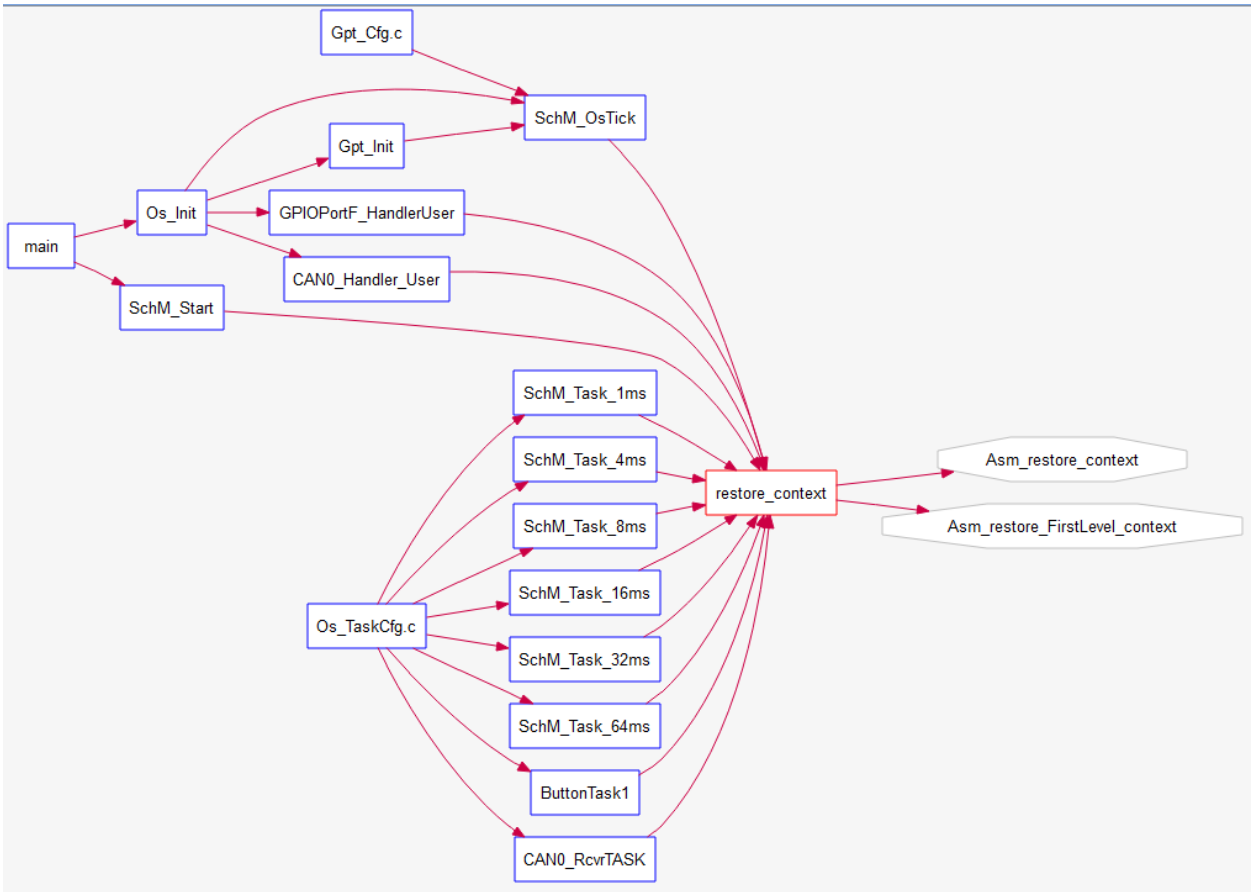


Figure 2-6, Arrows indicate periodic tasks and OsTick are calling [LMBB1]restore_context

The code shown in Table 2-3, loads the pointed task addressed by the dispatcher and updates the register values that correspond to it. There is where the context switch occurs.

```
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 2-3. Code segment to restore context.

2.10. Task state

A task typically can be in any one of the four following states [2]:

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 the task descriptor structure explained below. The main difference between them is that 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).

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.

2.11. Tasks Calls

In OSEK, each task 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();

2.12. Task Termination

Both types of tasks terminate themselves by calling TerminateTask(). This function terminates the task transferring it from RUNNING to a 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()).

2.13. 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 [2]. Any time an interrupt is asserted, 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 in a complex task, especially when the microcontroller automatically saves eight registers. The integrator should design a strategy to keep safe each piece of task information while the microcontroller handles the registers in the designated stack.

Each task has its own stack area on memory, such as it is shown in Figure 2-7.

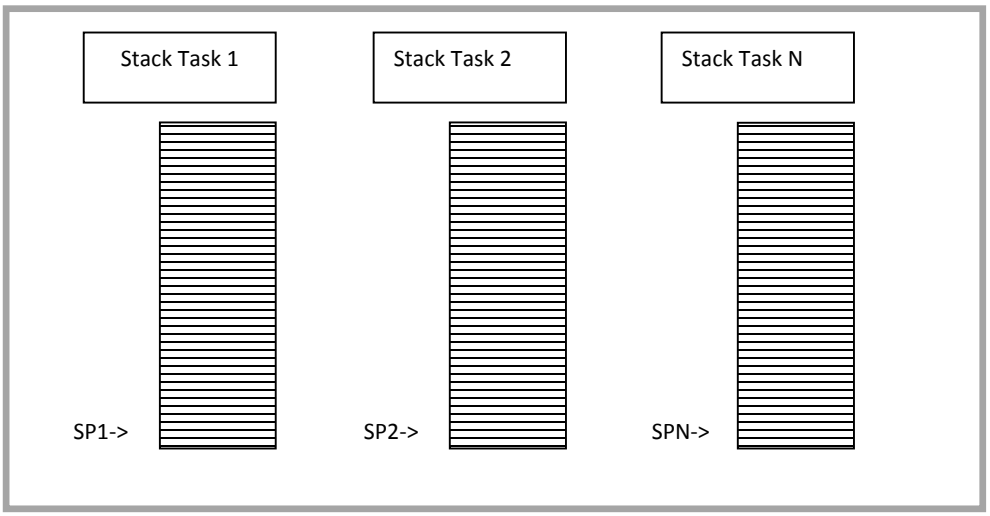


Figure 2-7, Stack pointer model. After initialization, all SP are pointing to the lower address of the task assigned memory known as "thumb bit".

