

SISTEMA I2C SIMPLE

Stellaris Lunchpad LM4F120

Descripción breve

Reporte del desarrollo de un sistema simple de comunicación I2C, el cual está conformado por un maestro y un esclavo, donde un dato es enviado desde la pc hacia el maestro y el maestro envía un dato a través del bus I2C para que el esclavo ejecute una tarea determinada.

Juan Miguel Vargas Sánchez
jmvarsan@gmail.com

Comunicación i2c entre LM4F120 y MSP430

El protocolo de comunicación i2c permite la transferencia de datos entre dispositivos electrónicos, dicha transferencia se hace a través de un bus de 3 líneas, una para el reloj, una para la transferencia de datos y la última para la línea común o tierra compartida entre los dispositivos, la frecuencia del reloj oscilar entre los 100 Kh y los 3.4 Mh.

Un sistema de comunicación i2c sencillo puede estar conformado por un maestro y un esclavo, para un sistema más complejo se pueden incluir varios maestro y varios esclavos, cabe mencionar que solo el maestro es quien puede iniciar la transferencia de datos y cada esclavo es identificado por un numero hexadecimal, para esta práctica se estará implementando un sistema de comunicación sencillo.

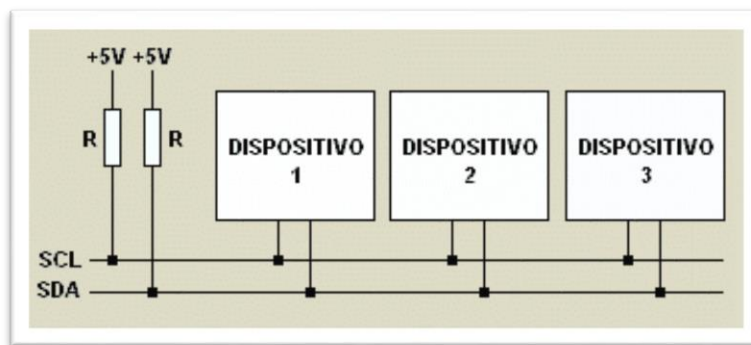


Figura 1. Sistema de comunicación I2C.

Desarrollo

EL sistema de comunicación i2c implementado está conformado por una tarjeta Stellaris LM4F120 y una Launchpad MSP430 ambas de Texas Instruments, dichas tarjetas programables estarán actuando como maestro y esclavo respectivamente, cabe mencionar que se explicara más a detalle el código implementado en la tarjeta Stellaris.

La tarjeta Stellaris cuenta con 4 módulos para implementar la comunicación i2c, para esta práctica se utilizara el módulo I2C_2 el cual ocupa los pines PE_4 (scl) para el reloj de sincronización y PE_5 (sda) para la transmisión de datos.

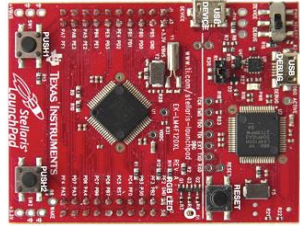
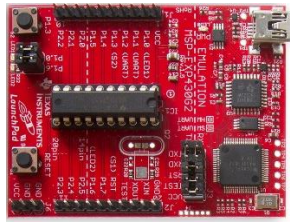


En esta práctica se hace uso de diversos recursos de la tarjeta Stellaris principalmente el módulo de i2c, interrupciones externas, periféricos de entrada/salida y módulo UART.

La funcionalidad del sistema implementado consiste en lo siguiente:

El maestro espera una cadena la cual puede ser “on” o “off”, dicha cadena es enviada desde una PC a través de la comunicación UART, el maestro al recibir la cadena la envía al esclavo a través del bus I2C el cual al recibir la cadena enciende o apaga el led de la tarjeta MSP430 según sea el caso y finalmente retorna un 0 como señal de que la transferencia se realizó con éxito.

Hardware y software

El hardware utilizado durante la práctica se muestra en la siguiente tabla:

Stellaris LM4F120	
Launchpad MSP430	
USB to TTL D-SUN-V3.0	
2 Resistores de 4.7 KOhms	

Cabe mencionar que para la comunicación serial se utiliza el USB to TTL, esto para una implementación rápida y así evitar el uso de un cable serial y el circuito integrado MAX232.

El software utilizado en esta práctica fue el Hercules el cual sirve para hacer el envío y recepción de datos por comunicación serial entre la pc y la tarjeta LM4F120.

