

# CHAPTER 8

## ARM® Cortex®-M4 Serial I/O Ports Programming

# ARM® CORTEX®-M4 SERIAL I/O PORTS PROGRAMMING

- This chapter provides general information about ARM® Cortex®-M4 microcontroller General Purpose Input Output (GPIO) Port programming.
- The discussion is mainly concentrated on the GPIO Ports related to **serial peripherals** used in TM4C123GH6PM MCU system.
- This discussion includes the GPIO Ports and serial peripherals specially designed for the TM4C123GH6PM MCU, general and special or alternative control functions for different GPIO Ports and pins, GPIO Ports and peripheral programming applications in TM4C123GH6PM MCU system.



## 8.1 OVERVIEW AND INTRODUCTION

- As we discussed in section 2.6.2 in Chapter 2, in the TM4C123GH6PM MCU system, the GPIO module provides interfaces for multiple peripherals or I/O devices.
- Generally GPIO provides **six** popular GPIO Ports, Ports **A ~ F**, to interface and access most system or on-chip peripherals as well as external peripherals and I/O devices to perform input/output functions.
- As we discussed in section 2.6.3.3 in Chapter 2, each GPIO Port can be mapped to an I/O block, and each block contains **8** bits or **8** pins. Each bit or each pin can be configured as either input or output pin. Also with the help of the **GPIOAFSEL**, each pin can be configured to perform multiple or different functions, which are called alternative functions.
- The advantage of using this **GPIOAFSEL** register in the GPIO module is that multiple functions can be configured and fulfilled by using a limited number of GPIO Ports and pins. The shortcoming is that this will make the GPIO Port configurations and programming more complicated and difficult.



## 8.1 OVERVIEW AND INTRODUCTION

- Most of peripherals, including system, on-chip and interfaces to the external peripherals, are connected and controlled to the GPIO Ports with related pins. In this Chapter, we will concentrate on most popular serial peripherals used in the TM4C123GH6PM MCU system:
  - **Synchronous Serial Interface (SSI)**
  - **Inter-Integrated Circuit (I<sub>2</sub>C) Interface**
  - **Universal Asynchronous Receivers/Transmitters (UARTs)**
- Some example projects related to those serial peripherals to be developed include:
  - **On-Board LCD Interface Programming Project**
  - **On-Board 7 Segment LED Interface Programming Project**
  - **Digital to Analog Converter Programming Project**
  - **I<sub>2</sub>C Interfacing Programming Project**
  - **UART Programming Project**
- The first three projects are related to SSI interfacing projects.



## 8.2 GPIO MODULE ARCHITECTURE AND GPIO PORT CONFIGURATION

- Figure 8.1 shows a functional block diagram for GPIO Ports A ~ F and all related pins used in the TM4C123GH6PM MCU system.

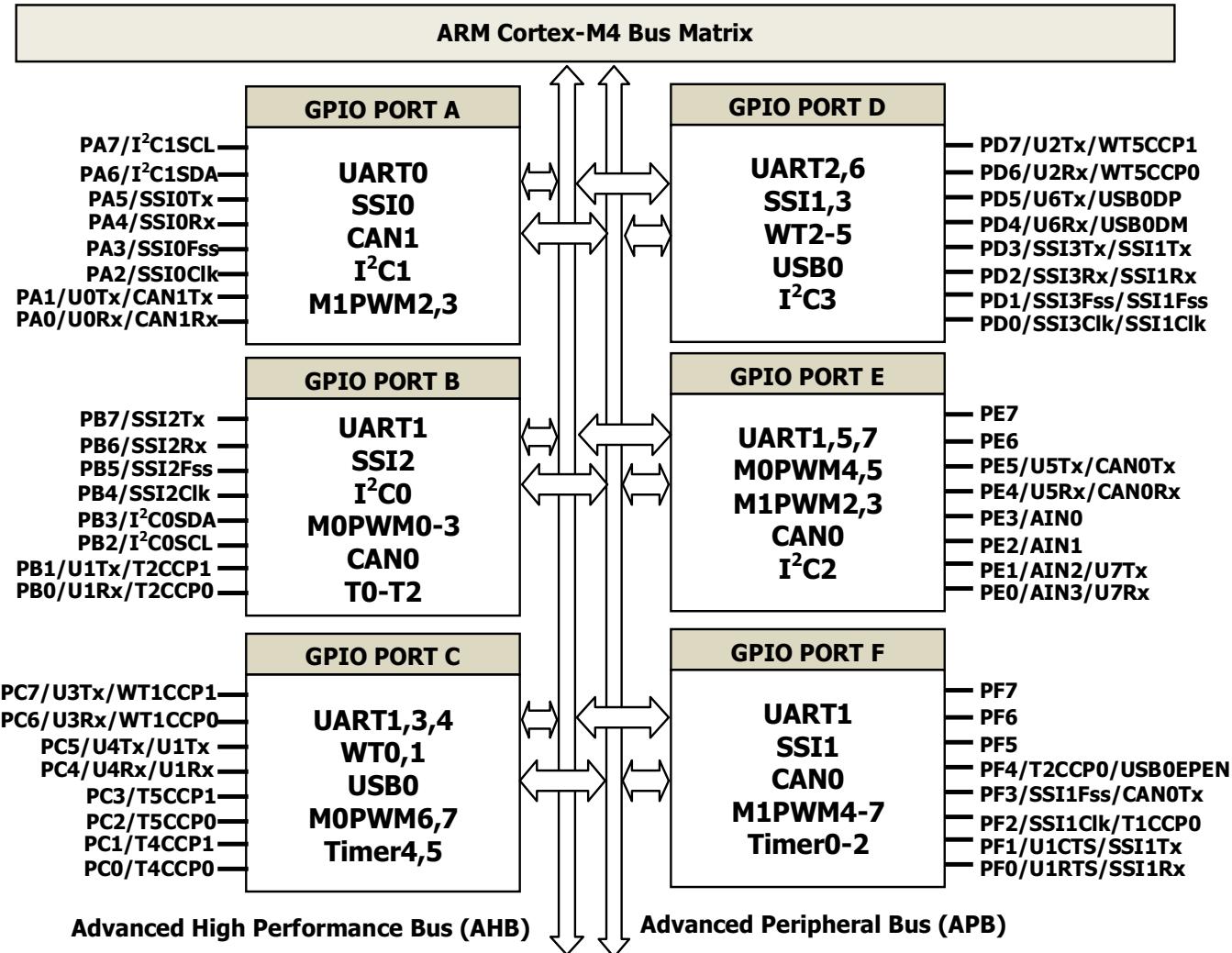


Figure 8.1 GPIO Port and pin configuration.