2.14. Stack distribution per Task

The task control block of each task stores its SP address into the Stack info structure by means of Os_Init function.

The assigned addresses are summarized in Table 2 5. A problem is detected and reflected in column Return from ISR start, the problem can be described as a non-deterministic behavior of the microcontroller as it is not possible to predict the return address after processing an interrupt. In other words, the program counter PC loses the return address due the stack pointer is automatically pointing to the expected address but somehow the microcontroller adds (+1 or +2 bytes) randomly, which makes it not possible to keep track of the address to be loaded into the PC register to continue with the program execution.

TaskID	Task Callback	SP start address	*SP End address	**Return from ISR start
0	SchM_Background	0x2000011c	0x20000190	0x20000158
1	SchM_Task_1ms	0x2000019c	0x20000210	
2	SchM_Task_4ms	0x2000021c	0x20000290	
3	SchM_Task_8ms	0x2000029c	0x20000310	
4	SchM_Task_16ms	0x2000031c	0x20000390	
5	SchM_Task_32ms	0x2000039c	0x20000410	
6	SchM_Task_64ms	0x2000041c	0x20000490	
7	ButtonTask1	0x2000049c	0x20000510	

Table 2-4, Stack distribution per TASK.

To solve this issue, the author designed something called “return address compensation” on the SW version “RecyPET_V1p6_with_CANTx_V1p20asc1p2” by applying the following technique (Figure 2-9):

```
.macro swipSP AddrReg
lCopyTCM1:
  POP {R8}                          ; r8 = SP
  MOV  R10,SP                       ; r4 = TCM(L|U) start address
  mvn   r0, 0                       ; r0 = 0xFFFF_FFFF
lLoopCopyTCM1:                      ; in the next 4 lines TCM(L|U) is filled with 0xFFFF_FFFF
  stmia r3!, {r0}                       ; *r3++ = r0   r0; *r3++ = 0xFFFF_FFFF; copy content of r0 to r3
  and increment r3
  cmp   r3, r4                          ; if ( r3 != r4 )
  bne   lLoopCopyTCM1                   ;    goto lLoopCopyTCM1
.endm
```

Figure 2-8.Return Address Compensation. Algorithm devised to compensate the not deterministic behavior of the microcontroller, in this operation the microcontroller checks if it is required to synchronize the SP address, this mitigates the loss of return address.

Figure 2-8, Implements a macro to compensate the data shift when the BX LR instruction is executed, bits 31-1 of LR register 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.

Table 2-5 shows one of the key parts in Os_Init (), this function is called just one time during initialization specifically after power on.

```
    for(u8Index=0;u8Index<Task_Number;u8Index++)
    {
      /*Relative addressing to store Stack allocations*/
      Stacktempo= (UINT32 )ptr_task+  sizeTCB - Ptr_TaskAdd; /// to be able to find end of stac
      ptr_task_end =(tTaskControlBlock *) Stacktempo;
      Stacktempo= (UINT32 ) ptr_task + Ptr_TaskCntxtlvl1;

      ptr_task->GenTaskInfo.StackInfo.Stack_Start = (UINT32 *) (ptr_task);
      ptr_task->GenTaskInfo.StackInfo.Stack_End   = (UINT32 *)ptr_task_end;
      ptr_task->GenTaskInfo.StackInfo.Stack_CntxLvl1 = (UINT32 *)Stacktempo; /*Redundant value
      ptr_task->GenTaskInfo.TaskId        = ptr_Sch_task_descriptor->task_ID ;
      ptr_task->GenTaskInfo.priority      = ptr_Sch_task_descriptor->Priority  ;
      ptr_task->GenTaskInfo.enTaskState   = SUSPENDED ;
      ptr_task->GenTaskInfo.ContextLevel  = 0;
      ptr_task->GenTaskInfo.ptrTask       = ptr_Sch_task_descriptor->Sch_Callback;
      /*It's an event triggered task <<Button Handler>> will return to intermediate GPIOPortF_H
      if (u8Index==(Button_Task1_ID))
      {
        ptr_task->GenTaskInfo.ptrSchM_Ostick = &GPIOPortF_HandlerUser; /*En ensamblador, el har
        PortF_handler=&GPIOPortF_HandlerUser;
      }
      else if (u8Index==(CAN0_Task_ID)) /*It's a periodic task systick handler will return to O
      {
        ptr_task->GenTaskInfo.ptrSchM_Ostick = &CAN0_Handler_User;
      }
      else  /*It's a periodic task systick handler will return to Ostick for activation*/
      {

        ptr_task->GenTaskInfo.ptrSchM_Ostick = &SchM_OsTick;
      }
```

Task information as declared at: tTaskControlBlock->tTaskInfo

Table 2-5, Filing Task Control Block information.

