Proyecto Tiva-C

S.E.P.A.

Fernando Dorado Rueda.

Fdorado.rueda@gmail.com

Contenido

TABLA DE ILUSTRACIONES

1. INTRODUCCIÓN

Vivimos en una época en la que la robótica y la automatización se va afianzando en nuestro día a día. En este trabajo implementaremos un sistema operativo en tiempo real (RTOS). Los RTOS están en contacto directo con el mundo exterior, sometidos a restricciones de tiempo y enfrentados a problemas y sistemas relativamente complejos.

El objetivo principal de nuestro proyecto ha sido simular un entorno industrial, en el cual existe un recurso compartido (un AGV) que compartes varias cintas de transporte de piezas. Para la sincronización y el correcto uso del recurso compartido usaremos semáforos binarios (MUTEX). Para la simulación de la llegada de pieza avisaremos al sistema por cola de mensajes introducidas por computadora mediante puerto serie.

Además, nuestro sistema cuenta con protecciones de humedad dado por el sensor YL-38 y gracias al ADC obtenemos la humedad que sufre el robot. También cuenta con un módulo habilitado como master de comunicación SSI, para, en caso de querer ampliar el entorno industrial, para añadir nuevos microcontroladores o periféricos solo bastaría con conectarlos a los pines descritos.

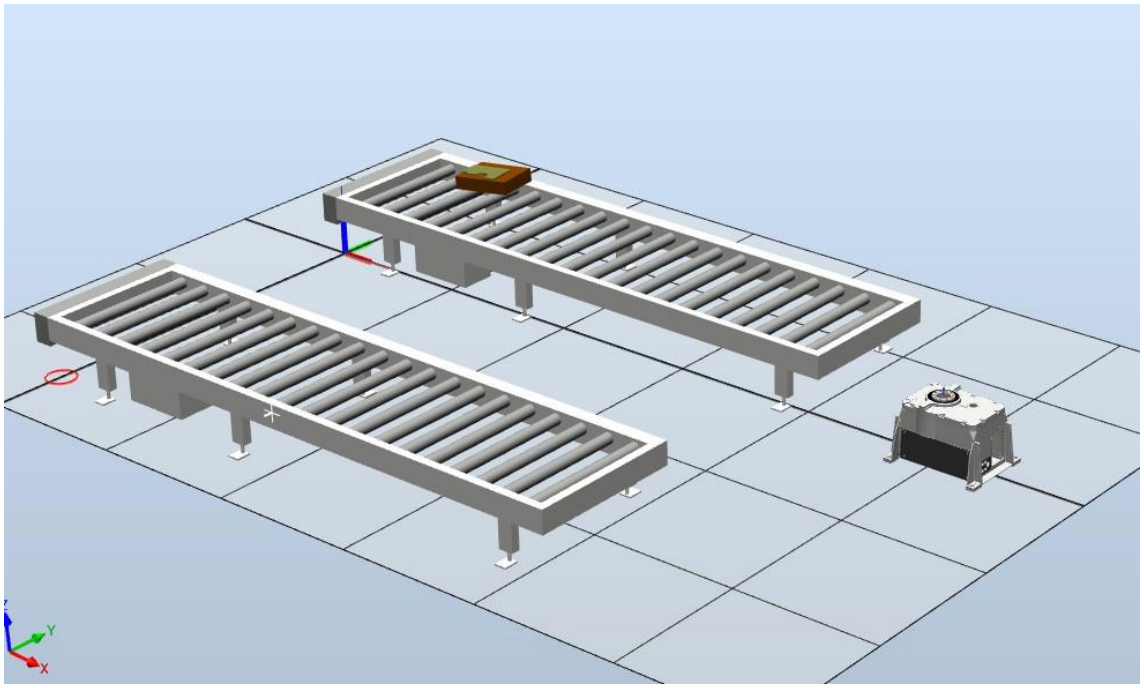

FIGURA 1. ENTORNO INDUSTRIAL A SIMULAR

1.1 SOLUCIÓN PROPUESTA

1.1.1 Elementos de software utilizados

Antes de profundizar en el método de resolución de nuestro problema explicaremos cómo funcionan los elementos que hemos usado para lograr la programación concurrente.

- **TI-RTOS:** Sistema operativo en tiempo real para microcontroladores de Texas-Instrument. Es una herramienta muy buena ya que combina un Kernel multitarea en tiempo real con componentes de middleware adicionales.