Conexiones

Las conexiones entre las tarjetas se listan a continuación:

Conexión	LM4F120	MSP430
SCL	PE4	P1.6
SDA	PE5	P1.7
Gnd	Gnd	Gnd
corriente	+3.3 v	-

Tabla 1. Conexiones entre Maestro y esclavo.

Conexión	LM4F120	MSP430
Ground	GND	GND
Transmisión (Im4f120)	PE1 (Tx)	RXD
Recepción (Im4f120)	PE0 (Rx)	TXD

Tabla 2. Conexión entre *Im4f120* y *TTL*.

El sistema I2C simple está conformado por una tarjeta Stellaris LM4F120 como maestro y una tarjeta MSP430 como esclavo como se muestra en la figura 2.

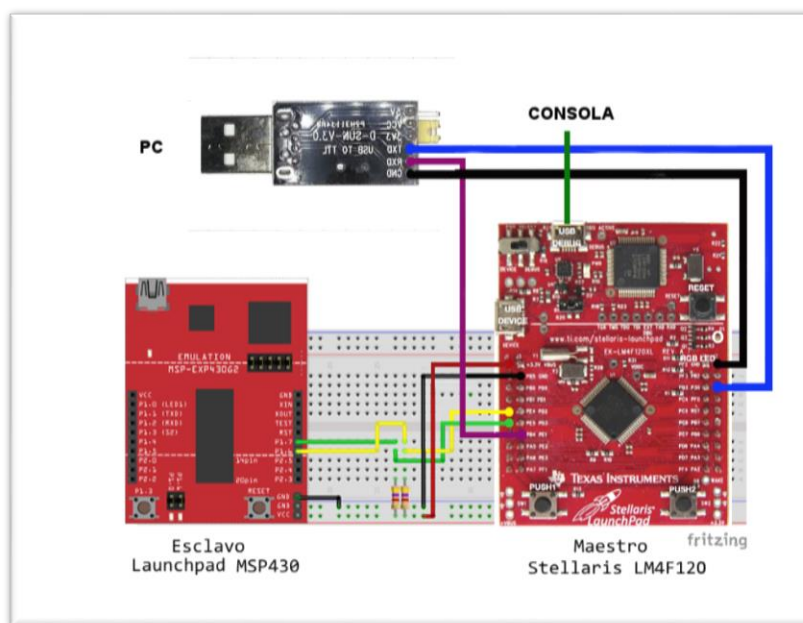


Figura 2. Diagrama general del sistema de comunicación I2C simple.

El Maestro recibe una cadena desde la PC a través de la comunicación UART (esta cadena puede ser “on” o “off”), este dato es enviado a través del bus I2C hacia el esclavo, si la cadena es “on” el esclavo enciende el LED rojo de la tarjeta MSP430 y si es “off” lo apaga, por último el esclavo envía un dato nulo como confirmación de que se ha ejecutado la tarea correctamente (este último paso no está funcionando).

A continuación se muestra el diagrama de flujo del funcionamiento del sistema.

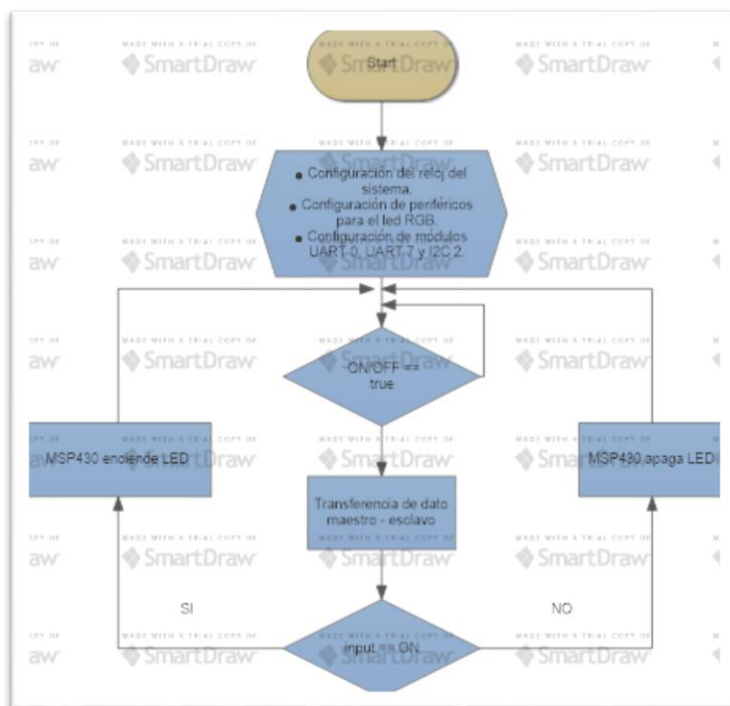


Figura 3. Diagrama de flujo del sistema de comunicación SPI simple.

