Application Report

SLAA703–May 2016

Software I^2C on MSP430 MCUs

Timothy Logan *MSP Applications*

ABSTRACT

Hardware and layout limitations often require smaller package sizes that could potentially create a tradeoff for the number of serial peripherals available to the user. With the I^2C protocol, special software can be created to use a pair of simple GPIO ports to emulate an I^2C master device. This allows programmers to be flexible with pin assignments and enables smaller package devices to overcome the hardware limitation of scarce number of hardware I^2C peripherals. Having a software-emulated I^2C master enables programmers to communicate effectively and fully control attached slave devices without excessive overhead. In addition to the basic I^2C functionality, several advanced features such as clock stretching and NAK detection are also supported by the software included in this application report.

Source code and other collateral discussed in this application report can be downloaded from http://www.ti.com/lit/zip/slaa703.

Contents

List of Figures

MSP430, MSP432, Code Composer Studio are trademarks of Texas Instruments.
IAR Embedded Workbench is a registered trademark of IAR Systems AB.
All other trademarks are the property of their respective owners.

1 Introduction

Hardware and layout limitations often require smaller package sizes that could potentially create a tradeoff for the number of serial peripherals available to the user. With the I^2C protocol, special software can be created to use a pair of simple GPIO ports to emulate an I^2C master device. This allows programmers to be flexible with pin assignments and enables smaller package devices to overcome the hardware limitation of scarce number of hardware I^2C peripherals. Having a software-emulated I^2C master enables programmers to communicate effectively and fully control attached slave devices without excessive overhead. In addition to the basic I^2C functionality, several advanced features such as clock stretching and NAK detection are also supported by the software included in this application report.

The implementation of the software in this application report was made to be highly generic and easily scaled to other MSP430™ and MSP432™ MCUs. The core code is written in bare-metal C and requires only a hardware timer for operation. In the provided source example, the highly accessible and abundantly available Timer_A peripheral is used to precisely handle all of the timing requirements of the I^2C master. If a timer other than Timer_A is required, the necessary changes that are needed to port to a different architecture are abstracted out to allow high portability and quick integration.

2 Use Cases

The main use case for having a software I^2C master is to overcome the limited number of hardware I^2C peripherals that arises from having a smaller package device. For example, a simple example of an MSP430FR5738 connected to an LDC1614 inductive sensor is an ideal use case. The MSP430FR5738 device in a DSBGA package has only one hardware I^2C peripheral available on the device. In applications that involve consumer electronics, the need for two I^2C peripherals (one to be a slave to a host application processor and one to be a master to a sensor) is required. These types of applications also require a small package size. Figure 1 shows the hardware configuration for this scenario.

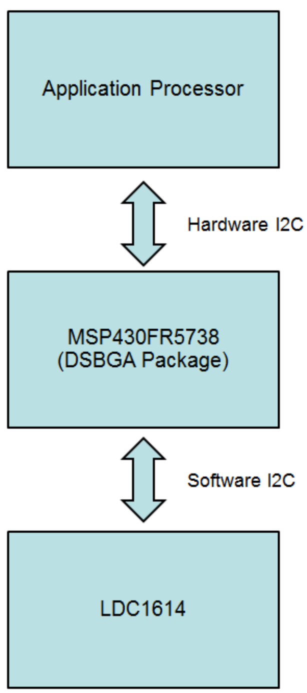

Figure 1. Sample Hardware Configuration

In this example, the MSP430 MCU acts as a buffer between the LDC1614 and the application processor. This sort of scenario is very common in applications such as smart phones and tablets where a low-power centric microprocessor such as the MSP430 MCU is required to relay sensor information to the application processor. The hardware I^2C peripheral of the MSP430 MCU is used as an I^2C slave to the application processor and the software-emulated I^2C is used as a master to the LDC1614.

3 Configuration

The entirety of the configuration parameters is stored within the *msp430_swi2c_master.h* header file. There are two different configuration settings that the user should be concerned about: the pin configurations and the timer period setting. The pin configurations control which pins on the MSP430 MCU are used for the software I^{2}C master. Figure 2 shows a sample configuration.

```
/* Pin Definitions. These  should be changed depending on the device that
 * you are using.
 */
#define SWI2C_SCL        BIT7
#define SWI2C_SDA        BIT6
#define SWI2C_PxDIR      P1DIR
#define SWI2C_PxOUT      P1OUT
#define SWI2C_PxIN       P1IN
#define SWI2C_SDA_LOW    SWI2C_PxDIR |= SWI2C_SDA
#define SWI2C_SCL_LOW    SWI2C_PxDIR |= SWI2C_SCL
#define SWI2C_SCL_HIGH   SWI2C_PxDIR &= ~SWI2C_SCL
#define SWI2C_SDA_HIGH   SWI2C_PxDIR &= ~SWI2C_SDA
```

Figure 2. Pin Configuration

As can be seen, this example sets P1.6 and P1.7 as I^{2}C SDA and I^{2}C SCL, respectively. Any ordinary GPIO can be used, and there are no special requirements such as interrupt capability. It is important to note that the configured pins should be reserved exclusively for I^{2}C operation. If another peripheral or function uses or reconfigures the selected pins for a different purpose, the behavior will be unreliable. The second configuration parameter is the parameter that controls the timer. This timer is used to generate the I^{2}C clock frequency. This parameter relies on the clock frequency of SMCLK. Figure 3 shows the formula to derive the desired I^{2}C clock frequency.

```
I2C Data Rate =     SMCLK Frequency
                    _______________

                    2 * TimerPeriod
```

Figure 3. Formula to Derive I^{2}C Clock Frequency

For example, assume that a 100-kHz clock frequency is required, and that the SMCLK frequency is 24 MHz. Putting this information into the formula in Figure 3, a value of 120 must be set for *SWI2C_TIMER_PERIOD*. The main limitation to this software solution is the maximum data rate that can be obtained on the I^{2}C bus. Using the MSP430FR5738 and LDC1614 combination previously described, a maximum of 200 kHz was achieved. This limitation can vary from device to device, and experimentation should be done on the target MSP MCU to discover the ceiling for the device in use.