- **Threads (Hilos):** Muchas aplicaciones en tiempo real deben realizar varias funciones aparentemente no relacionadas al mismo tiempo, a menudo en respuesta a eventos externos. Estas funciones son llamadas hilos. SYS/BIOS nos proporciona diferentes tipos de hilos con diferentes prioridades:
    1. **Hardware interrupts**
    2. **Software interrupts**
    3. **Task:** Las tareas siguen un diagrama de estados que permiten dar soporte a diversas actividades simultaneas con un solo procesador, de manera que se respeten sus diversas prioridades y puedan conmutarse de una a otra tarea según las prioridades y pueda conmutarse de una a otra según las necesidades.

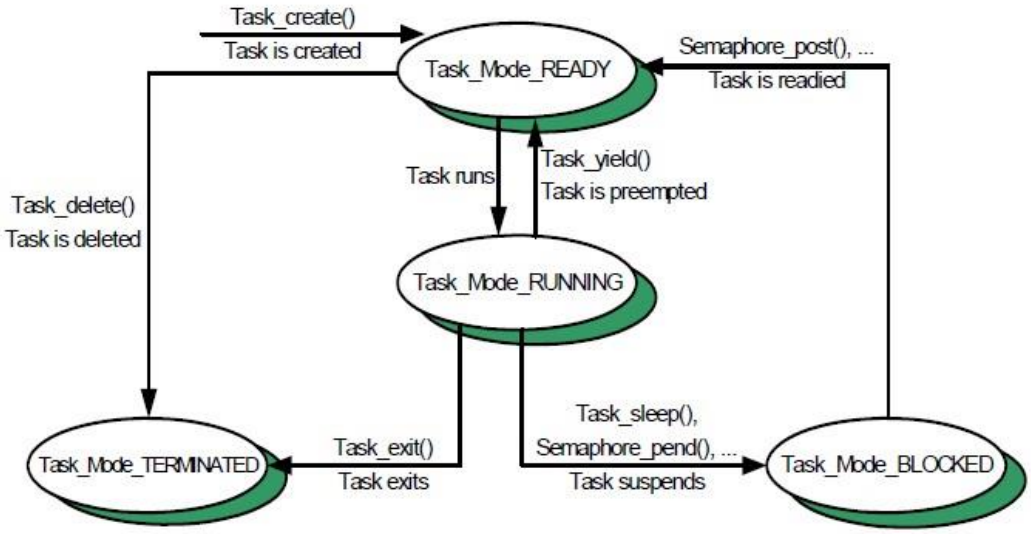

FIGURA 2. DIAGRAMA DE ESTADOS DE UNA TAREA

4. **Background thread (Idle)**

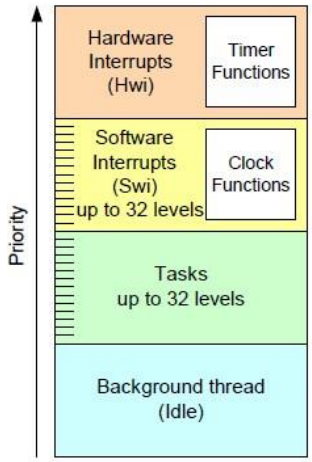

FIGURA 3. THREAD PRIORITIES

- **Cola de mensajes:** Son un medio de comunicación entre hilos basada en mensajes. Es muy útil para comunicar asíncronamente aplicaciones desacopladas, además, nos permite un mejor rendimiento, fiabilidad y escalabilidad. Los mensajes se almacenan en la cola hasta que se procesan y eliminan. Cada mensaje se procesa una vez sola.

  Las colas de mensajes permiten a diferentes partes de un sistema comunicarse y procesar operaciones de forma asíncrona. Una cola de mensajes ofrecer un búfer ligero que almacena temporalmente los mensajes, y puntos de enlace que permiten a los componentes de software conectarse a la cola para enviar y recibir mensajes.

- **Semáforos:** Los semáforos es una estructura diseñada para sincronizar dos o más hilos, de modo que el uso de los recursos que comparten los hilos se realice de forma ordenada y sin conflicto entre ellos. Un semáforo básico es una estructura formada por una posición de memoria y dos instrucciones, una para reservarlo y otra para liberarlo.
  En nuestro caso usaremos semáforos binarios (MUTEX), por el cual solo puede tomar el valor de 1 (un hilo puede acceder al recurso) o 0 (el recurso está siendo utilizado). Se empieza por inicializar la posición de memoria a 1, a continuación, cada vez que un hilo quiera acceder a dicho recurso, hará primero una petición con la primera de las llamadas disponibles. Cuando el RTOS ejectuta esa llamada, comprueba el valor que hay en la posición de memoria del semáforo, y si es distinta de cero, se limita a restarle 1 y devolver el control al programa. Sin embargo, si es nula, duerma el proceso que hizo la petición y lo mete en la cola de procesos que esperan al semáforo. Cuando el proceso ha terminado el acceso al recurso, usa la segunda llamada para liberar el semáforo.