```
  ptr_task->PADA= 0xFAFAFAFA;
 ptr_task->PADB= 0xFBFBFBFB;
 ptr_task->PADC= 0xFCFCFCFC;
 ptr_task->PADD= 0xFDFDFDFD;
 ptr_task->PADE= 0xFEFEFEFE;
 ptr_task->PADF= 0xFFFFFFFF;
 ptr_task->PADG= 0xEEEEEEEE;
 ptr_task->R4  = 0x04040404;
 ptr_task->R5  = 0x05050505;
 ptr_task->R6  = 0x06060606;
 ptr_task->R7  = 0x07070707;
 ptr_task->R8  = 0x08080808;
 ptr_task->R9  = 0x09090909;
 ptr_task->R10 = 0xAAAAAAAA;
 ptr_task->R11 = 0xBBBBBBBB;
 ptr_task->R0  = 0x00000000;
 ptr_task->R1  = 0x01010101;
 ptr_task->R2  = 0x02020202;
 ptr_task->R3  = 0x03030303;
 ptr_task->R12  = 0x12121212;
 ptr_task->R14  = 0x14141414;
 ptr_task->ptrTask_copy= ptr_Sch_task_descriptor->Sch_Callback;
 ptr_task->thumb_bit = 0x01000000;   // thumb bit

 ptr_task++;
 ptr_Sch_task_descriptor++;
}
```

Padding is a technique used to guard memory between TaskInfo and Registers information. In the end, Stack pointer points to the last element of this array "thumb bit".

Table 2-6, Padding as memory guard

The padding is a memory guard, it brings flexibility to the OS during initialization, in case more information is added in tTaskInfo. The current size of the stack is not increased by heap allocation so that performance is not compromised. This memory guard, also promotes a more readable stack.

2.15. Initialization

Initialization is the stage where the buffer size is defined, Os_init() is the function where Start and End address by task are calculated by (2-4) and (2-5).

$$ptr\_task{-}{>}\, StackInfo.StartAddress \;=\; (UINT16\ *)\&ptr\_task{-}{>}\, StackBuffer + 0; \quad (2\text{-}6)$$

$$ptr\_task{-}{>}\, StackInfo.EndAddress \;=\; (UINT16\ *)\&ptr\_task{-}{>}\, StackBuffer + 200; \quad (2\text{-}7)$$

Interrupts are enabled after initialization and once the background task is 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.

2.16. Context switch

Once the “SP End address” is calculated, this value is loaded into SP during restore_context() and before running into the ISR handler [3]. Figure 2-7 shows an example of a 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 Figure 2-7, this 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 the hardware while the entering interrupt handler PUSHES {R4-R11} and at the same time pushes into the SP, its address is decremented automatically.

As shown in Figure 2 10, during ISR handler the microcontroller pushes next registers automatically:

PSR, PC, LR, R12, R3, R2, R1, and R0, e.g. Next registers are automatically saved during GPIOPortF interruption [4].

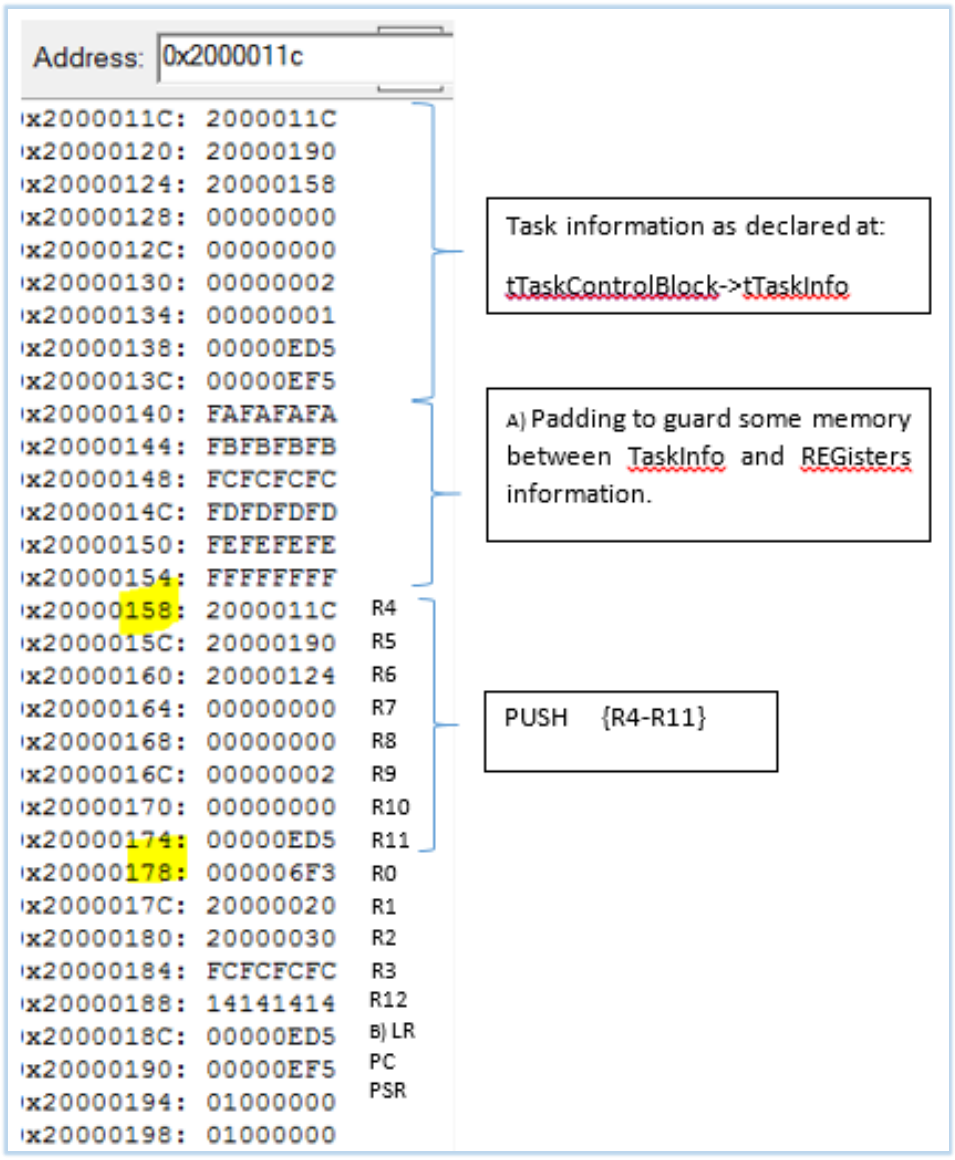


Figure 2-9, RAM memory distribution after Os_Init () is finished.

