Michael Alrøe, ma@ase.au.dk

Exercise: Intercomponent Busses

Purpose

The main purpose of the exercise is to understand the basic principles for commonly used inter-component busses, and to be able to use these busses in practice.

The I2C bus is examined by interfacing the temperature sensor LM75, and by using one PSoC as an I2C master device and one as an I2C slave device.

The SPI is examined by interfacing 2 PSoC devices, one as Master and the other as Slave.

The exercise should end up with a small journal holding essentials about the exercise. The essentials might be electrical wiring, oscilloscope/logic analyzer dumps, parts of source code, and results.

Literature

- Data sheet for LM75.
- PSoC Manuals.

The relevant documents can be downloaded from Blackboard.

I2C

In this part of the exercise ("I2C"), we will use the circuit below:

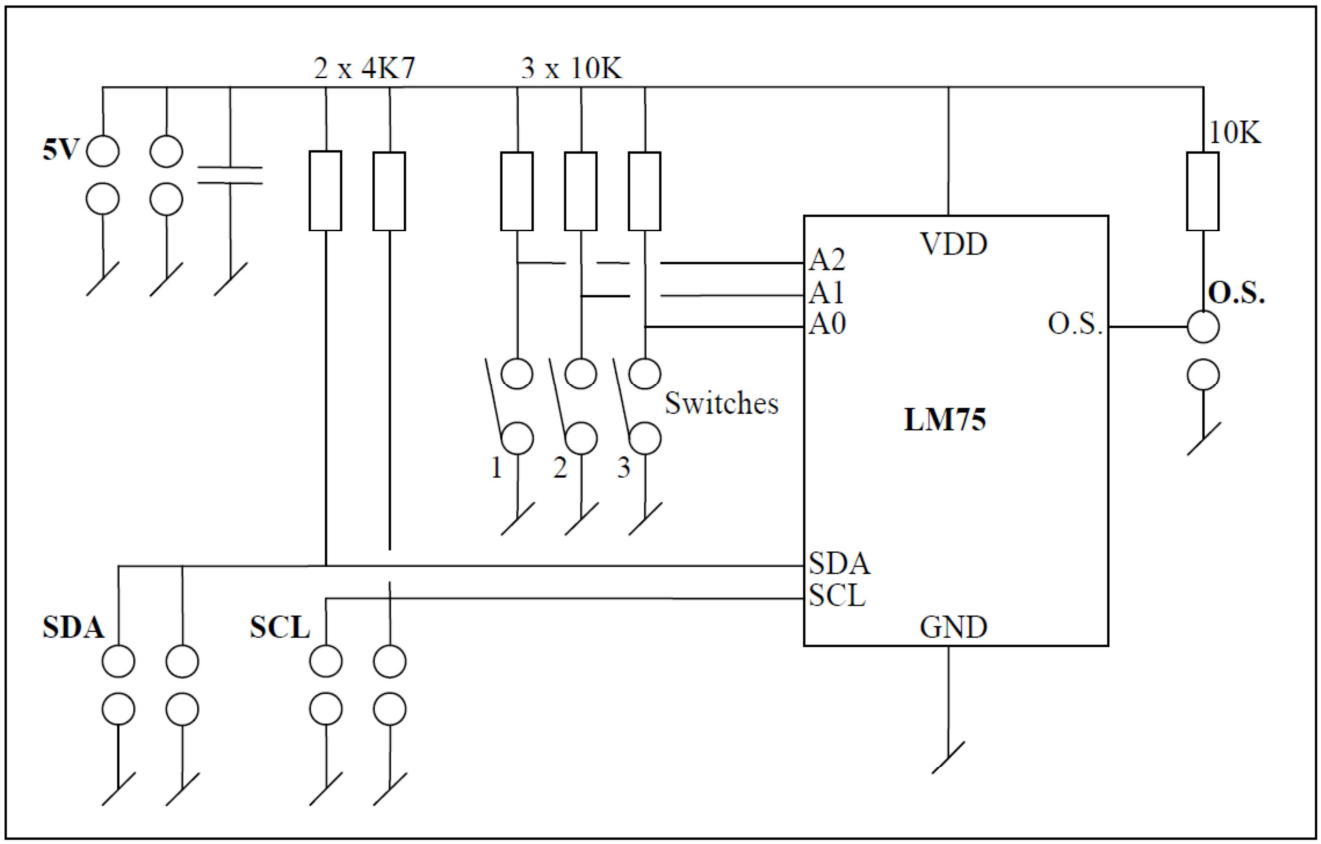

Figure 1 I2C Circuit

A certain number of these LM75 I2C slaves are available.