## 8.2 GPIO MODULE ARCHITECTURE AND GPIO PORT CONFIGURATION

- It is found from Figure 8.1 that **each pin** in each GPIO Port provides multiple or alternative functions. The actual function for each pin is determined by the combination of the **GPIOAFSEL** and the **GPIOCTRL** together.
- We provided detailed discussions about the combinations of these two registers to determine functions for each pin in section 2.6.3.3.6 in Chapter 2.
- It can also be found from Figure 8.1 that all GPIO Ports are involved in serial peripheral control and interfacing functions, such as **SSIO ~ SSI3**, **I<sub>2</sub>C<sub>0</sub> ~ I<sub>2</sub>C<sub>3</sub>**, **USBO**, **CAN<sub>0</sub> ~ CAN<sub>1</sub>** and **UART<sub>0</sub> ~ UART<sub>7</sub>**.
- All these serial control and interface functions for those serial peripherals are performed via related GPIO pins, and these GPIO pins can be easily defined and configured by setting up the **GPIOAFSEL** and **GPIOCTRL** registers.
- The lowest 8-bit in the **GPIOAFSEL** register determined whether a pin in a GPIO Port works as a GPIO pin or as an alternate function pin. If a bit is 0, the corresponding pin works as a GPIO pin. If a bit is 1, the corresponding pin works as an alternate function pin. The actual function is determined by the related 4-bit in the **GPIOCTRL** register.

## 8.2 GPIO MODULE ARCHITECTURE AND GPIO PORT CONFIGURATION

- Table 8.1 is a copy of Table 2.6 in Chapter 2 and Figure 2.19 is also redrew here as Figure 8.2 to make this issue clear.
- Refer to Figure 8.2 and Table 8.1, the alternate function for each selected pin can be uniquely determined. For example, the bits **0** and **6** in the **GPIOAFSEL** register in the GPIO Port A are set to **1**. This means that both pins **0** and **6** in the GPIO Port A should work as alternate function pins to connect to the special peripherals to provide special functions.

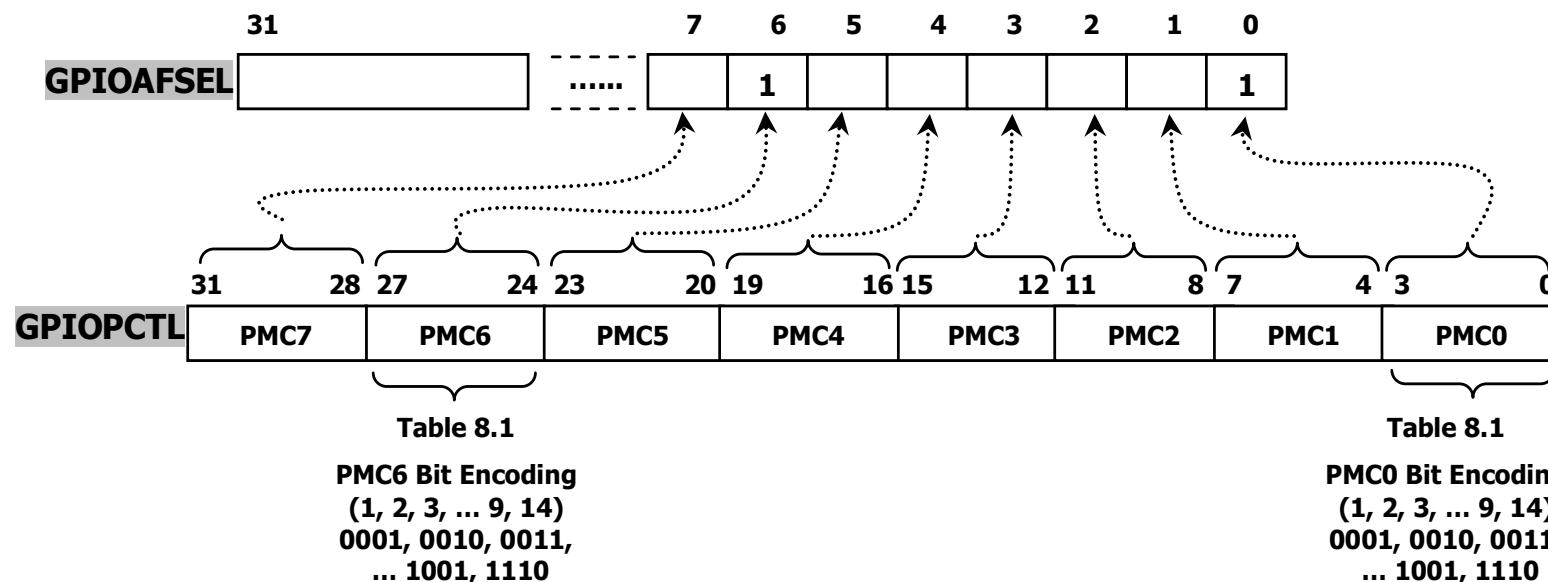


Figure 8.2 The illustration for GPIOAFSEL and GPIOPCTL registers.

## 8.2 GPIO MODULE ARCHITECTURE AND GPIO PORT CONFIGURATION

Table 8.1 GPIO Pins and Alternate Functions (Part I)

