Application Report
SLAA368A–September 2007–Revised May 2009

Using the USI I²C Code Library

Priya Thanigai MSP430 Applications

ABSTRACT

This document serves as an overview of the master and slave code libraries for I²C communication using the USI module as found on the MSP430F20xx. The USI I²C master and slave libraries encapsulate all the functions necessary to transmit and receive multiple bytes. The functions are written in assembly and can be accessed by any C program that includes the required header files.

Related source files can be downloaded from http://www.ti.com/lit/zip/slaa368.

Contents

1 Introduction 2
2 I²C Master Library 2
3 I²C Slave Library 9
4 Code Size 13
5 References 13

List of Figures

1 Program Flow for Master Code Library 3
2 Program Flow for Slave Code Library 9

List of Tables

1 Code Size (IAR) 13

1 Introduction

The USI module provides the basic functionality needed to support synchronous serial communication. When configured in the I^{2}C mode, the USI module is an 8-bit shift register that can output a stream of serial data. With minimal software, it can be used to setup a master/slave relationship to implement serial communication. A software I^{2}C library is especially useful for lower-pin-count devices that do not include a hardware module dedicated to I^{2}C communication. The USI I^{2}C libraries offer the functionality needed to configure the MSP430 as either a master or a slave device capable of transmission and reception of multiple bytes. The library functions are capable of servicing interrupts while allowing I^{2}C transactions to take place in the background, without interfering with user applications. The definitions of the interrupt service routines, however, prevent the master and slave libraries from existing on the same device. The example files include a master MSP430 interface to a slave EEPROM device and to a slave MSP430 device.

Note: Internal pullup resistors are enabled on the MSP430F20xx to support I^{2}C communication.

2 I^{2}C Master Library

The I^{2}C master initiates data transfer and generates the clock signal SCL. It can be used in two modes:

- Master transmit
- Master receive

The master code library provides the necessary functionality to support multiple byte and word transmission and reception. It allows the user to switch between transmit and receive operations on the fly using a repeated start condition. The library can execute in both blocking and nonblocking modes. While operating in the nonblocking mode (callback function provided), the I^{2}C function returns immediately and communication takes place in the background. In this mode, the result of the operation (ACK/NACK) is passed as an argument to the callback function. In the blocking mode (no callback function provided), the functions in the library are executed exclusively (without parallel application code) and provide a valid return value in the end. The return value reflects the result of the operation. Also, the CPU operating mode can be changed or modified using callback functions defined at the application level. Figure 1 is a high-level flow diagram explaining the state transitions in an I^{2}C routine.