El resultado final de las conexiones se muestra en la figura 4

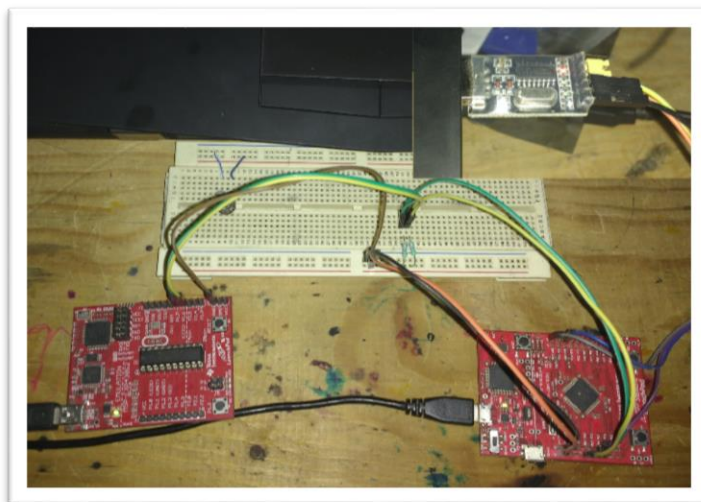


Figura 4. Conexión de tarjetas y ttl reales.

Al iniciarse el sistema se imprimen información útil para saber que el sistema está trabajando correctamente (Figura 5) y se despliegan instrucciones (Figura 6).



Figura 5. Información de inicialización correcta del sistema.

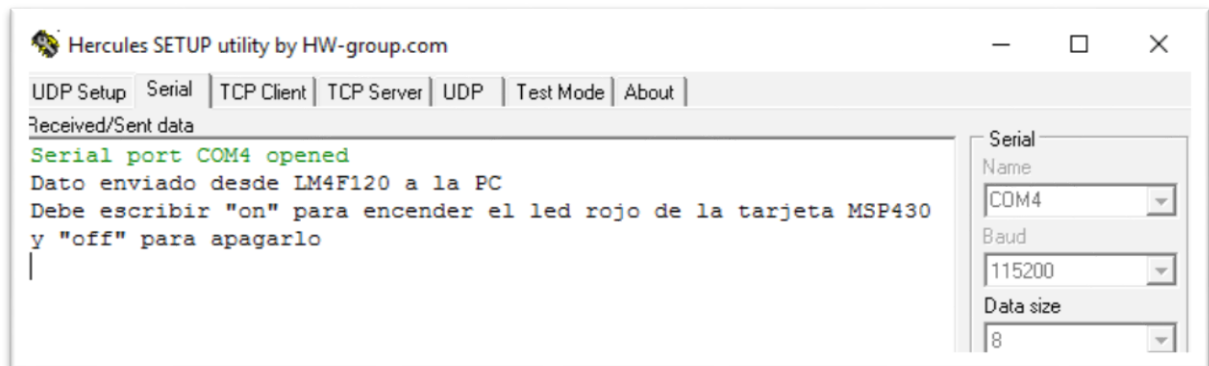


Figura 6. Despliegue de instrucciones en consola de comunicación serial.

De acuerdo a las instrucciones desplegadas se procede a hacer el envío de una cadena, en este caso se envía primero la cadena "on" con lo cual encenderemos el LED de la tarjeta MSP430.

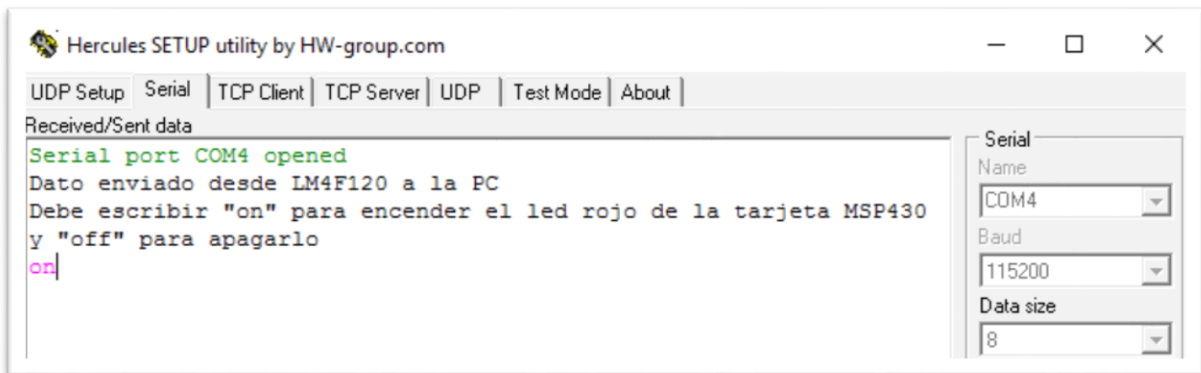


Figura 7. Envío de cadena "on".