I/O	Pin	Analog Function	Digital Functions (GPIO_PCTL PMCx Bit Field Encoding)									
			1	2	3	4	5	6	7	8	9	14
PA0	17	-	U0RX	-	-	-	-	-	-	CAN1RX	-	-
PA1	18	-	U0TX	-	-	-	-	-	-	CAN1TX	-	-
PA2	19	-	-	SSI0CLK	-	-	-	-	-	-	-	-
PA3	20	-	-	SSI0FSS	-	-	-	-	-	-	-	-
PA4	21	-	-	SSI0RX	-	-	-	-	-	-	-	-
PA5	22	-	-	SSI0TX	-	-	-	-	-	-	-	-
PA6	23	-	-	-	I2C1SCL	-	M1PWM2	-	-	-	-	-
PA7	24	-	-	-	I2C1SDC	-	M1PWM3	-	-	-	-	-
PB0	45	USB0ID	U1RX	-	-	-	-	-	T2CCP0	-	-	-
PB1	46	USB0VBUS	U1TX	-	-	-	-	-	T2CCP1	-	-	-
PB2	47	-	-	-	I2C0SCL	-	-	-	T3CCP0	-	-	-
PB3	48	-	-	-	I2C0SDC	-	-	-	T3CCP1	-	-	-
PB4	58	AIN10	-	SSI2CLK	-	M0PWM2	-	-	T1CCP0	CAN0RX	-	-
PB5	57	AIN11	-	SSI2FSS	-	M0PWM3	-	-	T1CCP1	CAN0TX	-	-
PB6	1	-	-	SSI2RX	-	M0PWM0	-	-	T0CCP0	-	-	-
PB7	4	-	-	SSI2TX	-	M0PWM1	-	-	T0CCP1	-	-	-
PC0	52	-	TCK SWCLK	-	-	-	-	-	T4CCP0	-	-	-
PC1	51	-	TMS SWDIO	-	-	-	-	-	T4CCP1	-	-	-
PC2	50	-	TDI	-	-	-	-	-	T5CCP0	-	-	-
PC3	49	-	TDO SWO	-	-	-	-	-	T5CCP1	-	-	-
PC4	16	C1-	U4RX	U1RX	-	M0PWM6	-	IDX1	WT0CCP0	U1RTS	-	-
PC5	15	C1+	U4TX	U1TX	-	M0PWM7	-	PHA1	WT0CCP1	U1CTS	-	-

## 8.2 GPIO MODULE ARCHITECTURE AND GPIO PORT CONFIGURATION

Table 8.1 GPIO Pins and Alternate Functions (Part II)

I/O	Pin	Analog Function	Digital Functions (GPIOPCTL PMCx Bit Field Encoding)									
			1	2	3	4	5	6	7	8	9	14
PC6	14	C0+	U3RX	-	-	-	-	PHB1	WT1CCP0	USB0EPEN	-	-
PC7	13	C0-	U3TX	-	-	-	-	-	WT1CCP1	USB0PFLT	-	-
PD0	61	AIN7	SSI3CLK	SSI1CLK	I2C3SCL	M0PWM6	M1PWM0	-	WT2CCP0	-	-	-
PD1	62	AIN6	SSI3FSS	SSI1FSS	I2C3SDC	M0PWM7	M1PWM1	-	WT2CCP1	-	-	-
PD2	63	AIN5	SSI3RX	SSI1RX	-	M0FAULT0	-	-	WT3CCP0	USB0EPEN	-	-
PD3	64	AIN4	SSI3TX	SSI1TX	-	-	-	IDX0	WT3CCP1	USB0PFLT	-	-
PD4	43	USB0DM	U6RX	-	-	-	-	-	WT4CCP0	-	-	-
PD5	44	USB0DP	U6TX	-	-	-	-	-	WT4CCP1	-	-	-
PD6	53	-	U2RX	-	-	M0FAULT0	-	PHA0	WT5CCP0	-	-	-
PD7	10	-	U2TX	-	-	-	-	PHB0	WT5CCP1	NMI	-	-
PE0	9	AIN3	U7RX	-	-	-	-	-	-	-	-	-
PE1	8	AIN0	U7TX	-	-	-	-	-	-	-	-	-
PE2	7	AIN1	-	-	-	-	-	-	-	-	-	-
PE3	6	AIN2	-	-	-	-	-	-	-	-	-	-
PE4	59	AIN9	U5RX	-	I2C2SCL	M0PWM4	M1PWM2	-	-	CAN0RX	-	-
PE5	60	AIN8	U5TX	-	I2C2SDC	M0PWM5	M1PWM3	-	-	CAN0TX	-	-
PF0	28	-	U1RTS	SSI1RX	CAN0RX	-	M1PWM4	PHA0	T0CCP0	NMI	C0O	-
PF1	29	-	U1CTS	SSI1TX	-	-	M1PWM5	PHB0	T0CCP1	-	C1O	TRD1
PF2	30	-	-	SSI1CLK	-	M0FAULT0	M1PWM6	-	T1CCP0	-	-	TRD0
PF3	31	-	-	SSI1FSS	CAN0TX	-	M1PWM7	-	T1CCP1	-	-	TRCLK
PF4	5	-	-	-	-	-	M1FAULT0	IDX0	T2CCP0	USB0EPEN	-	-

## 8.2 GPIO MODULE ARCHITECTURE AND GPIO PORT CONFIGURATION

- For bit **0** in the **GPIOAFSEL** register, the alternate function can be determined by the **PMC0** (lowest 4-bit) in the **GPIOPCTL** register, which is shown in Table 8.1.
- If **PMC0** in the **GPIOPCTL** register is **1** (0001), this means that the **PA0** should work as a receiving pin for the **UART0** (Table 8.1). However, for bit **6** in the **GPIOAFSEL** register, if the **PMC6** (bits 24 ~ 27) in the **GPIOPCTL** register is **5** (0101), the **PA6** pin should work as **M1PWM2** pin to connect to that PWM peripheral to provide a pulse width modulation output.
- As we discussed in section 2.6.3.4 in Chapter 2, each GPIO Port (Ports A ~ F) contains a set of GPIO Registers and each register can be accessed by using an offset address combining (plus) with a base address.
- In the following sections, we will discuss most popular serial peripherals used in the TM4C123GH6PM MCU system. Because of the page limitation, we move the CAN and QEI peripherals and their interfaces to Chapter 10.



## 8.3 SYNCHRONOUS SERIAL INTERFACE (SSI)

- Two terminologies are widely implemented in the serial communication systems: 1) ***Serial Communication Interface (SCI)***, and 2) ***Serial Peripheral Interface (SPI)***.
- The **SCI** belongs to the asynchronous serial communication interface but **SPI** belongs to the synchronous serial communication interface. In the TM4C123GH6PM MCU system, the **SPI** is used to work as a synchronous serial interface.
- Like any other serial communication interfaces, both **SCI** and **SPI** can use a full-duplex or half-duplex mode to perform the data transfers between terminals and peripherals.
  - The **full-duplex** communication mode uses two data lines (input and output data lines) with one line carrying serial data from peripheral to microcontroller (input) and the other line carrying serial data from microcontroller to peripheral (output).
  - The **half-duplex** communication mode uses only a single data line to transfer data in both directions (input and output) at the different period of the time. Obviously, the single data line cannot transfer data in both directions simultaneously. Therefore the full-duplex mode provides faster data transfer rate because it can transfer data in both directions at the same time with two separate data lines.