- **SSI:** Proporciona una funcionalidad de comunicación serie síncrona.
- **ADC:** Conversor de una señal analógica a digital.

1.1.2 Componentes elegidos

Nuestro proyecto es sobre todo a nivel de software, por tanto no hemos necesitamos muchos componentes.

TIVA-C TM4C1294CMPTD

Este microcontrolador nos proporciona numerosas ventajas gracias a su gran versatilidad. Tiene numerosos pines, pulsadores, botones, conexión USB, Ethernet.

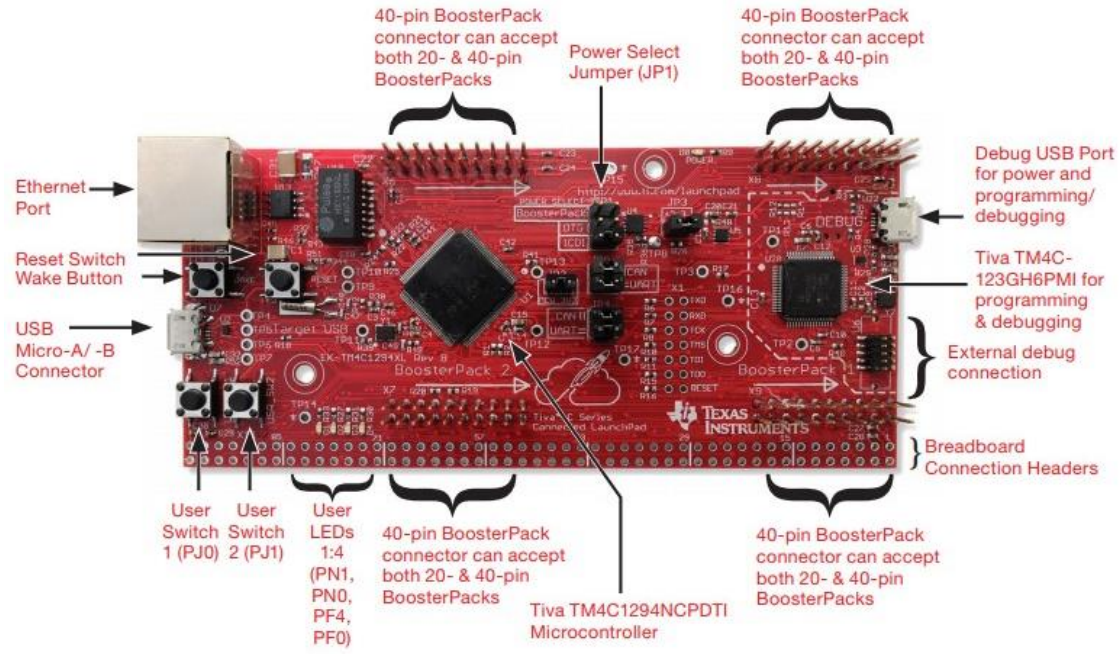

FIGURA 5. TIVA - C

2 SENSOR DE HUMEDAD YL-38

Este sensor tiene la capacidad de medir la humedad del suelo. Aplicando una pequeña tensión entre los terminales del módulo YL-69 hace pasar una corriente que depende básicamente de la resistencia que se genera en el suelo y ésta depende mucho de la humedad. Por lo tanto al aumentar la humedad la corriente crece y al bajar la corriente disminuye.

Características:

- Tensión de funcionamiento 3.3-5V.
- Corriente: 35mA
- A0: Salida analógica que entrega una tensión proporcional a la humedad.
- D0: Salida digital que nos permite ajustar el sensor.

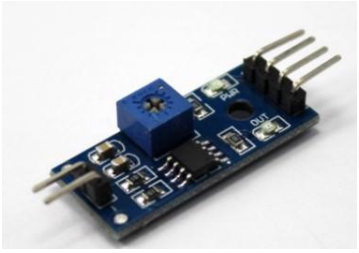

FIGURA 6. SENSOR HUMEDAD YL-38

2. ALCANCE

2.1 OBJETIVOS

2.1.1 OBJETIVO GENERAL

El objetivo general ha sido implementar la concurrencia y la programación en tiempo real en nuestro microcontrolador.

2.1.2 OBJETIVOS ESPECÍFICOS

- Lograr la comunicación entre diferentes hilos.
- Conseguir la sincronización entre hilos gracias a semáforos.
- Usar la cola de mensajes como un elemento de comunicación y sincronización entre hilos.
- Habilitar el modulo del protocolo de comunicación SSI para futuros usos.
- Afianzar los conocimientos sobre la configuración y uso de ADC.