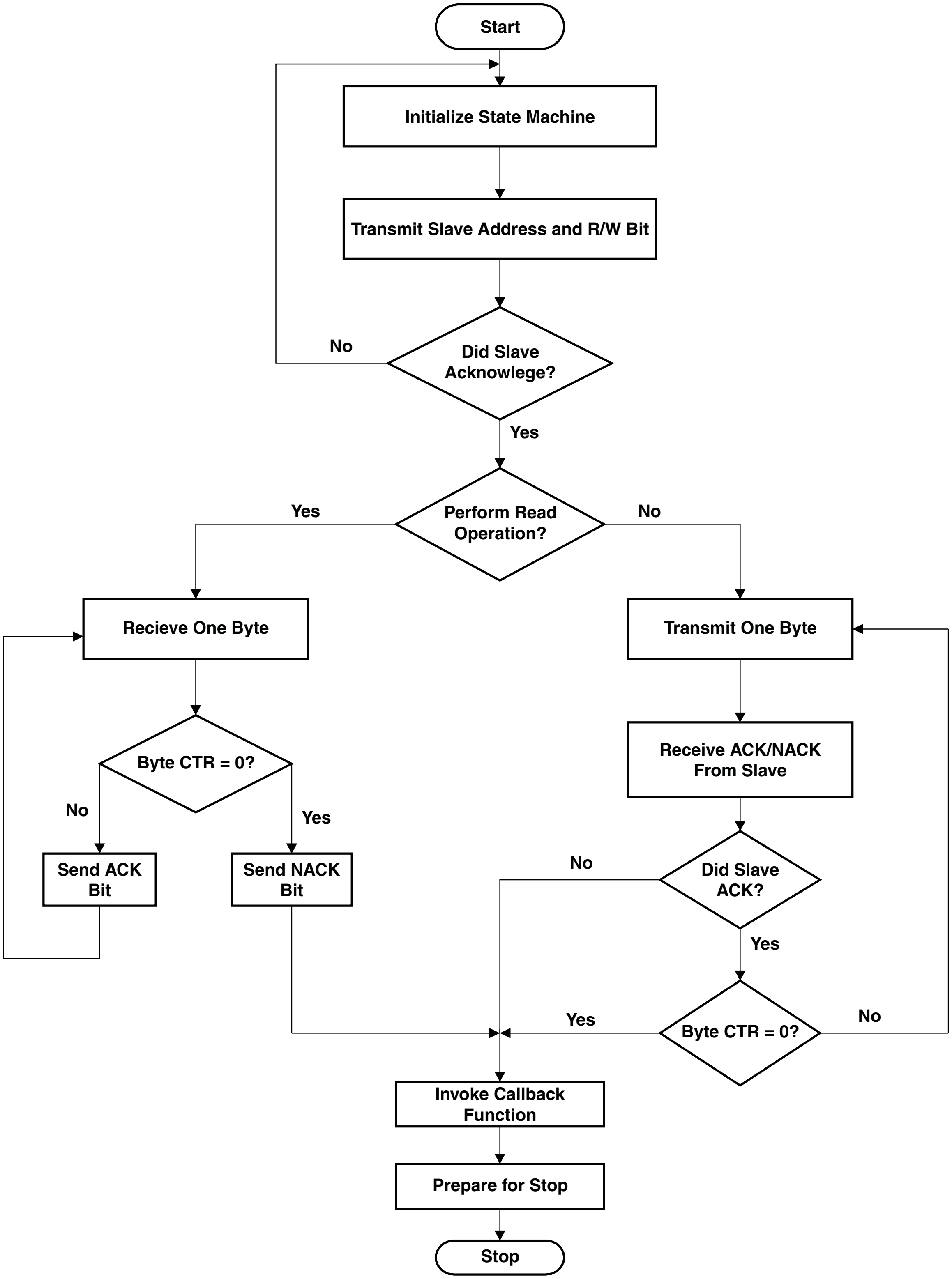

Figure 1. Program Flow for Master Code Library

2.1 Usage From C

```
Void main(void)
{

  WDTCTL = WDTPW+WDTHOLD;                   // Stop watchdog
  BCSCTL1 = CALBC1_1MHZ;                    // Load DCO constants
  DCOCTL = CALDCO_1MHZ;
  P1DIR |= 0x01;                            // P1.0 as output (LED)
  P1OUT = 0x00;

 /* Initialize USI module, clock ~ SMCLK/128 */
 TI_USI_I2C_MasterInit(USIDIV_7+USISSEL_2+USICKPL, StatusCallback);

  /* Acknowledge polling function - LED blinks continuously until slave device
     provides an ACK
 TI_USI_I2CSelect(unsigned char SlaveAddress) */

  while(TI_USI_I2CSelect(0x50))
  {
    P1OUT ^= 0x01;                          // Toggle LED
    for (i = 0; i < 0x3000; i++);           // Delay
  }
  P1OUT =0;                                 // Slave acknowledged, LED off
  /* Transmit data to the EEPROM device, prefixed by page address 0x01
   TI_USI_I2CWrite(SlaveAddress, Length, Multi, TxData) */

  __disable_interrupt();
  TI_USI_I2CWrite(0x50,9,0,TxData0);
  __bis_SR_register(LPM0_bits + GIE);

  /* Acknowledge polling function - loops continuously until slave device
  provides an ACK */
  while(TI_USI_I2CSelect(0x50));

  /* Transmit data to the EEPROM device, prefixed by page address 0x08
  TI_USI_I2CWrite(SlaveAddress, Length, Multi, TxData)*/

  __disable_interrupt();
  TI_USI_I2CWrite(0x50, 3, 0, TxData1);
  __bis_SR_register(LPM0_bits + GIE); //*/

  /* Acknowledge polling function - loops continuously until slave device
  provides an ACK */
  while(TI_USI_I2CSelect(0x50));

  /* Reset address counter of the EEPROM device by transmitting the page
  address to be read from (0x00) (Dummy write)
  TI_USI_I2CWrite(SlaveAddress, Length, Multi, TxData)*/

  __disable_interrupt();
  TI_USI_I2CWrite(0x50,1, 1,TxData0);
  __bis_SR_register(LPM0_bits + GIE);

  /* Read data from the EEPROM device, starting at page address 0x00
  TI_USI_I2CRead(SlaveAddress, Length, Multi, RxData)*/
  __disable_interrupt();
  TI_USI_I2CRead(0x50, 10, 1,RxData);
  __bis_SR_register(LPM0_bits + GIE);

  /* This function can be used to end any open I2C transaction.
  Use only if I2C transaction was left open previously by setting stop
  condition bit =1 */
  TI_USI_I2CStop();

  // check data for validity
  for (j = 0;j<10;j++)
  {
    if (RxData[j]!=j)
    {
      while(1);                             // data invalid, stay in loop
    }
  }
  P1OUT |= 0x01;                            // data valid, LED on
  while(1);                                 // program ends here