## 8.3 SYNCHRONOUS SERIAL INTERFACE (SSI)

- The TM4C123GH6PM microcontroller system includes **four** Synchronous Serial Interface modules (**SSI<sub>0</sub> ~ SSI<sub>3</sub>**).
- Each **SSI** module provides a master or a slave interface for synchronous serial communication with peripheral devices that have either **Freescale SPI**, **MICROWIRE**, or **Texas Instruments™** synchronous serial interfaces.
- The TM4C123GH6PM SSI modules provide the following features:
  - Programmable interface operation for **Freescale SPI**, **MICROWIRE**, or **Texas Instruments™** synchronous serial interfaces
  - Master or slave operation
  - Programmable clock bit rate and prescaler
  - Separate transmit and receive FIFOs, and each FIFO contains  $8 \times 16$ -bit data
  - Programmable data frame size from 4 to 16 bits
  - Internal loopback test mode for diagnostic/debug testing
  - Standard FIFO-based interrupts and End-of-Transmission interrupt
  - Efficient transfers using Micro Direct Memory Access Controller (**μDMA**)



### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- Some popular **asynchronous** serial communication protocols are:
  - **Acknowledge/Not Acknowledge (ACK/NAK) Flow Control:** When using this protocol to transfer data between a transmitter and a receiver, each time after a block of data is sent out by the transmitter, the transmitter will wait for an acknowledge signal from the receiver before it can send out the next block of data. After a block of data is received by the receiver, it checks and confirms whether the received data is valid. If the data is valid, the receiver sends back an (ACK) acknowledgement signal to the transmitter to inform the latter that this data transfer is successful and the transmitter can continue to send the next block of data. However, if the data is invalid, the receiver sends back a negative (NAK) acknowledgement signal to the transmitter to inform the latter that the last block of data transfer is failed, and the transmitter will resend the previous block of data.
  - **XON/XOFF Flow Control:** When using this protocol to transfer data, the receiver sends an XON signal to the transmitter to inform the transmitter that the receiver is ready to receive the data from the transmitter. After receiving this XON signal, the transmitter begins to send out a sequence of block data. If for some reason the receiver cannot continue to handle and process the received data, such as the receiver's buffer being full or the receiver being busy, the receiver sends an XOFF signal back to the transmitter to tell the latter to stop data transmission. XON and XOFF are the **ASCII** control characters DC1 (CTRL-Q) and DC3 (CTRL-S).

### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- The so-called **data framing** is the format of the data to be transmitted or received by master or slaver. This format is very important to the asynchronous interface.
- For asynchronous data communications, the *asynchronous* means that the data transmitter and the receiver cannot share a common timing base, which means that the transmitter and the receiver must use different clocks as their data transfer timing bases.
- In order to coordinate this asynchronous serial communications, the data transmitter must send special code format or code bits before and after the serial data bits to enable the receiver to check and confirm the correctness of that data transfer based on those code bits.
- These code bits can be considered as the *framing bits* or *frame* to inform the receiver when the data transfer begins and when it ends. Figure 8.3 shows some popular data framing bits structures (next slide).



### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

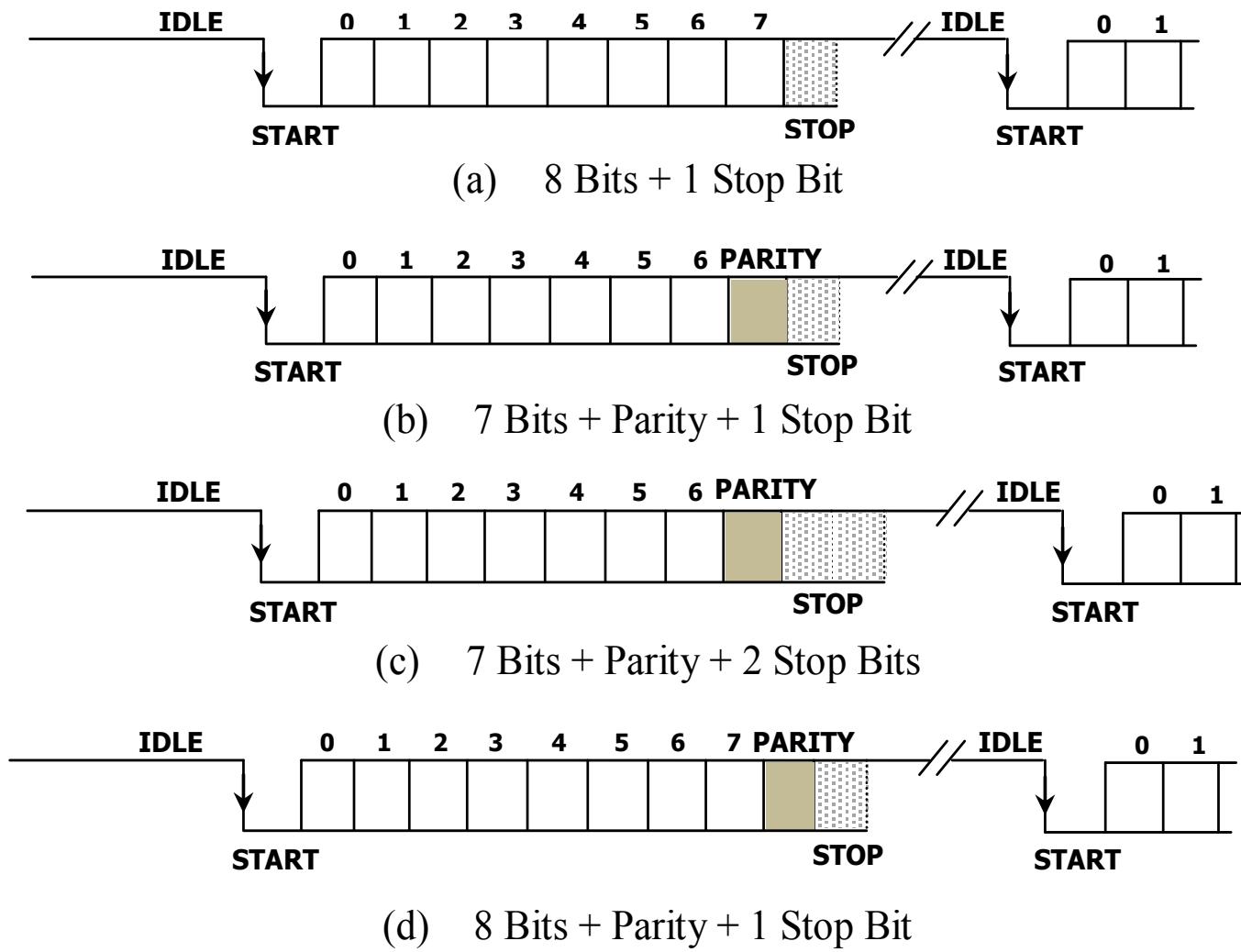


Figure 8.3 Some popular data transfer framing bits structures.

### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- The **synchronous** means that both transmitter and receiver use the same clock rate **SClk** as their data operation timing base. We will discuss the protocols and data framing for synchronous data in section 8.3.3.
- The basic data unit (group of bits) of information is the character or data frame, and normally it is a byte (**8-bit**) with bit by bit in a timing sequence. The transmitter can send a byte at any rate and the receiver needs to know when the byte starts and when it stops.
- As shown in Figure 8.3, depends on the different framing bits, the data can be transferred as a 8-bit data (1 byte) plus a stop bit (Figure 8.3a), a 7-bit data plus a parity bit with a stop bit (Figure 8.3b), 7-bit data plus parity with 2 stop bits (Figure 8.3c) or 8-bit data plus parity with a stop bit (Figure 8.3d).
- Note that a constant **LOW** or **HIGH** signal is always considered as an **idle** status with no data being transferred. A **LOW-TO-HIGH** or a **HIGH-TO-LOW** edge is always considered as the **starting signal** for a block of data to be transferred.



### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- Also note that the **data bits** are transmitted with the **MSB** first, and there is no any level transition between bits of the same value.
- For example, if all data bits were **1**, the transmitted frame would be **LOW** only during the start bit. This format is called non return to zero (**NRZ**), which means that the voltage does not return to zero between adjacent **1** bits.
- The **stop** bits can be either one or two. Some other systems may use **1.5** stop bits. Typically, the transmitter hardware automatically creates the framing bits and the receiver hardware automatically removes them.
- The **parity** bit is used to check and confirm the correctness of the last data to be transferred. It refers to the quantity of **1** bits in a binary number. If that quantity is **even**, the number has **even parity**. However, if it is **odd**, the number has **odd parity**. Note that the standard ASCII code is 7-bit, therefore the eighth bit may be used as a parity bit.



### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- The transmitter is responsible to create the **parity** bit and the receiver deciphers it. The parity bit's value can be either **0** or **1**, depending on the following two conditions:
  - The type of parity selected, **even** or **odd**.
  - The quantity of **1** bits in the data byte to be transferred.
- For example, the transmitter and receiver agree to use 7-bit plus **odd parity** with one **stop bit**, and an ASCII code 8 (**111000**) will be transferred between them. Because the **1** bits in this code is **3**, which is an odd number, the parity bit will be set to **1** by the transmitter to indicate that this transferring data contains an odd number of **1**'s.
- When the receiver gets this data, it will first count the number of **1**'s in the received data and check the value on the received parity bit. If both agree, it means that this data transfer is successful. Otherwise, if the number of **1**'s in the received data is an even number, and the parity is **1**, it means that this data transfer is wrong. The receiver will send a **NAK** signal back to the transmitter to ask it to resend this data.

### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- The following items are popular used in serial data communications:
  - **Mark and Space:** In addition to the data transfer framing bits, the serial data transfer voltage levels would also be noted. The **SSI** drives the **T<sub>X</sub>** line to **4.2V** for logic **1**, which is called a **Mark**, and to **0.4V** for logic **0**, which is called a **Space**. The **SSI** recognizes **3.5 ~ 5V** on its **R<sub>X</sub>** line as a **mark** and **0 ~ 1V** as a **space**.
  - **Data Speed and Baud Rate:** In serial data communications, the data speed is defined as the number of bits transmitted per second (**BPS**) or called **Baud rate**. Baud rate includes all bits involved in a serial data communication, including the **start, parity and stop** bits. A **9600** baud rate means that the serial data is transmitted and received at the rate of 9600 bits per second.
  - The **Break Signal:** Sometimes it may necessary to stop or break a normal serial data communication for some unforeseen reasons. The break signal provides this function to get a device's attention to abort a data transfer in midstream. The break signal is a signal that provides a logic **LOW** on the data transmission line for a longer period of time than a frame time. This long logic **LOW** is different with any normal data frame bits. As we know, each data unit to be transferred must include at least one stop bit (**HIGH**) and the idle line is a constant **HIGH** if no any data to be transferred. A long logic **LOW** indicates that a break signal is being transferred to inform the device to stop any further data transferring.

### 8.3.1 ASYNCHRONOUS & SYNCHRONOUS COMMUNICATION PROTOCOLS & DATA FRAMING

- The following items are popular used in serial data communications:
  - **Modems:** A modem can be considered as an I/O port and it is mainly used to coordinate and control the serial data communication with the following functions:
    - Perform **data format conversions** between the microcontroller and serial data communication lines. Two typical conversions are involved: 1) **Parallel-To-Serial data conversion** (used to convert the parallel data sent by microcontroller to the serial data that can be accepted by the serial transmitting line), and 2) **Serial-To-Parallel data conversion** (used to convert the serial data received from the serial transmitting line to the parallel data that can be accepted by the microcontroller).
    - Perform the **voltage level conversions** between the microcontroller and serial data communication lines. As we know, the typical serial data communication convention is RS-232. In this convention, the voltage level for logic **HIGH** is **12V** and logic **LOW** is **-12V**. However, inside the microcontroller, it uses TTL logic with **5V** as a logic **HIGH** and **0.7** as a logic **LOW**. Therefore a voltage level conversion is necessary for these different logic levels.