2.2 REQUISITOS

Vamos a comenzar exponiendo algunos requisitos que debe cumplir nuestro proyecto. En principio, se considerarán estos aspectos que se citarán a continuación:

Funcionales

F.1: El sistema será capaz de iniciar una función que se encargue del movimiento del AGV.

F.2: El ordenador deberá de proporcionar la corriente necesaria para el funcionamiento de nuestro micro.

Prestaciones

P.F.1.1: El sistema será capaz de dirigirse a la cinta por la cual llega una pieza.

P.F.1.2: La comunicación UART nos permite transmitir a 115200 b/r.

Operación

O.1: El sistema podría el funcionamiento introduciendo datos por puerto serie de nuestra computadora.

Eléctricos

E.1: Se deberá alimentar por USB a 5V al TM4C1294. Con esto, alimentar el circuito completamente a una tensión correspondiente a cada componente electrónico.

E.2: El voltaje del módulo de humedad es de 3.3V .

Test

A lo largo de la construcción se realizarán varios test para la comprobación del correcto funcionamiento del sistema. Entre ellos incluimos:

T.1: Se probará y calibrará el sensor de humedad.

T.2: Se comprobará el correcto uso del recurso compartido (AGV) entre hilos.

T.3: Se revisará la correcta comunicación por UART al ordenador.

T.4: Se revisará la cola de mensajes para leer correctamente.

Interfaz

I.1: El sistema constará de una interfaz por consola en nuestro ordenador.

Calidad

M.C.1: La calidad del proyecto se buscará que sea la máxima posible a nivel de software, implementando una correcta sincronización y fiabilidad del sistema.

Modularidad

M.1: El sistema cuenta con un módulo SSI para futuras ampliaciones.

2.3. ESQUEMA DE CONEXIONADO

El esquema de conexionado es bastante sencillo debido a que solo usamos un pin con el exterior (PB5), que es el encargado de leer del sensor de humedad.

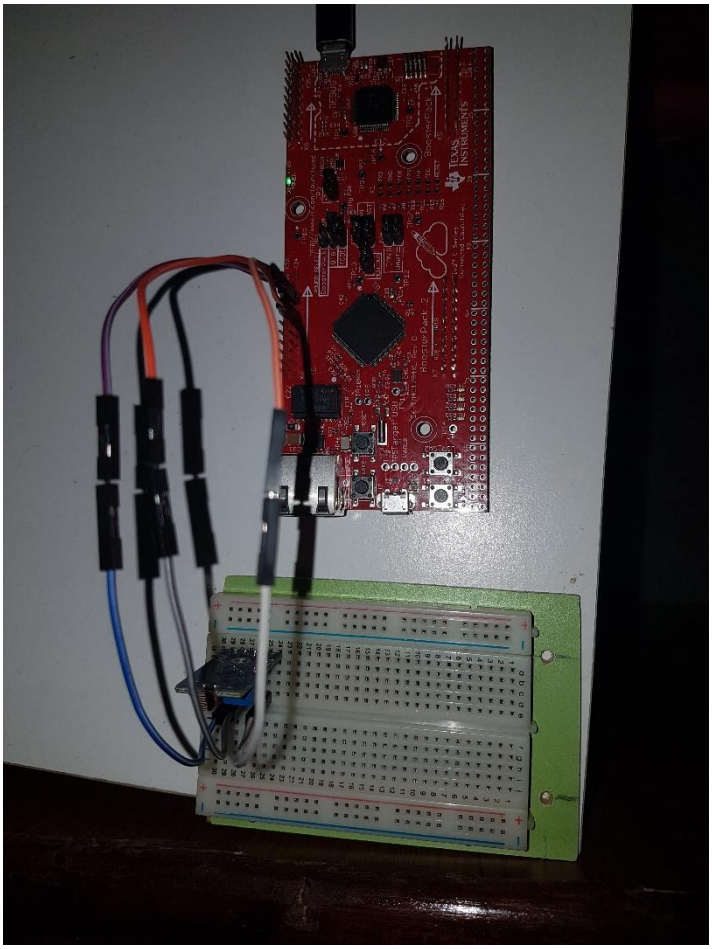

FIGURA 7. ESQUEMA ELÉCTRICO DE NUESTRO SISTEMA

3. CONFIGURACIÓN

Para configurar el Kernel correctamente haremos uso de la herramienta que nos proporciona T.I. : **XDCtools.** Es un software que nos proporciona las herramientas para la configuración del sistema junto con numerosos módulos y APIs. Para añadir o habilitar módulos tendremos que modificar el archivo **.cfg**