}

int StatusCallback(unsigned char c)
{
  return TI_USI_EXIT_LPM;                   // Exit active for next transfer
}
```

The file USI_I2CMaster.h must be included to initialize variables and functions when using the library from a C program. The TI_USI_I2C_MasterInit(…) function needs to be called only once at the start to initialize the USI module in master mode.

In the previous example (example_EEPROM.c), the MSP430 master transmits a stream of data to the AT 24C02, a two-wire serial EEPROM device. The device requires an 8-bit data word address following the device address before data bytes can be written to it. The internal word address counter of the EEPROM maintains the last address accessed during the previous data transmission. Hence, before a read, the internal counter needs to be reinitialized by transmitting to the required data word address. At the end of every write operation, the EEPROM enters an internally timed write cycle and does not respond until the write operation is complete. At this time, it can be polled by the master device for an acknowledge (ACK) using the TI_USI_I2CSelect function. This function returns a nonzero value if the slave sends a No Acknowledge (NACK). Once the slave has acknowledged, the data is ready to be read from the EEPROM. The transmitted and received data are compared and, if found to be valid, the LED is turned on. The data can also be viewed in the memory through a watch window. The callback function is used to modify the low-power mode for the next operation. The result of the previous operation is passed as an argument to the callback function. A zero indicates that the previous operation was successful. An additional example application to communicate with a dedicated slave MSP430 device is provided in the file example_master430interface.c.

Note: If C callback functions are used, the library needs to preserve additional CPU registers (C scratch registers) across the callback function call, as these registers could be affected by the callback function. If it can be ensured that the callback function does not modify the C scratch registers, they need not be preserved. Preserving the C scratch registers consumes additional code space and execution cycles. Therefore, the end user is given the option of using a predefined macro within the library to preserve the C scratch registers (USE_C_CONTEXT_SAVE is included) or leaving the registers as is (USE_C_CONTEXT_SAVE is commented out). See the compiler documentation for more details regarding C calling conventions.[2]