### 8.3.2 SYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

- The **Synchronous Serial Interface (SSI)** module provides the functionality for synchronous serial communications with peripheral devices, and can be configured to use different transmission protocols, such as Freescale SPI, MICROWIRE, or the Texas Instruments™ synchronous serial frame formats. The size of the data frame can be configured to be between **4** and **16** bits.
- The **SSI** module performs **serial-to-parallel** data conversion on data received side, and **parallel-to-serial** conversion on data transmitted side. The **T<sub>X</sub>** and **R<sub>X</sub>** paths are buffered with internal FIFOs allowing up to eight 16-bit values to be stored independently.
- The **SSI** module can be configured as either a master or a slave device. As a slave device, the SSI module can also be configured to disable its output, which allows a master device to be coupled with multiple slave devices.
- The **SSI** module also includes a programmable bit rate clock divider and prescaler to generate the output serial clock derived from the SSI module's input clock. Some devices can use the PIOSC as the serial bit clock. Bit rates are generated based on the input clock and the maximum bit rate supported by the connected peripheral.

### 8.3.2 SYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

- For parts that include a DMA controller, the **SSI** module also provides a DMA interface to facilitate data transfer via DMA.
- Figure 8.4 shows the architecture and functional block diagram for a single **SSI** module used in the TM4C123 system. For all four SSI modules, each module contains an identical and independent set of these registers to control and configure each SSI module's data transmitting and receiving functions.
- Each **SSI** module can be configured and controlled by the following groups of registers:

#### 1. Control/Status Register Group:

- SSI Control 0 Register (SSICR0)**: Setup protocol mode, clock rate and data size.
- SSI Control 1 Register (SSICR1)**: Setup master/slave mode.
- SSI Status Register (SSISR)**: Provide SSI working status (busy, FIFO empty or full).
- SSI Data Register (SSIDR)**: 16-bit register used to store data to be written or read.
- Transmit FIFO ( $T_x$ FIFO)**: Store  $8 \times 16$ -bit data to be transmitted.
- Receive FIFO ( $R_x$ FIFO)**: Store  $8 \times 16$ -bit data to be received.



## 8.3.2 SYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

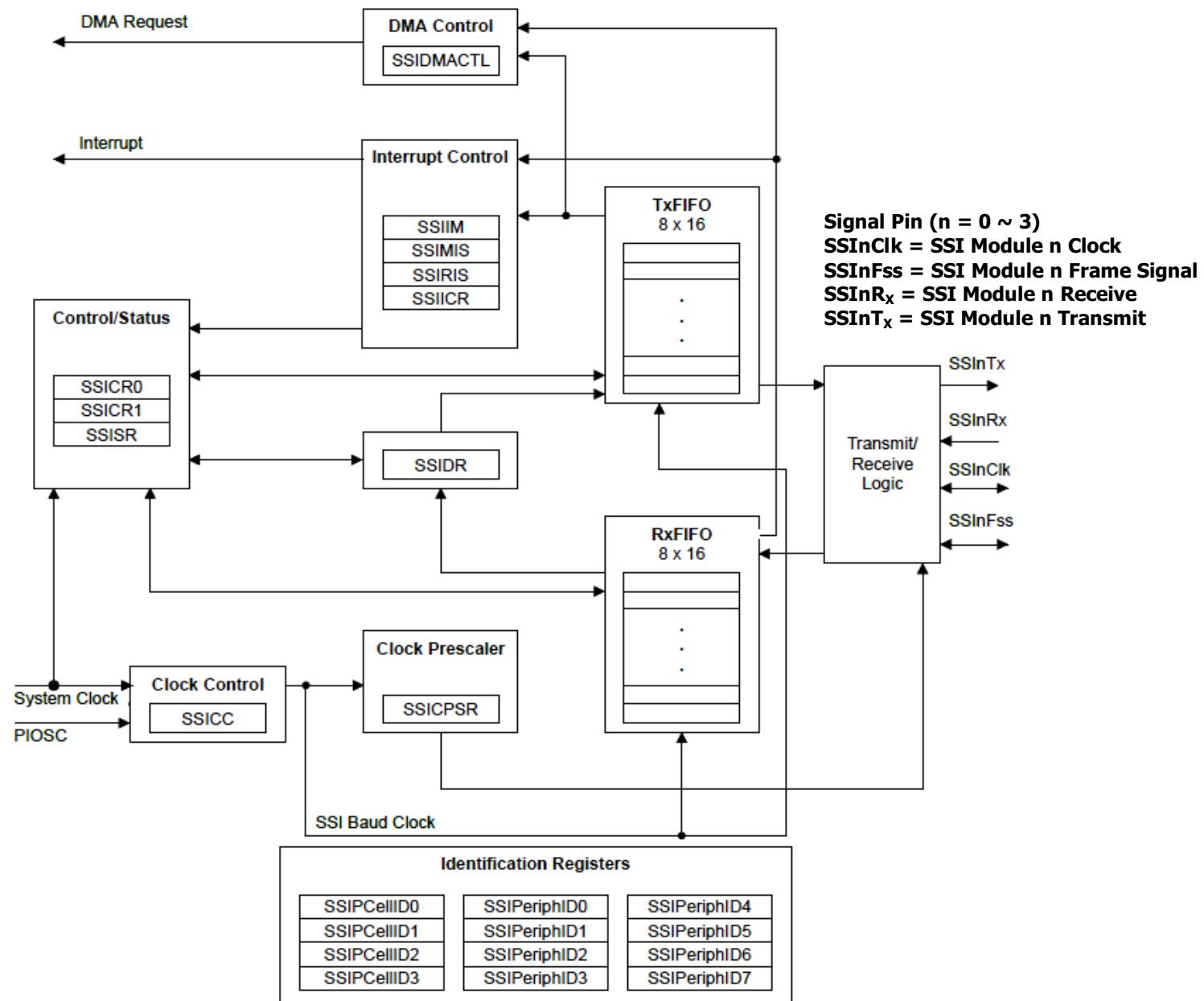


Figure 8.4 Functional block diagram of the SSI module.

## 8.3.2 SYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

### 2. Clock Control Group:

- SSI Clock Configure Register (**SSICC**): Control & select the clock source for the SSI.
- SSI Clock Prescale Register (**SSICPSR**): Determine the divisor for the system clock.

### 3. Interrupt and DMA Control Group:

- SSI Interrupt Mask Register (**SSIIM**): Enable (unmask)/disable (mask) SSI interrupt.
- SSI Raw Interrupt Status Register (**SSIRIS**): Provide SSI raw interrupt status.
- SSI Interrupt Clear Register (**SSIICR**): Clear SSI interrupts.
- SSI Masked Interrupt Status Register (**SSIMIS**): Provide masked SSI interrupt status.
- SSI DMA Control Register (**SSIDMACTL**): Provide DMA control.