5 volt: These connectors are for power supply (from PSoC) and for connecting power to the next slave unit (if any). Note that the LM75 might accept both 3.3V and 5V supply voltages

SDA and SCL: These connectors are for the I2C bus (connect to the corresponding I2C port pin at PSoC) and to the next slave unit (if any).

O.S.: This connector is the alarm output from the LM75. It might optionally be connected to an interrupt input at the PSoC.

The 3 switches are used for setting up the LM75 local I2C address (A2-A0).

Notice: Setting a switch ON, sets the corresponding address input LOW.

1. Exercise I2C: PSoC Master and LM75 Slave

Use the PSoC Creator to define a new empty project. Insert an I2C Master into the project.

Connect the SCL and SDA to some external pins on the PSoC.

Write and test the LM75 interfacing using a C program (see hints!) inside the PSoC Creator by reading the temperature from at least 2 slaves at the I2C bus. Implement an endless loop, where you read the temperatures. Test different local I2C addresses.

You might start reading the temperature from just 1 LM75 device, and then extend with more LM75 devices afterwards.

Make a conversion of the bits received to a variable holding real temperature value.

Timing diagram below is from the LM75 datasheet:

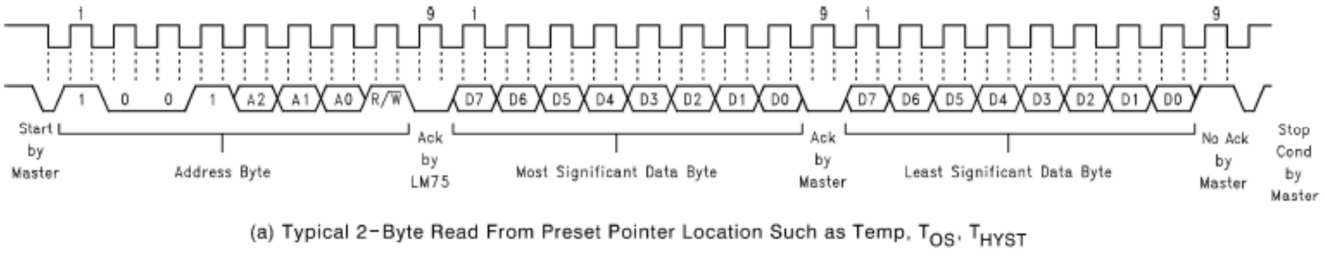

Figure 2 LM 75 timing diagram, 2-byte read

Connect an oscilloscope/logic analyzer to the SDA and SCL lines and verify timing diagram for the LM75.

Optional: You might also do experiments using the alarm features of the LM75.

Hints:

```
#define LM75_BASE_ADDRESS 0x48
uint8 buffer[2];
uint8 status;
```

The exercise might be solved with the following commands:

```
I2C_1_Start();
I2C_1_I2CMasterClearStatus(); /* Clear any previous status */

status = I2C_1_I2CMasterSendStart(LM75_BASE_ADDRESS, 1); /* Read! */
if (status == I2C_1_I2C_MSTR_NO_ERROR) /* Check if transfer completed without errors */
{
              buffer[0] = I2C_1_I2CMasterReadByte(I2C_1_I2C_ACK_DATA);
              buffer[1] = I2C_1_I2CMasterReadByte(I2C_1_I2C_NAK_DATA);

              // Handle 2's complement!
              .
              .
              .
}
I2C_1_I2CMasterSendStop(); /* Send Stop */
```

Figure 3 I2C source code (partly)

Extract from datasheet:

"Note 8: The conversion-time specification is provided to indicate how often the temperature data is updated. The LM75 can be accessed at any time and reading the Temperature Register will yield result from the last temperature conversion. When the LM75 is accessed, the conversion that is in process will be interrupted and it will be restarted after the end of the communication. Accessing the LM75 continuously without waiting at least one conversion time between communications will prevent the device from updating the Temperature Register with a new temperature conversion result. Consequently, the LM75 should not be accessed continuously with a wait time of less than 300 ms."

Above means that there must be a delay on min. 300 ms after the STOP sequence has been sent until a new START sequence is issued in order to have the LM75 updating the temperature register. Otherwise you might read identical temperature values at each reading.

A delay might be implemented by the CyDelay() command.