Figura 8. Resultado del envío de cadena "on".

En la consola de información se despliega lo siguiente.

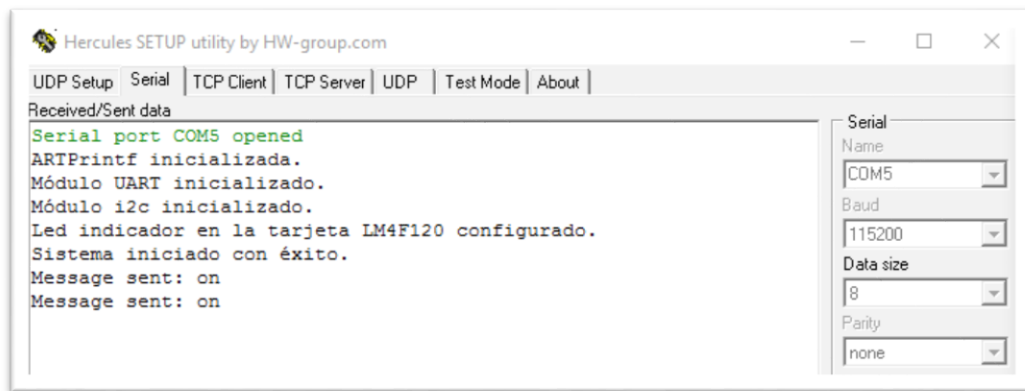


Ilustración 9. Consola de información despliega la cadena enviada al esclavo.

Por ultimo al enviar la cadena "off" el led de la tarjeta MSP430 se apaga.

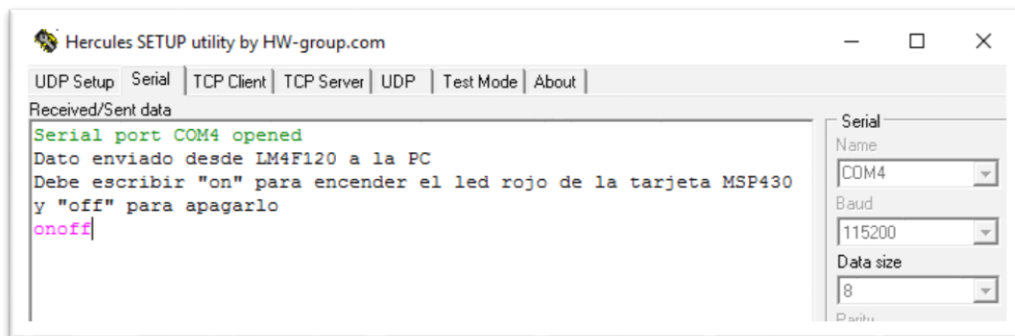


Ilustración 10. Envío de cadena "off".

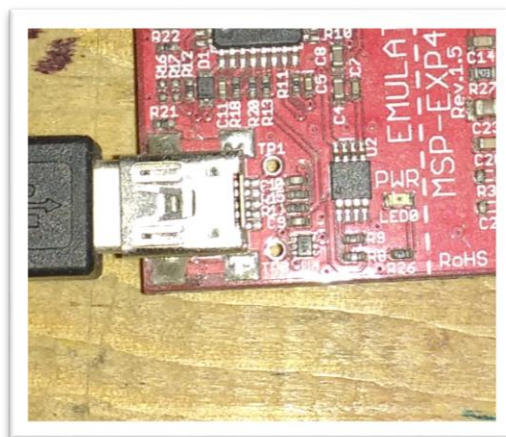


Ilustración 11. LED de la tarjeta MSP430 apagado.

Por último en la consola de información se despliega la cadena enviada.