3.1 CONFIGURACIÓN GENERAL

En el habilitaremos y añadiremos los bloques necesarios para habilitar las interrupciones por hardware, semáforos, el uso de logs,colas de mensajes, tareas y las funciones necesarias para arrancar el TI-RTOS.

```
var Defaults = xdc.useModule('xdc.runtime.Defaults');
var Diags = xdc.useModule('xdc.runtime.Diags');
var Error = xdc.useModule('xdc.runtime.Error');
var Log = xdc.useModule('xdc.runtime.Log');
var Main = xdc.useModule('xdc.runtime.Main');
var Memory = xdc.useModule('xdc.runtime.Memory');
var System = xdc.useModule('xdc.runtime.System');
var Text = xdc.useModule('xdc.runtime.Text');
var BIOS = xdc.useModule('ti.sysbios.BIOS');
var Clock = xdc.useModule('ti.sysbios.knl.Clock');
var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');
var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');

var SysMin = xdc.useModule('xdc.runtime.SysMin');

var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var Task = xdc.useModule('ti.sysbios.knl.Task');
var Queue = xdc.useModule('ti.sysbios.knl.Queue');
System.SupportProxy = SysMin;
```

3.2 CONFIGURACIÓN DEL KERNEL

Para arrancar el Kernel del sistema operativo añadimos las siguientes líneas:

```
/***************CONFIGURACION KERNEL****************/
/* Use Custom library */
var BIOS = xdc.useModule('ti.sysbios.BIOS');
BIOS.libType = BIOS.LibType_Custom;
BIOS.logsEnabled = true;
BIOS.assertsEnabled = true;
```

3.3 CONFIGURACIÓN DRIVER

```
var TIRTOS = xdc.useModule('ti.tirtos.TIRTOS');
```

Esta parte del código se refiere a la correcta inicialización de las interrupciones por hardware, semáforos, tareas y cola de mensajes.

3.3.1 INTERRUPCIONES POR HARDWARE

Habilitamos una interrupción por harware que será la encargada de activar la rutina de servicio de interrupción que habilitaremos después.

```
/* Interrupción timer */

var hwi0Params = new Hwi.Params();
hwi0Params.instance.name = "Timer_2A_INT";
Program.global.Timer_2A_INT = Hwi.create(39, "&ISR", hwi0Params);
Program.stack = 1024;
BIOS.heapSize = 0;
BIOS.cpuFreq.lo = 40000000;
LoggingSetup.sysbiosSwiLogging = true;
```

3.3.2 SEMÁFOROS

Crearemos 3 semáforos que serán encargados de la sincronización y correcto funcionamiento entre los diferentes hilos de nuestro programa. Después se profundizará en la explicación del uso de cada uno

```
LoggingSetup.sysbiosSemaphoreLogging = true;
LoggingSetup.loadTaskLogging = true;


var semaphore1Params = new Semaphore.Params();
semaphore1Params.instance.name = "Cola_Sem";
semaphore1Params.mode = Semaphore.Mode_BINARY;
Program.global.Cola_Sem = Semaphore.create(null, semaphore1Params);

var semaphore3Params = new Semaphore.Params();
semaphore3Params.instance.name = "AGV_Sem";
semaphore3Params.mode = Semaphore.Mode_BINARY;
Program.global.AGV_Sem = Semaphore.create(null, semaphore3Params);

var semaphore4Params = new Semaphore.Params();
semaphore4Params.instance.name = "UART_Sem";
semaphore4Params.mode = Semaphore.Mode_BINARY;
Program.global.UART_Sem = Semaphore.create(null, semaphore4Params);
```

3.3.3 COLA DE MENSAJES

Crearemos una cola de mensajes que será la encargada de almacenar las ordenes que tendrá que hacer nuestro AGV y además los irá apilando en una FIFO para realizar todas las órdenes.

```
/* Cola de mensajes */

var queue0Params = new Queue.Params();
queue0Params.instance.name = "Cola_AGV";
Program.global.Cola_AGV = Queue.create(queue0Params);
```

3.3.4 TAREAS

Crearemos dos tareas con diferente nivel de prioridad cada una, una se encargará del correcto movimiento de nuestro AGV, llevando dentro de él la lectura de la cola de mensajes y que hacer según la orden que llegue, y otra será la encargada de la lectura por UART de la orden y almacenarla en la cola de mensajes.

```
/* Tareas */

var task1Params = new Task.Params();
task1Params.instance.name = "movAGV";
task1Params.priority = 1;
Program.global.movAGV = Task.create("&mov_AGV", task1Params);

var task2Params = new Task.Params();
task2Params.instance.name = "UART";
task2Params.priority = 2;
Program.global.UART = Task.create("&leeUART", task2Params);
```