2. Exercise SPI: PSoC Master and PSoC Slave

Use 2 PSoCs in combination. Have one PSoC act as SPI Master, and have another act as PSoC Slave. Connect the master and the slave to each other. Write a program that sends data from the master to the slave, and reads data from the slave.

Verify the signals with oscilloscope/logic analyzer.

You might find some inspiration below in order to solve above exercise:

I made a strategy that on the SPI slave I want to be notified by an interrupt each time some data arrives. I just read the received data inside the interrupt service routine (ISR), and at end of the ISR is the last received byte put into the TX buffer. The data inside the TX buffer is going to be sent at the next SPI transaction.

The SPI master is just going to send and read a byte at each transaction. The SPI master is going to use a polling strategy for verifying that the SPI transaction has completed. The SPI transactions are running in a loop, having a 1000 ms delay.

SPI Slave

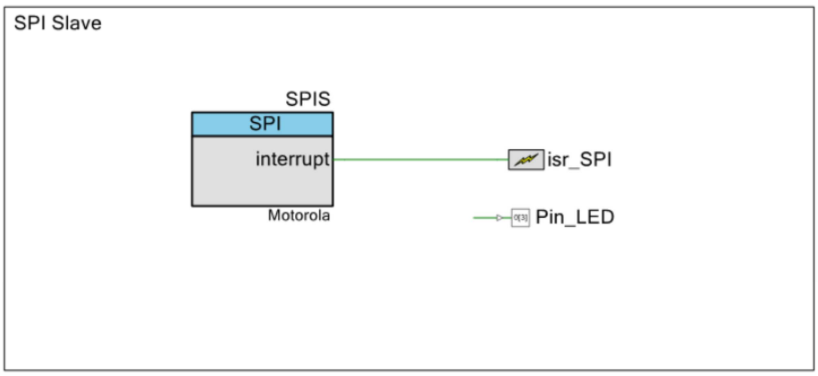

Figure 4 SPI Slave TopDesign

The isr_SPI component is configured “DERIVED”. The Pin_LED is just used for debugging toggling at each new interrupt received.

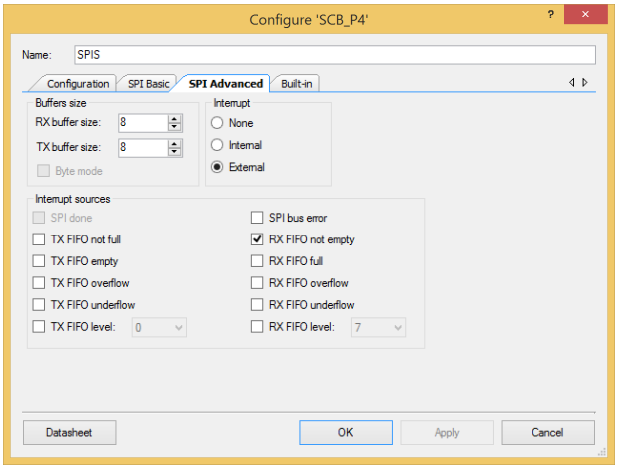

Figure 5 Configuring SPI Slave Interrupt

Next pins have to be assigned and project must be built in order to it synthesized and generated C API files. Source code is entered after successfully built of project. I recommend having the ISR inside the main.c file. However this means that you manually have to uncomment the system generated ISR at each rebuild!

```
/* ========================================
 *
 * Copyright
 * Aarhus University School of Engineering
 * All Rights Reserved
 *
 * Michael Alrøe
 *
 * ========================================
*/
#include <device.h>

/*******************************************************************************
* Function Name: isr_SPI_Interrupt
********************************************************************************
*
* Summary:
*   The default Interrupt Service Routine for isr_SPI.
*
* Parameters:
*   None
*
* Return:
*   None
*
* Remember to delete ISR routine in isr_SPI.C file on rebuild!!!
*******************************************************************************/
CY_ISR(isr_SPI_Interrupt)
{
    uint8 i, j, buf[8];
    uint32 source = 0u;

    // Check if any data available.
    j = SPIS_SpiUartGetRxBufferSize();

    // SPI Read data from SPIS RX software buffer to buf
    for(i=0u; i<j; i++)
    {
        buf[i] = SPIS_SpiUartReadRxData();
    }

    if (j>0)  // Process data received....
    {
        // Send last command back to master at next transaction!
        SPIS_SpiUartWriteTxData(buf[j-1]);
    }

    Pin_LED_Write(!Pin_LED_Read());  // Debug -> Toggle LED!

    // Clear Rx Interrupt Source
    source = SPIS_GetRxInterruptSourceMasked();
    SPIS_ClearRxInterruptSource(source);
}

int main()
{
    CyGlobalIntEnable;

    isr_SPI_Start();
    SPIS_Start();

    for(;;)
    {
        // Do nothing!!!
    }
}

/* [] END OF FILE */
```

Figure 6 SPI Slave main.c

Note that the PSoC 4 hardware blocks make use of a write-one-to-clear interrupt methodology. This means that that the interrupt flag has to be cleared manually inside the ISR.