2.2 Usage From Assembly

```
RESET       mov.w   #SFE(CSTACK),SP        ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT
SetupDCO    mov.b   &CALBC1_1MHZ,&BCSCTL1  ; Load DCO calibration constants
            mov.b   &CALDCO_1MHZ,&DCOCTL
SetupPx     bis.b   #0x01,&P1DIR           ; P1.0& 1.1 as output
            bic.b   #0x01,&P1OUT
InitCall    mov.b   #0xEA,R12              ; USIDIV_7+USISSEL_2+USICKPL
            mov.w   #StatusCallback,R14    ; Callback fn. pointer
            call    #TI_USI_I2C_MasterInit ; Initialize USI master
SlaveDetect
            xor.b   #0x01,&P1OUT           ; Detect if slave is present...
            mov.w   #0xFFFF,R5             ; ... LED toggles until slave ACKS
L1          dec.w   R5
            jnz     L1
Poll_0      call    #AcknowledgePoll       ; Slave device present?
            cmp.b   #0,R12                 ; Is result zero (ACK)?
            jnz     SlaveDetect            ; Loop until device acknowledges
            clr.b   &P1OUT                 ; Slave acknowledged, LED off
TransmitCall_1
            mov.w   #0x50,R12              ; Slave address
            mov.w   #9,R14                 ; length = 9
            push.w  #Transmit1             ; pointer to data buffer
            push.w  #0x00                  ; generate stop
            call    #TI_USI_I2CWrite       ; transmit data stream #1
            add.w   #4,SP                  ; compensate SP on return
            bis.w   #LPM0+GIE,SR           ; enter LPM, enable interrupts
Poll_1      call    #AcknowledgePoll       ; Device ready for next transfer?
            cmp.b   #0,R12                 ; Is result zero (ACK)?
            jnz     Poll_1                 ; Loop until device acknowledges
TransmitCall_2
            mov.w   #0x50,R12              ; Slave address
            mov.w   #3,R14                 ; length = 3
            push.w  #Transmit2             ; pointer to data buffer
            push.w  #0x00                  ; generate stop
            call    #TI_USI_I2CWrite       ; transmit data stream #2
            add.w   #4,SP                  ; compensate SP on return
            bis.w   #LPM0+GIE,SR           ; enter LPM, enable interrupts
Poll_2      call    #AcknowledgePoll       ; Device ready for next transfer?
            cmp.b   #0,R12                 ; Is result zero (ACK)?
            jnz     Poll_2                 ; Loop until device acknowledges
TransmitCall_3
            mov.w   #0x50,R12              ; Slave address
            mov.w   #1,R14                 ; length = 1
            push.w  #Transmit1             ; pointer to data buffer
            push.w  #0x01                  ; generate stop
            call    #TI_USI_I2CWrite       ; transmit to reset internal counter
            add.w   #4,SP                  ; compensate SP on return
            bis.w   #LPM0+GIE,SR           ; enter LPM, enable interrupts
ReceiveCall
            mov.w   #0x50,R12              ; Slave address
            mov.w   #10,R14                ; length = 10
            push.w  #Receive               ; pointer to data buffer
            push.w  #0x01                  ; do not generate stop
            call    #TI_USI_I2CRead        ; receive stream#1,#2
            add.w   #4,SP                  ; compensate SP on return
            bis.w   #LPM0+GIE,SR           ; enter LPM, enable interrupts
```

```
StopTransaction
            call    #TI_USI_I2CStop         ; Stop I2C transactions
CheckResult
            mov.w   #Receive,R6             ; store received data location
            clr.b   R4
Compare     cmp.b   R4,0(R6)
            jz      Increment               ; data valid, continue
            clr.b   &P1OUT                  ; data invalid, LED off
            jmp     EndProgram
Increment
            inc.b   R4
            inc.w   R6
            cmp.b   #10,R4
            jnz     Compare                 ; if all 10 bytes are correct.
            bis.b   #0x01,&P1OUT            ; ... turn on LED
EndProgram  jmp     $                       ; program ends here
AcknowledgePoll
            mov.b   #0x50,R12               ; slave address
            call    #TI_USI_I2CSelect       ; Ack polling function
            ret
StatusCallback
            mov.w   #1,R12                  ; wake up on exit
            ret
```

In assembly, the usage of the library is the same as in C. Arguments are passed using the C calling convention (making the library functions compatible to both C and assembly usage). The first two parameters are passed using registers R12 and R14; all others are pushed onto the stack.[2]

2.3 Function Description

The following functions are defined in the master code library.