When all of the configuration settings are set, call the *SWI2C_initI2C function* to persist the settings and configure the pins. This function not only sets the specified pins to the correct direction and output values, but also configures the hardware timer that is used for frequency generation. Note that whenever this function is called, all settings are reset and the software I^{2}C master is returned to the base default settings.

4 Simple Transactions

Transactions using the emulated software I^{2}C master are modeled largely after the same programming model that users might find on host systems such as Linux or UEFI. A simple transaction structure is used to control the slave address, how many bytes are sent, how many bytes are read, transaction buffers, and whether or not to perform a repeated start. Figure 4 shows the structure definition.

```
/* Configuration structure for performing an I2C transaction */
typedef struct _SWI2C_I2CTransaction
{
    uint8_t            address;
    uint_fast16_t      numWriteBytes;
    uint8_t*           writeBuffer;
    uint_fast16_t      numReadBytes;
    uint8_t*           readBuffer;
    bool               repeatedStart;
} SWI2C_I2CTransaction;
```

Figure 4. I^{2}C Transaction Configuration Structure

The address variable is the 7-bit slave address to perform the transaction on. Note that only 7-bit addresses are supported. The *numWriteBytes*, *writeBuffer*, *numReadBytes*, and *readBuffer* variables all contain the number of bytes to read or write and pointers to the source and destination buffers. After this structure is configured, a pointer to this structure is passed into the *SWI2C_performI2CTransaction* function, and the software manages the I^{2}C transaction. Note that the pointers being passed for reading data must be pre-allocated, and the software does not do any dynamic memory allocation or boundary checks. If a buffer is passed in for readBuffer that is too small to hold the number of read bytes, a buffer overflow will occur.

A variety of different I^{2}C transactions are supported with this software. When the *repeatedStart* variable is set to *false*, simple START/STOP I^{2}C transactions occur. Figure 5 shows this sort of transaction.

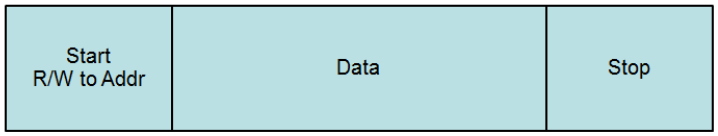

Figure 5. Simple I^{2}C Transaction

Note that if both write and read data is given to the transaction and the *repeatedStart* variable is false, two separate transactions occur on the bus, each having their own I^{2}C START and STOP. If *repeatedStart* is set as true, a repeated start transaction is performed (see Figure 6).

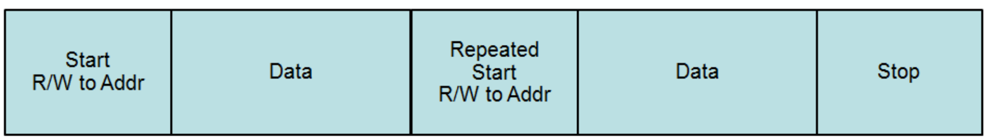

Figure 6. Repeated Start I^{2}C Transaction

Figure 7 shows a simple sample source snippet that configures the device, sets up the transaction, and executes a simple write command to an attached slave with address 0x48.

```
SWI2C_I2CTransaction myTransaction;
uint8_t myBuffer[5] = {1,2,3,4,5};

/* Initializing the master */
SWI2C_initI2C();

/* Setting up the transaction */
myTransaction.address = 0x48;
myTransaction.writeBuffer = myBuffer;
myTransaction.numWriteBytes = 5;

if(!SWI2C_performI2CTransaction(&myTransaction))
{
    /* Handle Error Code Here */
}

while(1);
```

Figure 7. Simple I^{2}C Write Transaction

The code included with this application report is meant to be generic and easily ported between multiple platforms and IDEs. This code compiles without issue on different IDEs such as Code Composer Studio™ IDE, IAR Embedded Workbench® IDE, and GCC.

5 Advanced Features

In addition to being able to perform simple I^{2}C transactions, the software I^{2}C master is also able to perform several advanced I^{2}C features. One of these features is the ability to detect when the slave device is holding the SCL line low in a "clock stretching" scenario. This is a technique that is done often by a slave device when it wants to stall the master and hold off from any additional transactions while it processes data. The software I^{2}C master can detect this condition and wait for the slave device to release the line before continuing with the transaction. Note that there is no time-out feature associated with the clock stretching, and the slave could conceivably cause the master to deadlock if care is not taken by the slave logic.

An additional extended feature of the software I^{2}C library is the ability for the master to detect when the slave device has sent a NAK condition over the I^{2}C line. A NAK condition can occur in several different situations. If the master tries to conduct a transaction on a slave address that does not exist on the I^{2}C bus, a NAK is returned. Additionally, if the master tries to perform a transaction and the slave is not yet ready for that transaction, the slave can NAK the attempt from the master. In the case of a NAK condition, the return value from the *SWI2C_performI2CTransaction* function is a Boolean false.

IMPORTANT NOTICE

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265