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}.

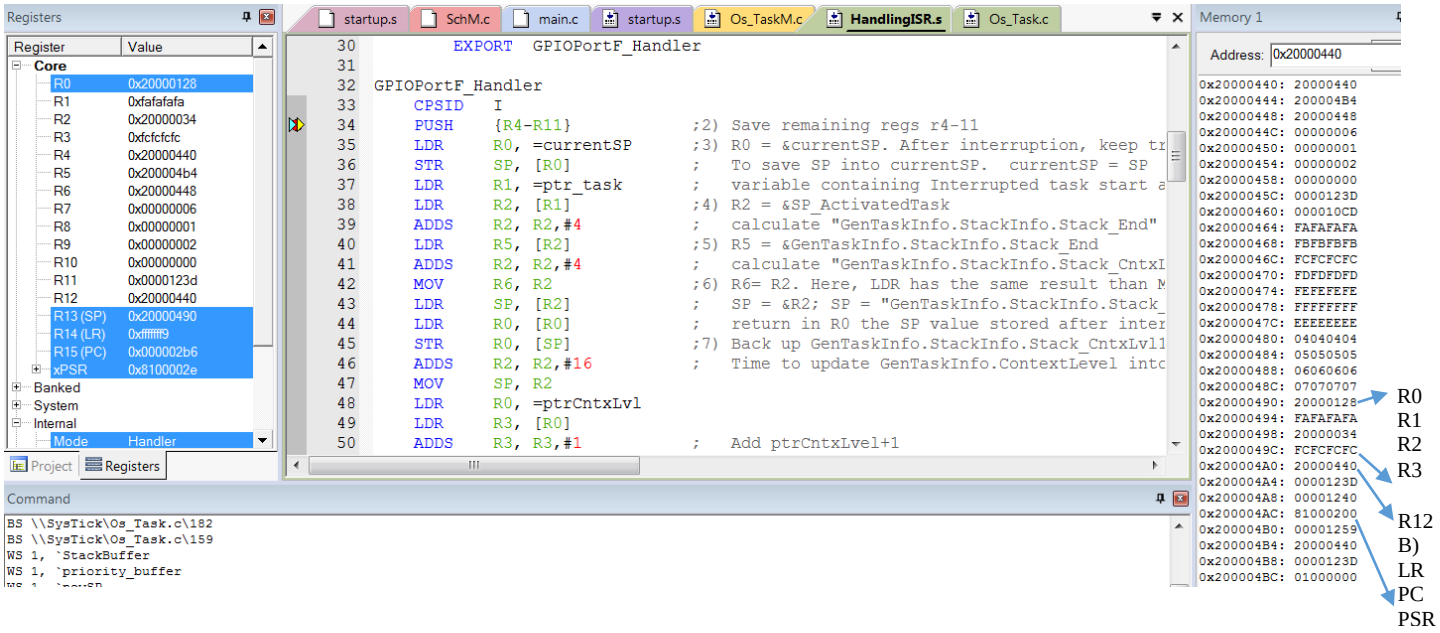


Figure 2-10, Attending GPIO PortF Interruption.

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. This 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, and some overhead is added in each ISR_Handler to make it possible.

A) LR. PC and PSR values are updated within ISR, these values are copied from TASKINFO-> Sch_Callback structure, originally defined within Os_Init().

2.17. 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 2-7 shows the register definitions [3].

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Table 2-7, Systick Registers address.

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 crystal frequency.

Run-Mode Clock Configuration (RCC)
Base 0x400F.E000
Offset 0x060
Type R/W, reset 0x078E.3AD1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved		PWRDN	reserved	BYPASS	XTAL					OSCSRC		reserved			MOSCDIS
Type	RO	RO	R/W	RO	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	RO	RO	RO	R/W
Reset	0	0	1	1	1	0	1	0	1	1	0	1	0	0	0	1

Write ´n´ number in SYSDIV2 bit into the register RCC2 to get the n+1 divisor

Run-Mode Clock Configuration 2 (RCC2)
Base 0x400F.E000
Offset 0x070
Type R/W, reset 0x07C0.6810

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
USERCC2	DIV400	reserved	SYSDIV2						SYSDIV2LSB	reserved					
Type	R/W	R/W	RO	R/W	R/W	R/W	R/W	R/W	R/W	R/W	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0

Figure 2-11, SYSDIV2 of RCC2.

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

2.18. PLL Initialization steps

This initialization is done at the very beginning during the main function [3], in this project, it was initially coded incorrectly into *Mcu* function, but it was corrected later to *PLL* function Table 2-8.