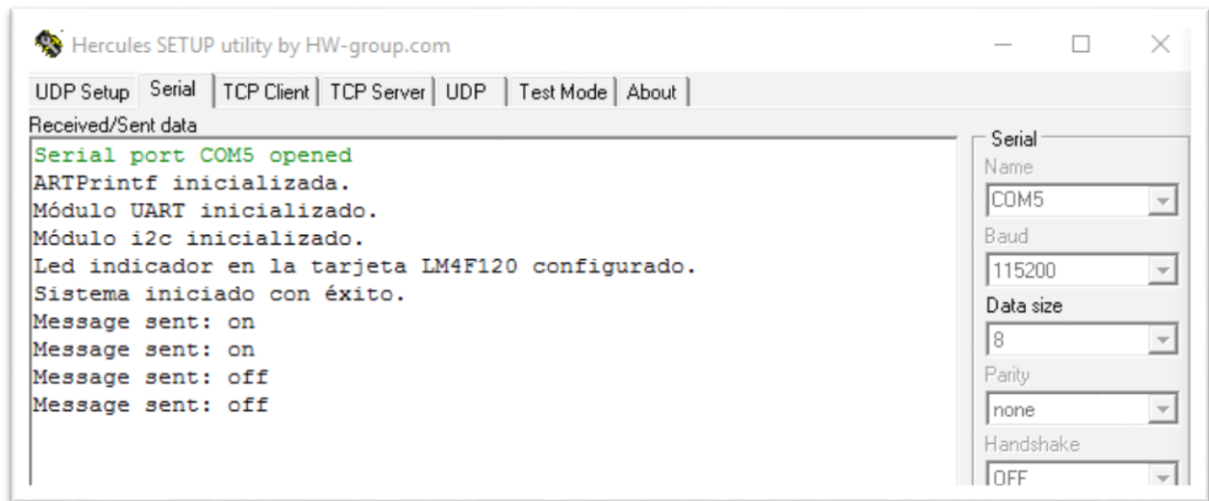


Ilustración 12. Despliegue de la cadena "off" en consola de información.

Código del sistema maestro

A continuación se muestra las librerías necesarias para los módulos utilizados en el programa junto con las variables globales, algo destacable de este segmento de código es la dirección con al que se identifica el esclavo para la transferencia de datos.

```
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_i2c.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/i2c.h"
#include "driverlib/interrupt.h"
#include "utils/uartstdio.h"

#define SLAVE_ADDRESS 0x48

unsigned char tx;
unsigned char rx;
unsigned char dataFromUart[4];
int i;
```

Método InitConsole

En este método se hace una configuración básica de la uart 0, donde dicho modulo hace la función de una consola para mostrar los logs de información durante la ejecución del sistema haciendo uso de la librería uartstdio.

```
void InitConsole(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);

    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    UARTStdioInit(0);
}
```

Método blinkled

Como su nombre lo dice este método sirve para ejecutar un parpadeo en el led rojo de la tarjeta stellaris, el cual sirve para notificar al usuario cuando se ejecutó alguna acción en el sistema.

```
void blinkled(void)
{
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 2);
    SysCtlDelay(SysCtlClockGet()/30);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
}
```

Método InitUARTComm

Este método hace la configuración necesaria para usar el módulo UART 7 de la tarjeta Stellaris, en el cual se habilitan periféricos y pines a utilizar, así como la configuración para la comunicación serial y habilitación de interrupciones.

```
Void InitUARTComm(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART7);

    GPIOPinConfigure(GPIO_PE0_U7RX);
    GPIOPinConfigure(GPIO_PE1_U7TX);

    GPIOPinTypeUART(GPIO_PORTE_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    UARTConfigSetExpClk(UART7_BASE, SysCtlClockGet(), 115200,
UART_CONFIG_WLEN_8|UART_CONFIG_PAR_NONE|UART_CONFIG_STOP_ONE);

    IntEnable(INT_UART7);
}
```

```
    UARTIntEnable(UART7_BASE, UART_INT_RX | UART_INT_RT);  
}
```

Método sendDataUART

Este método sirve para hacer el envío de datos desde la tarjeta Stellaris hacia la PC por medio del modulo UART7.

```
void sendDataUART(unsigned char *data){  
    while(UARTBusy(UART7_BASE));  
    while(*data != '\0')  
        UARTCharPut(UART7_BASE, *data++);  
    blinkled();  
}
```