2.3.1 TI_USI_I2C_MasterInit(…)

The function initializes the USI module for I^2C communication. The parameters passed as arguments are:

unsigned char ClockConfig

This value contains the clock select, frequency divider, and clock polarity bits that are to be loaded onto the control register USICTL0.

int (* StatusCallback) (unsigned char)

This function is called on the transmission/reception of the last byte in the communication stream. The result of the previous operation (ACK/NACK) is passed as an argument to this function. The function returns a zero if the existing low-power mode should be maintained or a nonzero value if the low-power mode should be exited.

2.3.2 TI_USI_I2CSelect(…)

This function detects the presence of a slave by addressing it and determining if the slave responds with an ACK/NACK. The following parameter is passed as an argument:

unsigned char SlaveAddress

This is the device address of the slave.

The return value is the result of the operation. The function returns a zero if the slave is present and has acknowledged, and it returns a nonzero value if the slave is absent or not ready to acknowledge. For example, this function can be used to poll slower slave devices like EEPROM while waiting for it to complete an internally timed write cycle.

2.3.3 TI_USI_I2CRead(…)

This function implements block-read master receiver functionality. The parameters passed as arguments are:

unsigned char SlaveAddress

This is the device address of the slave.

unsigned int Length

This value is the number of bytes to be received.

unsigned char Multi

This variable is used to suppress the generation of I²C stop condition if the user wants to leave the channel open for further transactions.

void *RxData

This pointer variable points to the memory location where the received data should be stored.

The function returns a zero if the operation was successful and a nonzero value if a NACK was received at any time during the I²C transaction. It is valid only when no callback function is provided. Otherwise, the function returns immediately, and the I²C transaction takes place in the background.

2.3.4 TI_USI_I2CWrite(…)

This function implements block-write master transmitter functionality. The parameters passed as arguments are:

unsigned char SlaveAddress

This is the device address of the slave.

unsigned int Length

This value is the number of bytes to be transmitted.

unsigned char Multi

This variable is used to suppress the generation of I²C stop condition in case the user wants to leave the channel open for further transactions.

void *TxData

This pointer variable points to the memory location of the data to be transmitted.

The function returns a zero if the operation was successful and a nonzero value if a NACK was received at any time during the I²C transaction. It is valid only when no callback function is provided. Otherwise, the function returns immediately, and the I²C transaction takes place in the background.

2.3.5 TI_USI_I2CStop( )

This function generates the I²C stop condition and can be used to end an open I²C transaction. It is typically not used, as both transmit and receive functions generate a stop condition by default. It can be used in conjunction with a read or write function when the Multi parameter has a nonzero value.

2.4 Included Library Files

USI_I2CMaster.s43

This library file includes all the functionality necessary to transmit and receive single and multiple bytes as an I²C master device. It also contains the acknowledge polling function, which polls for the presence of a slave device.

USI_I2CMaster.h

This header file has the necessary definitions and functions for the master library and must be included in any C program that uses the library.

3 I^2C Slave Library

The I^2C slave does not control the clock and can be used in two modes:

- Slave transmit
- Slave receive

The slave code library allows for multiple byte transmission and reception. The library executes in a nonblocking manner and allows for the I^2C transactions to take place in the background. Callback functions are provided to allow the user to modify the CPU operating mode after each byte is transmitted or received and also store the received data for application use. Figure 2 is a high-level flow diagram explaining the state transitions in an I^2C slave routine.