XTAL	*Crystal Freq (MHz)*
0x0	Reserved
0x1	Reserved
0x2	Reserved
0x3	Reserved
0x4	3.579545 MHz
0x5	3.6864 MHz
0x6	4 MHz
0x7	4.096 MHz
0x8	4.9152 MHz
0x9	5 MHz
0xA	5.12 MHz
0xB	6 MHz (reset value)
0xC	6.144 MHz
0xD	7.3728 MHz
0xE	8 MHz
0xF	8.192 MHz

XTAL	*Crystal Freq (MHz)*
0x10	10.0 MHz
0x11	12.0 MHz
0x12	12.288 MHz
0x13	13.56 MHz
0x14	14.31818 MHz
0x15	16.0 MHz
0x16	16.384 MHz
0x17	18.0 MHz
0x18	20.0 MHz
0x19	24.0 MHz
0x1A	25.0 MHz
0x1B	Reserved
0x1C	Reserved
0x1D	Reserved
0x1E	Reserved
0x1F	Reserved

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R

	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Table 2-8, RCC2 (PLL registers) in TM4C microcontroller.

Steps to configure the PLL to operate at 16MHz Xtal Oscillator at 80MHz [6].

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 2-8 using XTAL table for the desired configuration. To clear OSCSRC2 bits, this action selects the main oscillator as the 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 for SYS_R to become high, this is controlled by the processor and indicates the PLL is stabilized.

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

3. Serial driver configuration

SSI (Nokia Display) driver is developed according to the following requirements:

- The 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.

3.1. Serial ports configuration file

This application, configures all channels on a single file Protocol_cnf.c, number of channels and configuration per channel.

Serial configuration is based on the pointer to their registers, this allows to remove switch-case sentences on the serial initialization

4. Photoelectric Sensors

4.1. Infrared pairs

Designing a low cost solution is crucial to produce an affordable machine, the solution is to implement infra-red sensors stage devised as four transmitter-receiver paired elements connected to an ADC as shown in Fig. 4-1 where the ADC module is configured to work [3]. It is attended by SysTick_Handler interruption every 25ms [3]. Only 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.

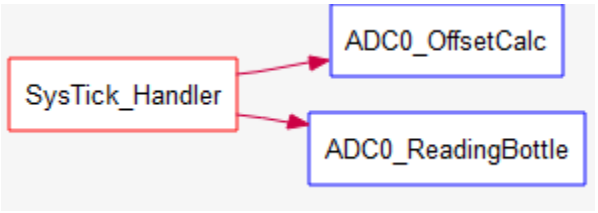


Figure 4-1, ADC0_OffsetCalc, is called once during initialization, the ADC0_Reading bottle is called inside the periodical 32ms task, only when an object is detected on the conveyor.

The function ADC_ReadSensorOffset reads the instant value and calculated in *(4-1)*

$$p\_medicion.u16InstantVal = (ADC0\_SSFIFO3\_R \& 0xFFF);$$

$$if\ (p\_medicion.u16InstantVal < p\_medicion.u16Offset\_Buffer[ch\_count])\ p\_medicion.u16Offset\_Buffer[ch\_count] = p\_medicion.u16InstantVal; \quad (4\text{-}2)$$

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 *(4-3)*

$$pmedicion.u16Mean = (word)((float)u16suma/MEAN\_BUFFER\_SIZE + 0.5); \quad (4\text{-}4)$$

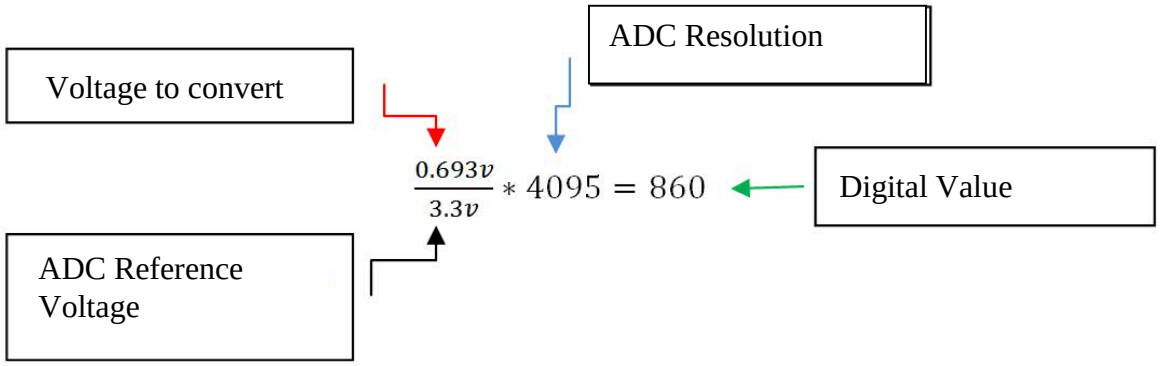
The usage of Offset calculation comes later in (4-5):

$$p\_medicion.u16Mean\_wOffset[ADC\_CH] = p\_medicion.u16Mean - p\_medicion.u16Offset\_Buffer[ADC\_CH]; \quad (4\text{-}5)$$

It takes 23 ms to be fully executed, this time involves waiting till ADC Hardware triggers its end of conversion flag as in (4-6).

$$while((ADC0\_RIS\_R \& 0x08) == 0)\{\}; \quad (4\text{-}7)$$

The challenge is to integrate this change into the operative system, where Systick_Handler was replaced by a TimerA0 interrupt [3] and the IR mean voltage calculation was enabled, once a bottle was over the feeding band.



Channel	Count	Volt PD3	Expected Volt
0	1056	0,851	0,850989
1	1073	0,862	0,8646886
2	1071	0,868	0,8630769
3	1044	0,843	0,8413187

Table 4-1, Real Voltage (VoltPD3) vs. Converted Voltage (Volt).

The ADC Resolution depends on the amount of bits the microcontroller utilizes for voltage conversion from analog to digital, TMC123 uses 12 bits or 2k (4095 samples) it uses a 3.3 V source

power as reference voltage and calculates the equivalent voltage as digital value as shown in Table 4-1.