Método UARTIntHandler

El método UARTIntHandler es el handler que se dispara cuando una interrupción es generada al momento de recibir datos de la PC a la tarjeta Stellaris a través de la comunicación serial, cabe mencionar que en este handler se espera recibir las cadenas “on” o “off” para que inmediatamente el dato sea enviado al esclavo por medio de la comunicación I2C.

```
void UARTIntHandler(void)  
{  
    I2CMasterIntDisable(I2C2_MASTER_BASE);  
  
    int i = 0;  
    unsigned long ulStatus;  
    unsigned char *bufEcho = dataFromUart;  
  
    ulStatus = UARTIntStatus(UART7_BASE, true);  
  
    UARTIntClear(UART7_BASE, ulStatus);  
  
    while(UARTCharsAvail(UART7_BASE))  
    {  
        *bufEcho = UARTCharGetNonBlocking(UART7_BASE);  
        *bufEcho++;  
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);  
        SysCtlDelay(SysCtlClockGet() / (1000 * 3));  
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);  
        i++;  
    }  
    I2C_write(SLAVE_ADDRESS, dataFromUart, i);  
    I2CMasterIntEnable(I2C2_MASTER_BASE);  
    UARTprintf("Message sent: %s\n", dataFromUart);  
}
```

Como parte de la configuración del proyecto es necesario hacer la configuración del handler UARTIntHandler en el archivo “lm4f120h5qr_startup_ccs.c”, agregando las siguientes líneas:

```
#include <stdint.h>

//*****
//
// Forward declaration of the default fault handlers.
.
.
.

// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****
extern void _c_int00(void);
extern void UARTIntHandler(void);

.
.
.

IntDefaultHandler,          // UART5 Rx and Tx
IntDefaultHandler,          // UART6 Rx and Tx
    UARTIntHandler,         // UART7 Rx and Tx

.
.
.
```

Módulo I2C

La función initI2C hace la configuración inicial del módulo I2C_2 de la tarjeta Stellaris, a continuación se listan los métodos utilizados en esta función:

- **“SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C2)”** habilita el módulo I2C_2.
- **“SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE)”** habilita el puerto E para el uso del módulo I2C_2.
- **“GPIOPinTypeI2CSCL(GPIO_PORTE_BASE, GPIO_PIN_4)”** habilita el pin E4 para ser usado con el módulo I2C_2.
- **“GPIOPinTypeI2C(GPIO_PORTE_BASE, GPIO_PIN_5)”** habilita el pin E5 para ser usado con el módulo I2C_2.
- **“GPIOPinConfigure(GPIO_PE4_I2C2SCL)”** configura el pin E4 como SCL.
- **“GPIOPinConfigure(GPIO_PE5_I2C2SDA)”** configure el pin E5 como SDA.
- **“I2CMasterInitExpClk(I2C2_MASTER_BASE, SysCtlClockGet(), false)”** configure el módulo I2C como maestro con una frecuencia de reloj de 100

KHz, usando como frecuencia base los 16 MHz que se establecen en el main del programa.

- **"I2CMasterEnable(I2C2_MASTER_BASE)"** habilita el módulo I2C como maestro.
- **"IntEnable(INT_I2C2)"** habilita las interrupciones para el módulo I2C.
- **"I2CMasterIntEnable(I2C2_MASTER_BASE)"** habilita las interrupciones para el módulo en modo maestro.

```
void initI2C(void) {  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C2);  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);  
    GPIOPinTypeI2CSCL(GPIO_PORTE_BASE, GPIO_PIN_4);  
    GPIOPinTypeI2C(GPIO_PORTE_BASE, GPIO_PIN_5);  
    GPIOPinConfigure(GPIO_PE4_I2C2SCL);  
    GPIOPinConfigure(GPIO_PE5_I2C2SDA);  
    I2CMasterInitExpClk(I2C2_MASTER_BASE, SysCtlClockGet(), false);  
    I2CMasterEnable(I2C2_MASTER_BASE);  
    IntEnable(INT_I2C2);  
    I2CMasterIntEnable(I2C2_MASTER_BASE);  
}
```

A continuación se muestra la función que hace la transmisión de datos a través del bus I2C, esta función recibe los siguientes parámetros:

- slave_address es la dirección del esclavo al que se hará la transferencia del dato.
- *data es la cadena a enviar.
- Length es el tamaño de la cadena que será enviada.