4. EXPICACIÓN DEL CÓDIGO

4.1 VARIABLES GLOBALES

Definiremos las variables globales asociada al reloj, la medida de la humedad dada por el ADC y una variable tipo estructura que nos servirá para almacenar el mensaje dentro de nuestra cola de mansaje.

```
uint32_t reloj;
uint32_t hum;

typedef struct MsgObj {
Queue_Elem   elem;
char val;                              // Valor del mensaje
} MsgObj, *Msg;                        // Usamos Msg como puntero a MngObj
```

4.2 FUNCION PRINCIPAL

Nuestra función main principal solo se encargará de llamar a una función en la que habilitaremos el hardware que vamos a usar. Llamarémos a BIOS_start() para arrancar el SYS/BIOS y la ejecución del archivo .cfg. Por último tenemos un bucle while(1) para que el programa nunca acabe.

```
void main(void)
{
   ini_hardware();                     // Inicializamos el hardware
   BIOS_start();
   while(1);
}
```

4.3 INICIALIZACIÓN HARDWARE

La función **void ini_harware(void)** se encargará de inicializar todos los periféricos que usemos en nuestro programa. La comentaremos por partes:

1) Inicializamos el reloj a 120Mhz

```
reloj=SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480), 120000000);
```

2) Habilitamos los LEDs como salidas

```
 /* Habilitar los periféricos implicados: GPIOF, N */

  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);

 /* Definir tipo de pines */
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0 |GPIO_PIN_4);
    GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0 |GPIO_PIN_1);
```

3) Habilitamos el TIMER2 y las interrupciones asociadas a dicho TIMER. Al producirse la interrupción ejecutaremos la rutina de servicio de interrupción (ISR).

```
/* Timer2 */
// Habilitamos el periferico
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER2);

// Establecemos como periódico
TimerConfigure(TIMER2_BASE, TIMER_CFG_PERIODIC);

// Periodo dividido por 2
ui32Period = (SysCtlClockGet() /2);

// Establecemos el timer con dicho periodo
TimerLoadSet(TIMER2_BASE, TIMER_A, ui32Period);

// Habilitamos las interrupciones por timer
TimerIntEnable(TIMER2_BASE, TIMER_TIMA_TIMEOUT);

// Activamos

TimerEnable(TIMER2_BASE, TIMER_A);
```

4) Habilitamos el periférico UART para la comunicación por puerto serie con nuestro ordenador. Esto nos permite leer por puerto serie y almacenar dicho valor leído en la cola de mensajes. Las características de nuestro periférico UART son las siguientes:

- Periféricos: UART0 y Pines 0 y 1 del periférico A
- Velocidad de transmisión: 115200 b/s
- 8 bits de dato
- Sin paridad
- Stops bits: 1 bits

```
/* UART */
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
UARTConfigSetExpClk(UART0_BASE, reloj, 115200,(UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
```

5) Autorizamos la comunicación SSI como máster para futuras ampliaciones de nuestro proyecto y su interconexión con otros controladores. El procedimiento para el correcto funcionamiento del sistema es:

- Habilitamos el módulo SSI0.
- Los pines para dicha comunicación serán los : 2, 3,4,5 del puerto A.
- Habilitaremos la comunicación para dichos pines.
- Los argumentos de la función **SSIConfigSetExpClk** son:
    - La dirección del módulo SSI.
    - Velocidad del reloj suministrado al módulo SSI.
    - Protocolo de comunicación para transferencia de datos.
    - Modo de operación.
    - Velocidad del reloj.
    - Numero de bits transmitidos por barrido.

```
/* SSI */
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
GPIO_PIN_2);
GPIOPinConfigure(GPIO_PA2_SSI0CLK);
GPIOPinConfigure(GPIO_PA3_SSI0FSS);
GPIOPinConfigure(GPIO_PA4_SSI0XDAT0);
SSIConfigSetExpClk(SSI0_BASE, 48000000, SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER,
2000000, 16);
SSIEnable(SSI0_BASE);
```

Una vez tengamos habilitado el periférico, transmitiremos los datos con la función **SSIDataput(uint32_t ui32Base, uint32_t ui32Data).**

6) Activación ADC asociado al pin PB5 para la lectura del sensor de humedad. Leeremos el ADC por una interrupción de harware.

```
 /* ADC */
 SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
 GPIOPinTypeADC(GPIO_PORTB_BASE, GPIO_PIN_5);
 SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
 ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_TIMER, 0);
 ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH11 | ADC_CTL_IE |
ADC_CTL_END);
 ADCIntEnable(ADC0_BASE, 3);
 IntEnable(INT_ADC0SS3);
 ADCSequenceEnable(ADC0_BASE, 3);
```