Even though the mentioned technique accurately identifies when a PET bottle is processed, the following problem was found during development. The system crashes if the interrupt service routine does not either acknowledge or disarm the device released after completeness, for example in the sensor stage; After finishing the analog to digital conversion, the control register raises a status bit indicating the conversion is finished, the software shall poll this flag and deactivate it to continue with the software execution or the *microcontroller* will call a reset indicating its malfunction.

4.2. OMRON sensor

The sensor is allocated at the end of the conveyor as shown in Figure 4-2, the sensor part number is E3ZM-B PET Bottle Detection.

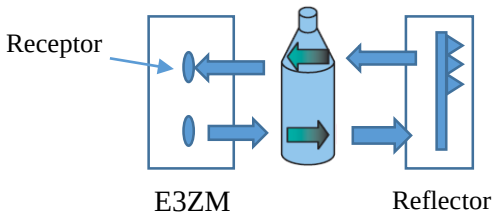


Figure 4-2, Omron sensor-reflector polarizes with PET bottle.

As shown in Figure 4-2, the E3ZM-B sensor 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 [11].

5. Measurements

The conveyor is the first contact to feed the machine, a DC motor shall push the bottle inside the sensor box (element 1 in Figure 1-1). The first two columns in the truth table shown in Appendix A, control EN and /D2 enabling OUT1/2 states to be defined by controlling IN1 and IN2 states [6].

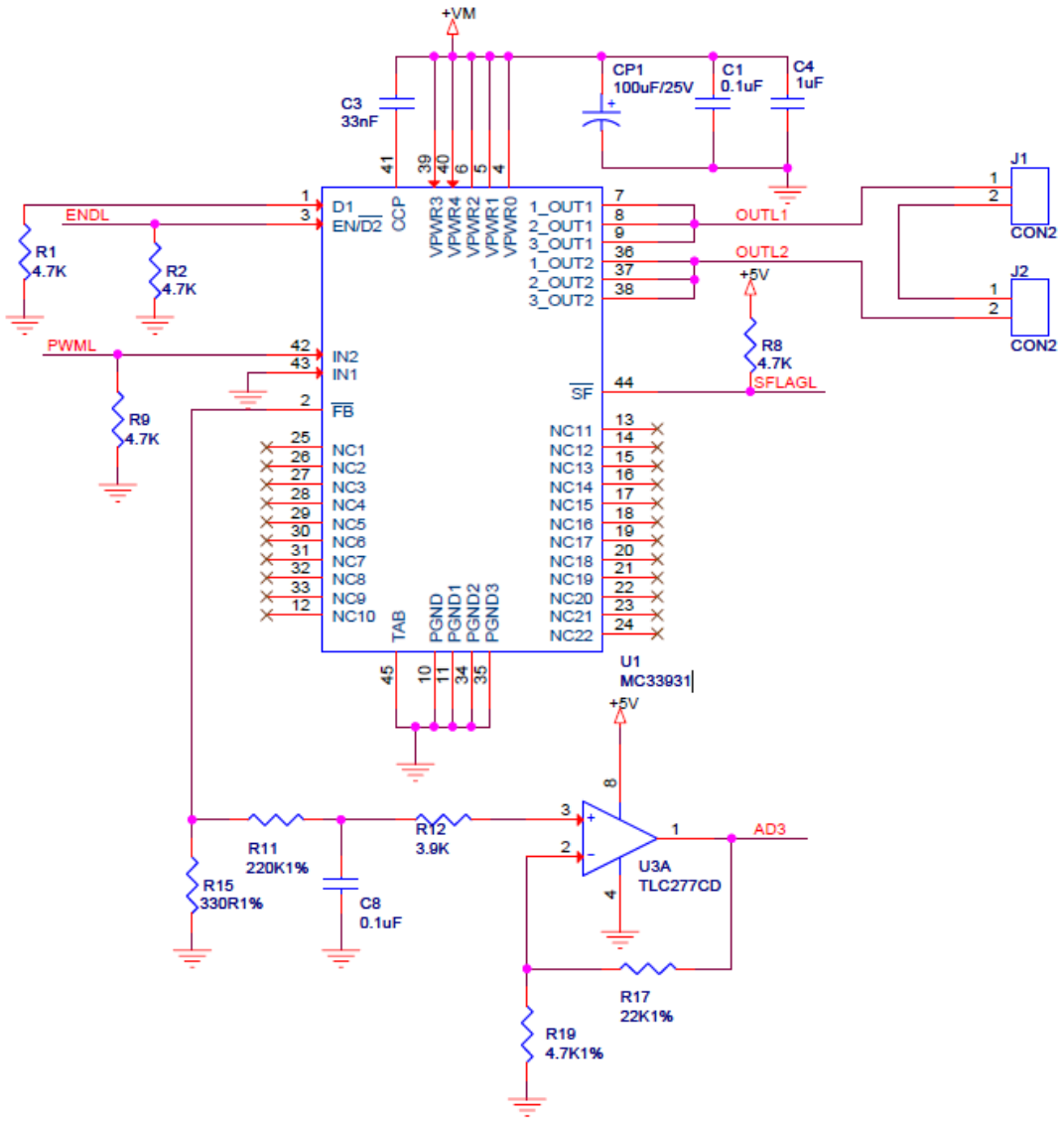


Figure 5-1. The H Bridge Schematic, the DC motor is connected between J1 and J2.

Figure 5-1 shows the schematic H Bridge, where IN1 is directly connected to GND. In order to enable breaking it is necessary to lift pin 43, attach a wire to it and control it properly [6]. Below the board connection for MC33931 is shown:

Once a power source is connected to J4 pin 1 (+ 9V), and pin 2 to (GND), the current consumption is equal to 360mA.