Los métodos utilizados de la librería correspondiente al módulo I2C son los siguientes:

- **"I2CMasterSlaveAddrSet(I2C2_MASTER_BASE, slave_address, false)"** configura la dirección del esclavo al cual se hará la transferencia, cabe mencionar que para indicar que será un dato de salida se debe pasar el tercer parámetro como "false".
- **"I2CMasterBusy(I2C2_MASTER_BASE)"** verifica si el bus está ocupado.
- **"I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_START)"** prepara el bus para la transferencia de datos.
- **"I2CMasterDataPut(I2C2_MASTER_BASE, *(data+i))"** agrega el dato en el bus de transferencia.
- **"I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_CONT)"** hace el envío del dato.
- **"I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH)"** indica que se terminó la transferencia de datos.

```

void I2C_write(unsigned char slave_address, unsigned char *data, unsigned long length)
{
    volatile int i = 0;

    // Inicia la transmisión i2c.
    I2CMasterSlaveAddrSet(I2C2_MASTER_BASE, slave_address, false);
    I2CMasterDataPut(I2C2_MASTER_BASE, *(data));
    while(I2CMasterBusy(I2C2_MASTER_BASE)) {} // Check, the bus isn't busy (low?)

    // Se configura para la transmisión de múltiples bytes.
    I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_START);

    // Espera mientras el bus está ocupado.
    while(I2CMasterBusy(I2C2_MASTER_BASE)) {}

    // Hace la transmisión de los datos.
    for(i=1; i < length; i++) {
        I2CMasterDataPut(I2C2_MASTER_BASE, *(data+i));
        I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_CONT);
        while(I2CMasterBusy(I2C2_MASTER_BASE)) {}
    }

    // Termina la transmisión de datos.
    I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);
    while(I2CMasterBusy(I2C2_MASTER_BASE)) {}
}

```

La función que se utiliza para hacer la lectura de datos es I2C_read la cual recibe como parámetros:

- slave_address es la dirección del esclavo al que se hará la transferencia del dato.
- *dataRx es la el apuntador donde se almacenaran los datos recibidos.
- Length es el tamaño de la cadena que será recibida.

En esta función se utilizan los siguientes métodos del módulo I2C:

- “**I2CMasterSlaveAddrSet(I2C2_MASTER_BASE, slave_address, true)**” configura la dirección del esclavo desde donde se recibirán los datos, cabe mencionar que para indicar que será un dato de entrada se debe pasar el tercer parámetro como “true”.
- “**I2CMasterBusy(I2C2_MASTER_BASE)**” verifica si el bus está ocupado.
- “**I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_START)** prepara el bus para la recepción de datos.
- “**I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_CONT);**” obtiene los datos del bus.
- “**I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_FINISH)**” indicar que se ha terminado la lectura de los datos de entrada.

```

void I2C_read(unsigned char slave_address, unsigned char *dataRx, unsigned long length)
{
    volatile int i = 0;
    I2CMasterSlaveAddrSet(I2C2_MASTER_BASE, slave_address, true);
    I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_START);
    while(I2CMasterBusy(I2C2_MASTER_BASE)) {}
    *dataRx = I2CMasterDataGet(I2C2_MASTER_BASE);
    for(i = 1; i < (length-2); i--) {
        I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_CONT);
        while(I2CMasterBusy(I2C2_MASTER_BASE)) {}
        *dataRx = I2CMasterDataGet(I2C2_MASTER_BASE);
    }
    I2CMasterControl(I2C2_MASTER_BASE, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);
    while(I2CMasterBusBusy(I2C2_MASTER_BASE)) {}
    *dataRx = I2CMasterDataGet(I2C2_MASTER_BASE);
    if(rx == '1'){
        UARTprintf("Dato recibido desde el esclavo: %c", dataRx);
    }
}

```

Por último el handler `i2c_rx_handler` es disparado cuando se recibe algún dato desde el esclavo y este a su vez ejecuta la función `I2C_read` donde se hace la recepción de los datos cabe mencionar que cada que se dispara una interrupción es necesario limpiar el registro.

```

void i2c_rx_handler (void) {
    I2CMasterIntClear(I2C2_MASTER_BASE);
    I2C_read(SLAVE_ADDRESS, &rx, 1);
}

```

Cabe mencionar que para que funcione el handler de la interrupción de datos de entrada es necesario agregar el nombre del handler para que la tarjeta sepa que función disparar al momento de ejecutarse dicha interrupción.

```

//*****
//
// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****
extern void _c_int00(void);
extern void UARTIntHandler(void);
extern void i2c_rx_handler (void);
.
.
.
    0,                                // Reserved
    0,                                // Reserved
    i2c_rx_handler,                   // I2C2 Master and Slave
    IntDefaultHandler,                // I2C3 Master and Slave
.
.
.

```


Programa principal del sistema.