The `GetRxInterruptSourceMasked` function is returning the source interrupt, which is stored inside the `source` variable. Then the `ClearRxInterruptSource` function is trying to clear the interrupt flag if the RX buffer is empty, otherwise the flag remains. Read more inside the SPI Component datasheet in the section for the `ClearRxInterruptSource` function description.

SPI Master

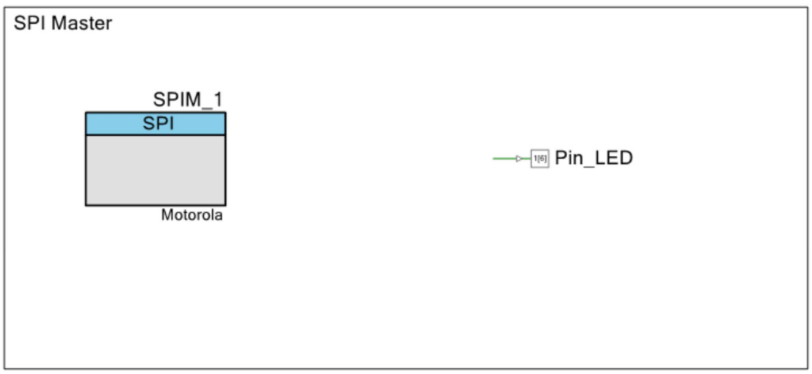

Figure 7 SPI Master TopDesign

The Pin_LED is just used for debugging toggling at each new cycle.

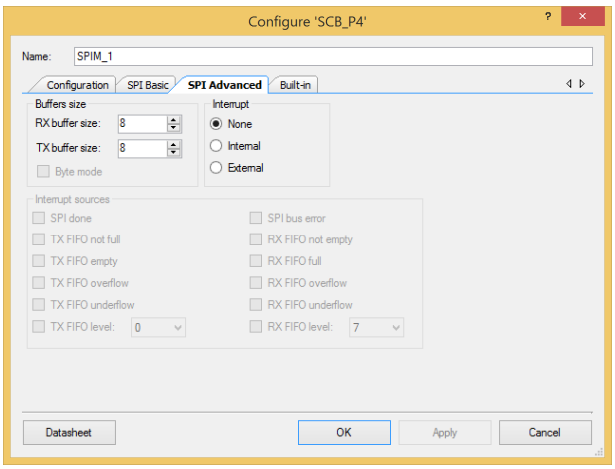

Figure 8 Configuring SPI Master

Next pins have to be assigned and project must be built in order to it synthesized and generated C API files. Source code is entered after successfully built of project.

```
/* ========================================
 *
 * Copyright
 * Aarhus University School of Engineering
 * All Rights Reserved
 *
 * Michael Alrøe
 *
 * ========================================
*/
#include <project.h>

int main()
{
            uint8 TxCmd, RxCmd;

    CyGlobalIntEnable;

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    SPIM_1_Start();

    TxCmd=0;
    for (;;)
    {
      // SPI Write
      SPIM_1_SpiUartWriteTxData(TxCmd++);

      // Wait for SPI to complete
      while(0u == (SPIM_1_GetMasterInterruptSource() & SPIM_1_INTR_MASTER_SPI_DONE))
      {
        /* Wait while Master completes transaction -> Polling for SPI_DONE flag! */
      }

      // SPI Read.
      RxCmd = SPIM_1_SpiUartReadRxData();

      // Process RxCmd..............

      Pin_LED_Write(!Pin_LED_Read());  // Debug -> Toggle LED!

      CyDelay(1000);
    }
    return 0;
}

/* [] END OF FILE */
```

Figure 9 SPI Master main.c

3. Exercise I2C: PSoC Master and PSoC Slave (Optional)

Use one PSoC as an I2C master, and use another PSoC as an I2C slave device. Connect the SCL and SDA between the PSoC devices. Make sure to handle the pull-up resistors!

Make experiments with read and write transactions between the master and slave devices.

Connect an oscilloscope/logic analyzer to the SDA and SCL lines and verify timing between the devices.