4.4 RUTINA DE SERVICIO DE INTERRUPCIÓN (ISR)

Esta rutina está asociada al timer que hemos configurado en el archivo .cfg . Su funcionamiento es muy importante para lograr la exclusión mutua y permitir a los hilos acceder a los recursos compartidos. Será la encarga de limpiar el flag de interrupción y además incrementaran en una unidad los semáforos asociados a la cola de mensajes y la lectura de datos por UART. Esto es muy importante ya que hará que las tareas se ejecuten según su nivel de prioridad y en un principio se encuentran esperando a la habiitación del semáforo.

```
void ISR(void)
{
      TimerIntClear(TIMER2_BASE, TIMER_TIMA_TIMEOUT); // Limpiamos el flag
       Semaphore_post(Cola_Sem);
       Semaphore_post(UART_Sem);
}
```

4.5 LECTURA DE UART

Esta tarea será la encargada de leer los datos por consola y los apilará en la cola de mensajes. Para almacenar los valores necesitamos :

- Definir el mensaje usando la estructura que hemos creado al principio.
- Crear un puntero a MsgObj para usar la cola.
- Inizializar el puntero apuntando a la dirección del mensaje

Una vez dentro del bucle cíclico, esperaremos a la habilitación del semáforo dado por el ISR y almacenaremos los valores leidos por consola en la estructura **msg** dentro del campo **val.** Apilaremos dicha variable en la cola a la espera de ser leída, pero de esto se encargará la otra tarea. Una vez completado dicho proceso habilitamos el semáforo para que vuelva a ser usado.

```
void leeUART(void){
MsgObj msg;
Msg msgp;
msgp = &msg;

 while (1)
    {
//Esperamos la habilitación por ISR
 Semaphore_pend(UART_Sem, BIOS_WAIT_FOREVER);

//Si hay un caracter disponible lo guardamos en la variable tipo estructura
if (UARTCharsAvail(UART0_BASE)) {
    //Enviamos por consola el valor introducido
    msg.val = UARTCharGet(UART0_BASE);
    //Imprimimos el valor tecleado por consola
    UARTCharPut(UART0_BASE,  msg.val);
    //Enviamos dato al periférico esclavo
    SSIDataPut(SSI0_BASE,msg.val);
    // Pasamos usando punteros el valor a la cola de mensajes
    Queue_put(Cola_AGV, (Queue_Elem*)msgp);
    // Desbloqueamos el semaforo asociado a la cola de mensajes
    Semaphore_post (Cola_Sem);
        }
    }
}
```

4.6 MOVIMIENTO AGV

Esta tarea será la encargada de leer de la cola de mensajes, y en función del valor de la cola el AGV se dirigirá a una zona u otra. La habilitación del mensaje es idéntica a la tarea anterior. Para que el AGV funcione correctamente es necesario no interrumpirlo cuando se está dirigiendo a una zona, y por ello hemos creado el semáforo asociado al AGV. Dicho semáforo no se habilita por ISR como los anteriores, lo inicializamos a 1. Dentro de cada orden de movimiento deberemos de asegurarnos que el semáforo vale 1, ya que si vale uno solo puede significar que el AGV se acaba de inicializar o que no tiene ninguna tarea en funcionamiento. Una vez que se ejecute la orden de moviento, decrementaremos el semáforo, protegiendo asi el recurso compartido (AGV) y al final el movimiento liberaremos el semáforo para que se realice otra orden. Además, también usaremos el semáforo asociado a la cola de mensajes para solo ejecutar el AGV cuando le lleguen mensajes y evitar problemas.

Dentro de cada modo de funcionamiento hemos imprimido por pantalla lo que haría el AGV y hemos dormido la tarea. A la hora de implementar dicho código en un entorno real, deberemos cambiarlo y poner las órdenes necesarias para el movimiento real del AGV.

Hemos omitido dentro de este código las funciones asociada a escribir caracteres por pantalla. La totalidad del código se encuentra en el .c

Cabe destacar que las ordenes se irán apilando en la cola de mensajes, y el robot las irá realizando cuando acabe la anterior, no habrá una respuesta asíncrona de una orden. Tan solo se podrá parar una orden con el reset asociado a la placa.

```
void mov_AGV(void)
{
  MsgObj mensaje;

Msg msgp;

msgp = &mensaje;
//Habilito el recurso compartido (AGV) para que los diferentes hilos accedan
a el
Semaphore_post(AGV_Sem);

      while(1)
      {
            Semaphore_pend(Cola_Sem, BIOS_WAIT_FOREVER);
                                    // Esperamos que la cola de mensajes
reciba un mensaje
            msgp = Queue_get(Cola_AGV);
// Leemos el contenido de la cola para saber donde se dirige el AGV

            if (msgp->val == '0'){
                /* Modo vuelta a empezar */
                Semaphore_pend(AGV_Sem, BIOS_WAIT_FOREVER);
                Task_sleep(15000);
                Semaphore_post(AGV_Sem);
```

```
else
       if(msgp->val == '1'){
       /* Recibida petición cinta 1 */
      Semaphore_pend(AGV_Sem, BIOS_WAIT_FOREVER);
      Task_sleep(10000);
      Semaphore_post(AGV_Sem);
                }

 else

       if(msgp->val == '2'){
 /* Recibida peticion cinta 2 */
Semaphore_pend(AGV_Sem, BIOS_WAIT_FOREVER);
Task_sleep(10000);
Semaphore_post(AGV_Sem);
                        }
```