### 4. Transmit/Receive Logic: Provide 4 data communication control/handshaking signals:

- **SSInClk**: SSI Module n Clock. (n = 0 ~ 3)
- **SSInFss**: SSI Module n Frame.
- **SSInRx**: SSI Module n Receive.
- **SSInTx**: SSI Module n Transmit.



## 8.3.2 SYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

### 5. Identification Group:

- Twelve registers in this group are used to provide identification numbers for all used SSI peripherals.
- We will discuss these registers and their functions one by one in the following sections.
- First let's have a closer look at the data transmission format or frame used in the serial data communications.



### 8.3.3 THE SYNCHRONOUS DATA TRANSMISSION FORMAT AND FRAME

- General transmission and receiving process for synchronous serial data are:
  - 1) Under the control and configuration of related **SSI control registers**, each piece of data to be transmitted is first stored into the **FIFO** via **SSIDR**.
  - 2) When the transmission and reception process begins, each transmitted data is sent from the **FIFO** to the serial communication line via **SSInTx** pin.
  - 3) The transmission/reception speed is controlled by the **baud rate** determined by the **SSICC** and **SSICPSR** registers.
  - 4) The data transmission/reception frame is controlled by the **SSInFss** and the **SSInClk** signals together.
- Each data frame is between **4** and **16** bits long depending on the size of data programmed and is transmitted starting with the **MSB**.
- Three basic frame types that can be selected by programming the **FRF** bit in the **SSICRo** register are:
  - 1) **Texas Instruments™ Synchronous Serial**
  - 2) **Freescale SPI**
  - 3) **MICROWIRE**



### 8.3.3 THE SYNCHRONOUS DATA TRANSMISSION FORMAT AND FRAME

- For all three formats, the serial clock **SSInClk** is held inactive while the **SSI** is idle, and it becomes active and transitions at the programmed frequency only during the active transmission or reception of data.
- The idle state of **SSInClk** is used to provide a receive timeout indication that occurs when the receiving **FIFO** still contains data after a timeout period.
- In summary, when transmitting/receiving data with three different data frames, the **SSInFss** pin is forced to **LOW** to provide an active signal to enable the data to be transmitted/received between master and slave devices. The **SSInFss** is also used to select the slave device. The differences between these three frames are:
  - For the **TI Synchronous Serial** frame, the rising edge of the **SSInClk** is always an active edge used to trigger the transmitter or receiver to latch the data bit into the **SSInTx** or **SSInRx** line.
  - For the **Freescale SPI** frame, the active level or the polarity of the **SSInClk** and the active triggering edge or the phase of the **SSInClk** can be programmed by using two bits, **SPO** (Serial Clock Polarity) and **SPH** (Serial Clock Phase) controls, in the **SSICRo** register.
  - For the **MACROWIRE** frame, it is similar to **TI Synchronous Serial** format except that its transmission is **half-duplex** instead of full-duplex and uses a master-slave message passing technique.

### 8.3.3 THE SYNCHRONOUS DATA TRANSMISSION FORMAT AND FRAME

- Table 8.2 lists some important properties for three data frames used in the synchronous serial interface.

Table 8.2 Three data frames used in synchronous serial interface SSI.

Frame	Mode	SSInClk Active level	SSInFss Active level	Function
<b>TI Synchronous Serial</b>	Full-Duplex 4 wires	LOW	One pulse at one clock period	The <b>SSInFss</b> is pulsed High for one <b>SSInClk</b> period. The value to be transmitted is transferred from the transmit FIFO to the serial shift register of the transmit logic. On the next rising edge of <b>SSInClk</b> , the MSB of the 4 to 16-bit data frame is shifted out on the <b>SSInTx</b> pin. The MSB of the received data is shifted onto the <b>SSInRx</b> pin by the off-chip serial slave device.
<b>Freescale SPI</b>	Full-Duplex 4 wires	Programmed by SPO & SPH	LOW	The Freescale SPI interface is a four-wire interface where the <b>SSInFss</b> signal behaves as a slave select. The main feature of the Freescale SPI format is that the inactive state and phase of the <b>SSInClk</b> signal are programmable through the <b>SPO</b> and <b>SPH</b> bits in the <b>SSICR0</b> control register.
<b>MICROWIRE</b>	Half-Duplex 4 wires	Rising Edge	LOW	<b>MICROWIRE</b> format is very similar to SPI format, except that transmission is half-duplex instead of full-duplex and uses a master-slave message passing technique. Each serial transmission begins with an 8-bit control word that is transmitted from the SSI to the off-chip slave device. During this transmission, no incoming data is received by the SSI. After the message has been sent, the off-chip slave decodes it and, after waiting one serial clock after the last bit of the 8-bit control message has been sent, responds with the required data. The returned data is 4 to 16 bits in length, making the total frame length anywhere from 13 to 25 bits.

### 8.3.3.1 TEXAS INSTRUMENTS™ SYNCHRONOUS SERIAL FRAME

- Figure 8.5 shows an illustration for the data transmission/reception using the **TI Synchronous Serial frame**. The operational sequence for data transmitting/receiving in this frame is:
  - When the **SSI** is idle, the transmission line **SSInTx** is in tri-state and the **SSInClk** and the **SSInFss** signals are forced to **LOW**.

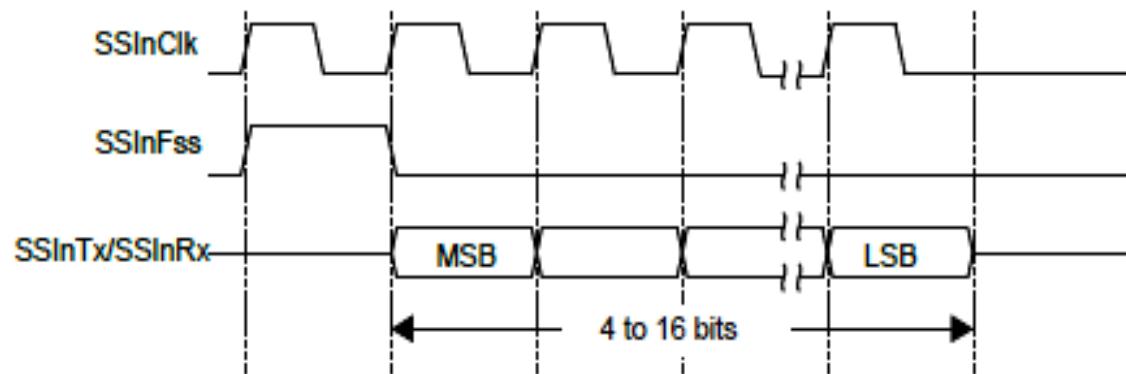


Figure 8.5 Operational timing sequence of the TI synchronous serial frame.