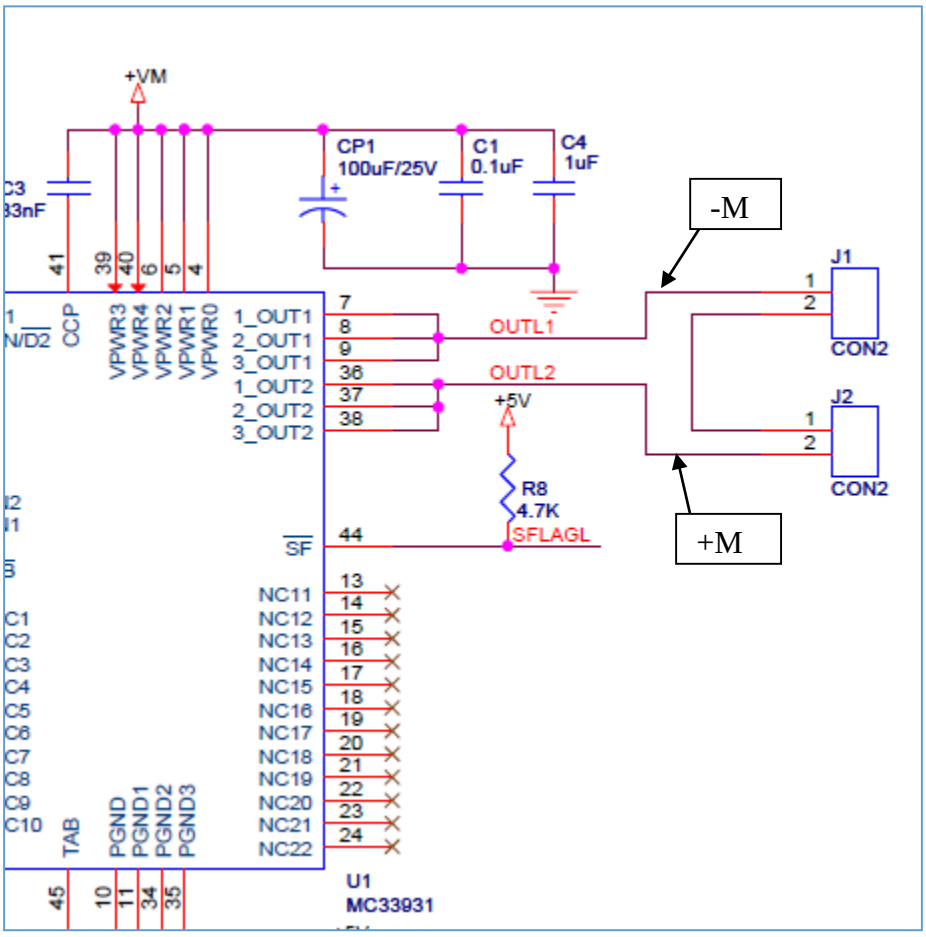


Figure 5-2. H Bridge controls the DC motor that feeds the machine.

5.1. Feeding machine, bottle to Inside

Terminal ENDL (JP2 pin1) +5V connected,

T = 1ms
Duty cycle = 100us

Figure 5-3, Signal used to Feed the machine, measured between PWML (JP2 pin 15) and GND.

5.2. Returning Bottle to outside

Terminal ENDL (JP2 pin1) GND connected,

T = 1ms
Dutycycle = 900us

Figure 5-4, Signal used to Return the Bottle, measured between PWML (JP2 pin 15) and GND.

6. Prototype

The prototype was assembled in parallel the SW development. Using Figure 1-1 as reference, we can identify on the upper left the sensor box, it is made out of acrylic and painted with aerosol to block all beam of light, the infrared sensors are attached in the middle the circuit to amplify the infrared beam of light is installed on the top of the box. At the bottom, the Nokia graphical display shows the ITESO logo during initialization.