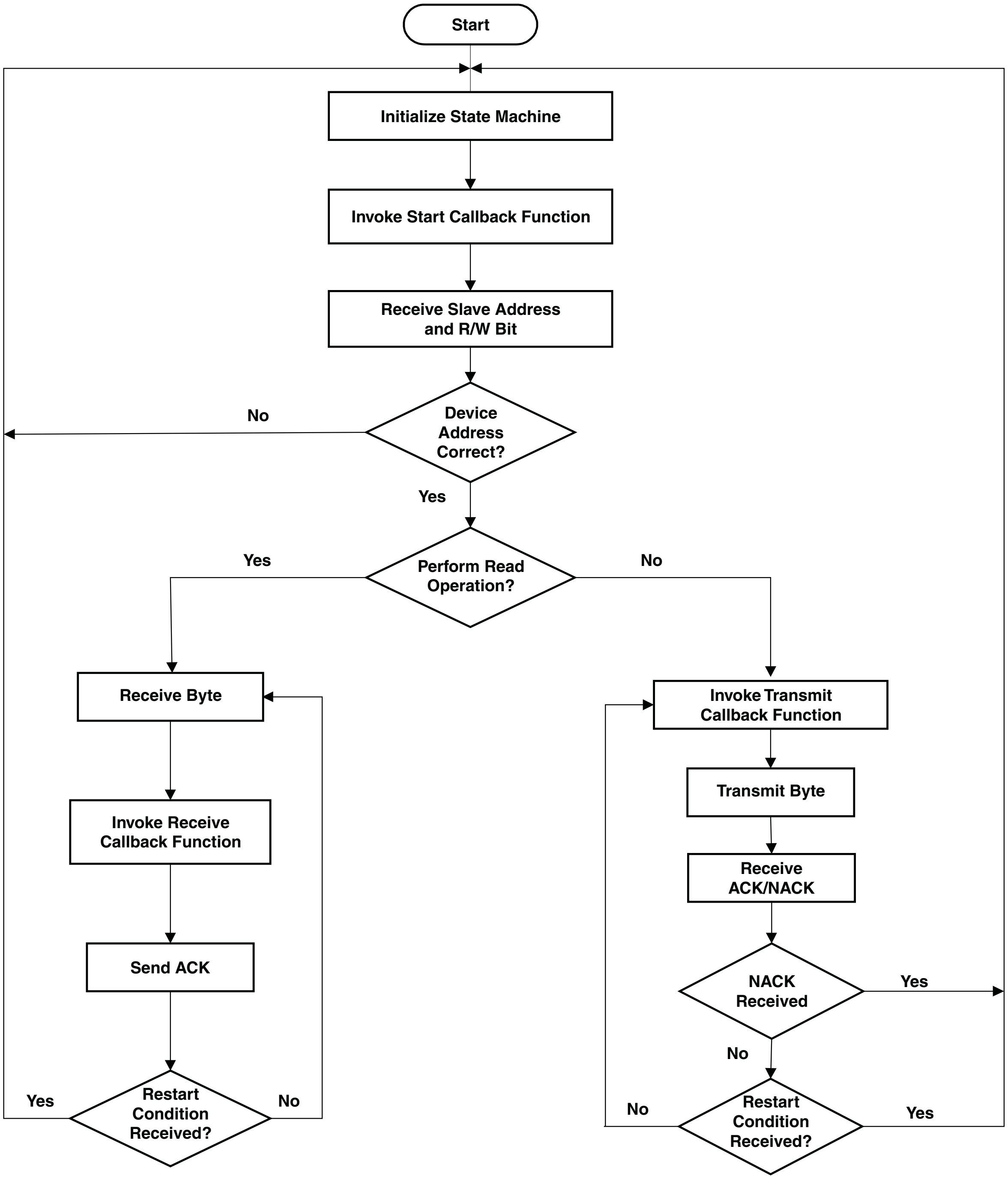
Start
Initialize State Machine
Invoke Start Callback Function
Receive Slave Address and R/W Bit
Device Address Correct?
No
Yes
Perform Read Operation?
Yes
No
Receive Byte
Invoke Receive Callback Function
Send ACK
Restart Condition Received?
Yes
No
Invoke Transmit Callback Function
Transmit Byte
Receive ACK/NACK
NACK Received
Yes
No
Restart Condition Received?
No
Yes