4.7 LECTURA ADC

En dicha interrupción leeremos el ADC y en caso de que la humedad sea alta enviaremos una señal de error.

```
void ADC_INT(void){
        ADCIntClear(ADC0_BASE, 3);
        ADCProcessorTrigger(ADC0_BASE, 3);
        ADCSequenceDataGet(ADC0_BASE, 3, &hum);

        /* Proteccion humedad para AGV */
        if(hum<350){
            UARTCharPut(UART0_BASE, 'E');
            UARTCharPut(UART0_BASE, 'R');
            UARTCharPut(UART0_BASE, 'R');
            UARTCharPut(UART0_BASE, 'O');
            UARTCharPut(UART0_BASE, 'R');
        }

}
```

5. HERRAMIENTAS USADAS

Para la lectura por puerto serie usaremos **Tera-Term.** Es un programa de software libre que nos permite la correcta lectura del puerto serie. La configuración de dicho programa será la siguiente:

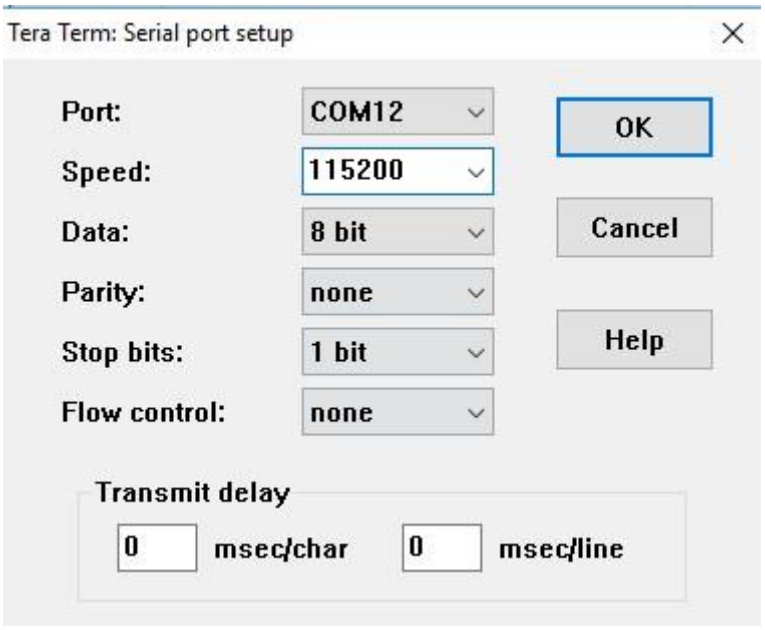

FIGURA 8. CONFIGURACIÓN TERA-TERM

Para comprobar el correcto DEBUG y funcionamiento de nuestro programa, utilizaremos la herramienda que nos brinda Texas Instrument : ROV.

Los resultados de esta herramienta se encontraran junto a los test, ya que según el modo de funcionamiento el sistema cambia.

6. FUTURAS AMPLIACIONES

Este proyecto es muy escalable, ya que cuenta con los medios correctos de sincronización para no tener problemas a la hora de añadirle funcionalidades, dentro de esas funcionalidades son las siguientes:

- Comunicar varios AGVs y que funcionen conjuntamente.
- Adición de módulos asociados a evitar colisiones.
- Hacer logs de los movimientos del AGV para hacer un reparto óptimo entre las cintas y envitar los tiempos muertos.
- Proponer una zona de carga batería y que el AGV se vaya directamente ahí al acabar la jornada de trabajo.
- Monitorizar en tiempo real el punto actual del AGV y cuantas veces realiza cada tarea.

7. BIBLIOGRAFÍA

Para la realización del proyecto me he basado en los manuales que nos proporciona Texas Instrument:

- **TI-RTOS 2.16 for TivaC : Getting Started Guide**
- **TI-RTOS 2.20 : User's Guide**
- **SYS/BIOS (TI-RTOS Kernel) v6.45 : User's Guide**
- **TivaWare Peripheral Driver Library**