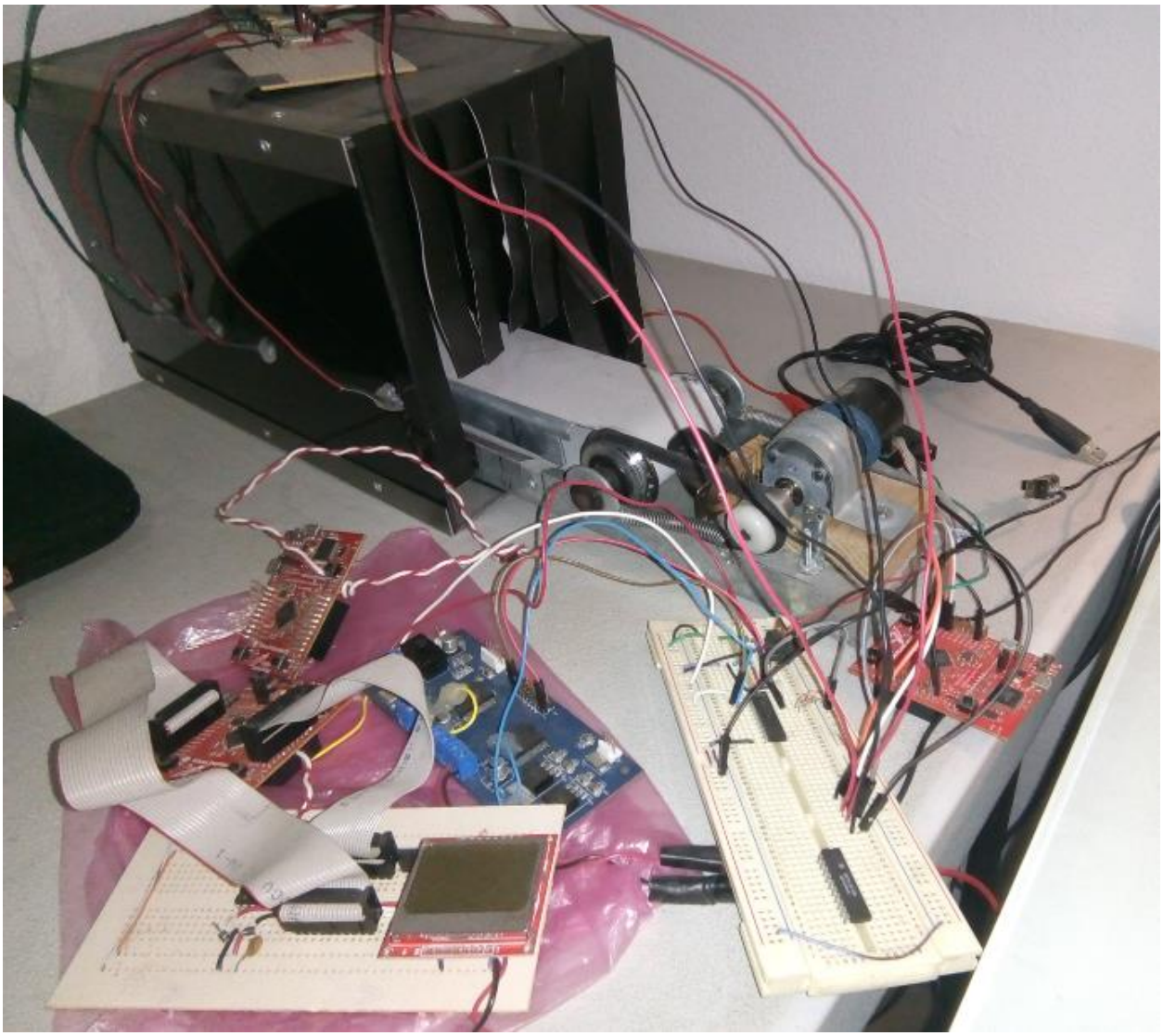


Figure 6-1, Recycling machine, feeding stage prototype.

The red boards are the CortexM devices known as development board (Texas instruments Tiva TMC123), there is where the operative system reside.

The blue board is used to control the motor spin direction and speed, (Motor H bridge board MC33931 from NXP).

The prototype is functional but there is a reset after performing context switch. The shared system “RecyPET_V1p8_with_CANTx_V1p26asc1p6 “can handle the periodical tasks but cannot be preempted by a higher priority interrupt, in other words a task cannot be interrupted by a second task with higher priority, there is no preemptive functionality, meaning the project is not fulfilling the requirement during the context switch where the SP is loaded with a wrong address, this address is pushed into the PC register, as a consequence, the program execution is lost.

7. Conclusion

The goal of this work is to produce the feeding stage of a PET bottle recycling machine at the lower cost possible, as well, all the development tools were selected due to their low cost. For example, the Tiva TMC123 was selected as the main development board due its flexibility, its low power consumption, and its low price, the design environment, such as the compiler and debugger can be downloaded for free for academic purposes. One of the findings was that this toolset works well for small projects, however, a multi task, fully preemptive operative system demands more stability; For example, during debugging the tool loses synchronization with the development board. A suggestion for future research would be to invest more in a proper toolset.

The toolset limitations made it very difficult to find the reason why the operative system crashes when an OS Task executes the context switch. After several tests, it was found that the SP is loaded to the expected address but the microprocessor automatically increments (+1 or +2) randomly to this address. Later, the command BX LR is executed to return from context switch, which loads the PC register with the dereferenced value from SP, as a consequence, the program execution is lost as it is not possible to load the PC register with the expected address,

Appendix

A. CONVEYOR, H BRIDGE CONTROL

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

Device State	Input Conditions				Status	Outputs	
	EN/$\overline{D2}$	D1	IN1	IN2	$\overline{SF}$	OUT1	OUT2
Forward	H	L	H	L	H	H	L
Reverse	H	L	L	H	H	L	H
Freewheeling Low	H	L	L	L	H	L	L
Freewheeling High	H	L	H	H	H	H	H
Disable 1 (D1)	H	H	X	X	L	Z	Z
IN1 Disconnected	H	L	Z	X	H	H	X
IN2 Disconnected	H	L	X	Z	H	X	H
D1 Disconnected	H	Z	X	X	L	Z	Z
Under-voltage Lockout(30)	H	X	X	X	L	Z	Z
Over-temperature(31)	H	X	X	X	L	Z	Z
Short-circuit(31)	H	X	X	X	L	Z	Z
Sleep Mode EN/$\overline{D2}$	L	X	X	X	H	Z	Z
EN/$\overline{D2}$ Disconnected	Z	X	X	X	H	Z	Z

The EN/!D2 pin performs the same function as the D1 pin when it goes to a logic LOW the outputs are immediately tri-stated [11].

8. References

[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*, 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*, 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

[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

[11] OMRON Technical Staff, *Retro-reflective with M.S.R. function,* Aug.2013. *[Online]. Available: https://www.ia.omron.com/product/item/606/*

[12] "Operative Systems in Embedded microprocessors", class notes for Embedded Systems Specialty, summer 2013.

[13] El Pais "México, a la cabeza del reciclaje de plástico en América", ROCÍO AGUILERA VÁZQUEZ. May. 2018. [Online]. Available: https://elpais.com/internacional/2018/05/16/actualidad/1526429688_205528.html