Figure 2. Program Flow for Slave Code Library

3.1 Usage From C

```
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog
  BCSCTL1 = CALBC1_1MHZ;                    // Set DCO
  DCOCTL = CALDCO_1MHZ;
  P1DIR |= 0x01;                            // Enable P1.0 as output
  P1OUT = 0;
  FCTL1 = FWKEY + ERASE;                    // Enable flash erase
  FCTL2 = FWKEY + FSSEL_2 + FN0;            // Flash timing setup
  FCTL3 = FWKEY;                            // Disable lock
  *(unsigned int *)0x1000 = 0;              // Dummy write to erase flash
  FCTL1 = FWKEY;
  FCTL3 = FWKEY+LOCK;                       // Diasble flash write

  /* Initialize USI module in Slave mode */
  TI_USI_I2C_SlaveInit(OwnAddress,StartCallback,RxCallback,TxCallback);
   while(1)
    {
      __disable_interrupt();
      __bis_SR_register(LPM4_bits + GIE);     // enter LPM, enable interrupts
      __no_operation();
    }
}

void StartCallback()
{
  P1OUT |= 0x01;                              // start received, turn LED on
}

int RxCallback(unsigned char RxData)
{
  // Received data byte is stored in flash
  FCTL3 = FWKEY;
  FCTL1 = FWKEY+ WRT;                       // Enable flash write
  *(unsigned char*)ptr_rx = RxData;          // Write data to flash
  ptr_rx++;                                 // Increment Rx pointer
  FCTL1 = FWKEY;                            // Disable flash write
  FCTL3 = FWKEY + LOCK;
  return TI_USI_STAY_LPM ;                  // stay in LPM
}

int TxCallback(int* TxDataPtr)
{
  // Data byte to be transmitted is passed through reference to the library
 *(unsigned char*)TxDataPtr = *(unsigned char*)ptr_tx;
  ptr_tx++;                                 // Increment tx pointer
  return TI_USI_STAY_LPM ;                  // stay in LPM
}
```

The file USI_I2CSlave.h must be included to initialize variables and functions when using the library from a C program. The TI_USI_I2C_SlaveInit(…) function needs to be called only once at the start to initialize the USI module in slave mode. The slave program (example_slave430interface.c) runs on the dedicated slave MSP430, while the master program (example_master430interface.c) runs in parallel on the dedicated master MSP430 device. The slave device uses the Info D segment (0x1000 to 0x104F) to emulate an EEPROM with 64-byte data storage. On reception of the start condition from the master, the StartCallback() function is called. This function can be used to refresh data pointers before a transmit or a receive operation.

In the previous example, the StartCallback() function turns on the LED to indicate that the I²C transaction has started. The master MSP430 transmits 16 bytes of data and then reads it back from the slave MSP430. On the reception of each data byte, the RxCallback() function is called. The received data byte is passed as an argument to this function, and the return value is used to modify the low-power mode. The TxCallback() function is called before each transmit operation. The function passes the data byte to be transmitted indirectly using a reference pointer. It also returns a zero if the existing low-power mode should be maintained or a nonzero value if the low-power mode should be exited. The data can be validated by viewing the Info D segment of flash.

Note: If C callback functions are used, the library needs to preserve additional CPU registers (C scratch registers) across the callback function call, as these registers could be affected by the callback function. If it can be ensured that the callback function does not modify the C scratch registers, they need not be preserved. Preserving the C scratch registers consumes additional code space and execution cycles. Therefore, the end user is given the option of using a predefined macro within the library to preserve the C scratch registers (USE_C_CONTEXT_SAVE is included) or leaving the registers as is (USE_C_CONTEXT_SAVE is commented out). See the compiler documentation for more details regarding C calling conventions.[2]