En el “main” del programa se hace la inicialización de variables, módulos y recursos necesarios para el funcionamiento del sistema, a continuación se listan las operaciones que se ejecutan en este trozo de código:

- Configura el reloj para que sea ejecutado directamente desde el cristal/oscilador externo, el cual tiene una frecuencia de 16 MHz.
- Indica el puerto y pines de salida para usar el led RGB de la tarjeta para indicar ciertas acciones del sistema.
- Inicializa los pines 1 y 2 del led RGB de la tarjeta.
- Configura la consola serial para desplegar mensajes de información durante la ejecución del sistema.
- Configura el módulo UART para la comunicación serial.
- Inicializa el módulo I2C.
- Habilita el interruptor maestro.
- Inicializa los pines 1 (led rojo) y 2 (led azul) del led RGB de la tarjeta.
- Enciende y apaga el led Azul como indicador de que el sistema se ha terminado de configurar correctamente.

```
int main(void){
    SysCtlClockSet(SYSCTL_SYSDIV_1|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN
);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2|GPIO_PIN_1);
    InitConsole();
    UARTprintf("UARTprintf inicializada.\n");
    InitUARTComm();
    UARTprintf("Módulo UART inicializado.\n");
    initI2C();
    UARTprintf("Módulo i2c inicializado.\n");
    IntMasterEnable();
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
    SysCtlDelay(SysCtlClockGet() / (1000 * 3));
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
    UARTprintf("Led indicador en la tarjeta LM4F120 configurado.\n");
    unsigned char *buf = "Dato enviado desde LM4F120 a la PC\n\n";
    sendDataUART(buf);
    while(UARTBusy(UART7_BASE));
    buf = "Debe escribir \"on\" para encender el led rojo de la tarjeta MSP430 y
\"off\" para apagarlo\n";
    sendDataUART(buf);
    UARTprintf("Sistema iniciado con éxito.\n");
    while(1)
    {
    }
}
```

Código del esclavo para el sistema i2c

Para el esclavo del sistema de comunicación i2c se utilizó una tarjeta LAUNCHPAD MSP430, a continuación se mostrara el código correspondiente al esclavo.

```
#include "msp430g2553.h"
void receive_cb(unsigned char receive);
void transmit_cb();
void start_cb();
unsigned char flag=0;
unsigned int flag1=0;
unsigned char PRxData[3];           // Pointer to RX data
unsigned char RXByteCtr;
volatile unsigned char RxBuffer[128]; // Allocate 128 byte of RAM
unsigned char TXData = 0;
unsigned char TXByteCtr;
unsigned char data;
unsigned char ok_success[2] = {'0','k'};

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;           // Stop WDT
    P1DIR |= BIT0;

    P1SEL |= BIT6 + BIT7;               // Assign I2C pins to USCI_B0
    P1SEL2 |= BIT6 + BIT7;              // Assign I2C pins to USCI_B0

    UCB0CTL1 |= UCSWRST;                 // Enable SW reset
    UCB0CTL0 = UCMODE_3 + UCSYNC;         // I2C Slave, synchronous mode
    UCB0I2COA = 0x48;                    // Own Address is 048h
    UCB0CTL1 &= ~UCSWRST;                 // Clear SW reset, resume operation
    UCB0I2CIE |= UCSTPIE + UCSTTIE;      // Enable STT and STP interrupt
    IE2 |= UCB0RXIE + UCB0TXIE;          // Enable RX and TX interrupt

    TXData = 0x00;                       // Holds TX data

    _EINT();
    BCSCTL1 = CALBC1_16MHZ;
    DCOCTL = CALDCO_16MHZ;
    LPM4;                                // read out the RxData buffer
}

void start_cb(){
    PRxData[0] = 0;
    flag1 = 0;
}

void receive_cb(unsigned char receive){
    PRxData[flag1] = receive;
    flag1++;

    if(strncmp(PRxData, "on", 2) == 0){
        P1OUT = 1;
        data = '1';
    }else if (strncmp(PRxData, "off", 3) == 0) {
        P1OUT = 0;
        data = '1';
    }
}
```

```

void transmit_cb(){
    if (data == '1') {
        UCB0TXBUF = data;
        data = '0';
    }else{
        UCB0TXBUF = data;
        flag++;
    }
}

// USCI_B0 Data ISR
#pragma vector = USCIAB0TX_VECTOR
__interrupt void USCIAB0TX_ISR(void)
{
    if (IFG2 & UCB0TXIFG){
        transmit_cb();
    }
    else{
        receive_cb(UCB0RXBUF);
    }
}

// USCI_B0 State ISR
#pragma vector = USCIAB0RX_VECTOR
__interrupt void USCIAB0RX_ISR(void)
{
    UCB0STAT &= ~(UCSTPIFG + UCSTTIFG);
    start_cb();
}

```