3.2 Usage From Assembly

```
RESET       mov.w   #SFE(CSTACK),SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
SetupDCO    mov.b   &CALBC1_1MHZ,&BCSCTL1   ; Load DCO calibration constants
            mov.b   &CALDCO_1MHZ,&DCOCTL
SetupPx     bis.b   #0x03,&P1DIR            ; P1.0 and 1.1 as output
EraseFlash  mov.w   #FWKEY+ERASE,&FCTL1     ; Set erase bit
            mov.w   #FWKEY+FSSEL_2+FN0,&FCTL2 ; Set flash timing
            mov.w   #FWKEY,&FCTL3           ; Clear lock bit
            mov.w   #0,&0x1000              ; Dummy write to start Flash erase
            mov.w   #FWKEY,&FCTL1
            mov.w   #FWKEY+LOCK,&FCTL3
            mov.w   #0x1000,&RxFlashPtr     ; Initialize Rx pointer
            mov.w   #0x1000,&TxFlashPtr     ; Initialize Tx pointer
SlaveCall
            mov.b   #0x48,R12               ; Address of slave
            mov.w   #StartCallback,R14      ; StartCallback fn. ptr
            push.w  #TxCallback             ; Tx Callback fn. ptr
            push.w  #RxCallback             ; Rx Callback fn. ptr
            call    #TI_USI_I2C_SlaveInit   ; I2C Slave initialize
            add     #4,SP                   ; compensate stack pointer
            bis.w   #LPM4+GIE,SR            ; enter LPM, enable interrupts
            jmp     $
StartCallback
            bis.b   #0x01,&P1OUT            ; start received, LED on
            ret
TxCallback
            mov.w   &TxFlashPtr,R6          ; pass data to be transmitted ...
            mov.b   @R6,0(R12)              ; ... to the library
            inc.w   &TxFlashPtr             ; increment tx pointer
            mov.w   #0,R12                  ; stay in LPM
            ret

RxCallback
            mov.w   #FWKEY+WRT,&FCTL1       ; Set WRT bit
            mov.w   #FWKEY,&FCTL3           ; Clear lock bit
            mov.w   &RxFlashPtr,R6          ; Move rxed byte to ...
            mov.b   R12,0(R6)               ; ... flash location
            mov.w   #FWKEY,&FCTL1
            mov.w   #FWKEY+LOCK,&FCTL3      ; lock flash
            inc.w   &RxFlashPtr
            mov.w   #0,R12                  ; stay in LPM
            ret
```

In assembly, the usage of the library is the same as in C. Arguments are passed using the C calling convention (making the library functions compatible for both C and assembly usage). The first two parameters are passed using registers R12 and R14; all others are pushed onto the stack.[2]

3.3 Function Description

The following functions are defined in the slave code library:

3.3.1 TI_USI_I2C_SlaveInit(...)

This is the slave initialization function. The following parameters are passed as arguments to this function:

unsigned char OwnAddress

This is the address of the slave MSP430 device.

int (* StartCallback) (void)

This function is called on the detection of the start condition.

int (* RxCallback) (unsigned char)

This function is called on the reception of a data byte. The received data is passed as an argument to the function. The function returns a zero if the existing low-power mode should be maintained or a nonzero value if the low-power mode should be exited.

int (* TxCallback) (int*)

This function is called before the transmission of a data byte. The data byte to be transmitted is stored at the location provided as an argument to this function. The function returns a zero if the existing low-power mode should be maintained or a nonzero value if the low-power mode should be exited.

3.4 Included Library Files

USI_I2CSlave.s43

This library file includes all the functionality necessary for an I^{2}C slave device to transmit and receive single and multiple bytes.

USI_I2CSlave.h

This header file has the necessary definitions and functions for the slave library and must be included in any C program that uses the library.

4 Code Size

Table 1 shows the code size for the master and slave code libraries in IAR.

Table 1. Code Size (IAR)

CODE LIBRARY	USE_C_CONTEXT_SAVE	SIZE (bytes)
Master	0	616
	1	628
Slave	0	408
	1	444

5 References

1. *MSP430x2xx Family User's Guide* (SLAU144)

2. *IAR MSP430 C/C++ Compiler Reference Guide* (ftp://ftp.iar.se/WWWfiles/msp430/guides/oc430.pdf)

3. *I^{2}C-Bus Specification and User Manual*, NXP Semiconductors, 2007 (http://www.nxp.com/acrobat/usermanuals/UM10204_3.pdf)

Revision History

Revision	Comments
SLAA368	Initial release
SLAA368A	Updated associated zip file (slaa368.zip); no changes to document

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

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