### 8.3.3.1 TEXAS INSTRUMENTS™ SYNCHRONOUS SERIAL FRAME

2. Once the transmit **FIFO** contains any data, the **SSInFss** is pulsed **High** for one **SSInClk** period to indicate that a transmission begins. Then the **SSInFss** is kept in **LOW** for the entire data transmission/reception period. The value to be transmitted is also transferred from the transmit FIFO to the serial shift register of the transmit logic.
3. On the next rising edge of **SSInClk**, the **MSB** of the 4 to 16-bit data frame is shifted out on the **SSInTx** pin. Likewise, the **MSB** of the received data is shifted onto the **SSInRx** pin by the off-chip serial slave device at the same time.
4. Both the **SSI** and the off-chip serial slave device then clock each data bit into their serial shifter on each **falling** edge of **SSInClk**.
5. After the **LSB** has been latched, the entire received data (4 ~ 16 bits) that are located at the serial shifter is then transferred from the serial shifter to the receive **FIFO** on the next rising edge of **SSInClk**.

### 8.3.3.2 FREESCALE SPI FRAME

- Similar to the **TI Synchronous Serial** frame, the **Freescale SPI** interface is also a four-wire interface where the **SSInFss** signal works as a slave select.
- The special feature of the **Freescale SPI format** is that the inactive state and phase of the **SSInClk** signal can be programmable through the **SPO** and **SPH** bits in the **SSICRo** control register.
  - The **SPO** bit determines the inactive level of the **SSInClk**. When the **SPO** clock polarity control bit is **0**, it creates **Low** value on the **SSInClk** pin. If the **SPO** bit is set, a steady state **High** value is placed on the **SSInClk** pin when data is not being transferred.
  - The **SPH** phase control bit selects the clock active edge that captures data and allows it to change state. The state of this bit has the most impact on the first bit transmitted by either allowing or not allowing a clock transition before the first data capture edge. When the **SPH** phase control bit is **0**, data is captured on the first clock edge transition. If the **SPH** bit is **1**, data is captured on the second clock edge transition.
- Figure 8.6 (next slide) shows the timing sequence when **SPO = 0** and **SPH = 0**, which means that the inactive level of the **SSInClk** is **Low** and the data is captured by the first clock edge transition.

### 8.3.3.2 FREESCALE SPI FRAME

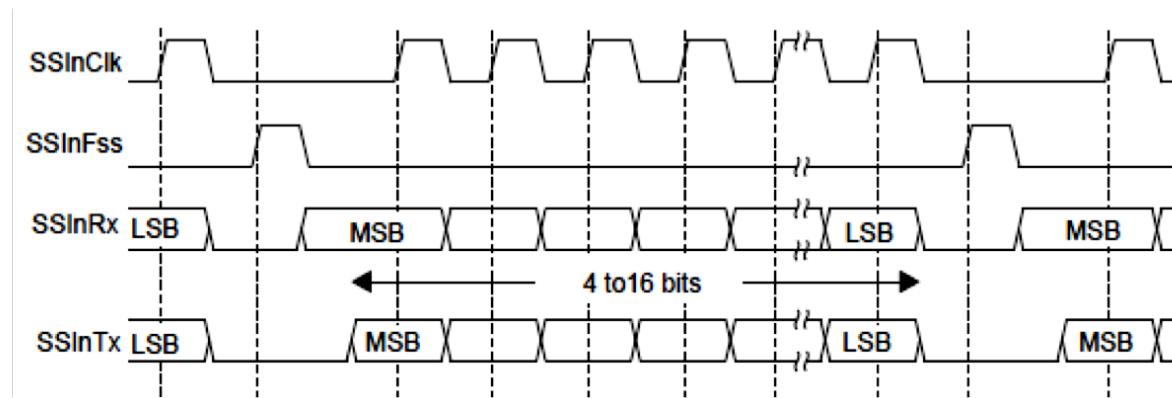


Figure 8.6 The operational sequence for Freescale SPI frame (**SPO=SPH=0**).

- Figure 8.7 shows the operational sequence when **SPO = 0** and **SPH = 1**. This setting means that the inactive level of the **SSInClk** is **Low** and the data is captured at the second clock edge.

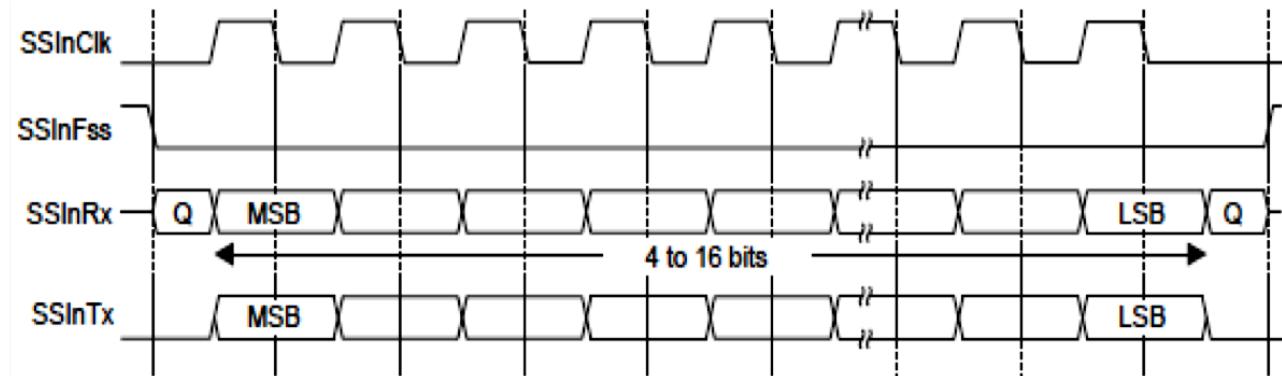


Figure 8.7 The operational sequence for Freescale SPI frame (**SPO=0, SPH=1**).

### 8.3.3.3 MICROWIRE FRAME

- Figure 8.8 shows an illustration of the operational sequence for MACROWIRE frame.

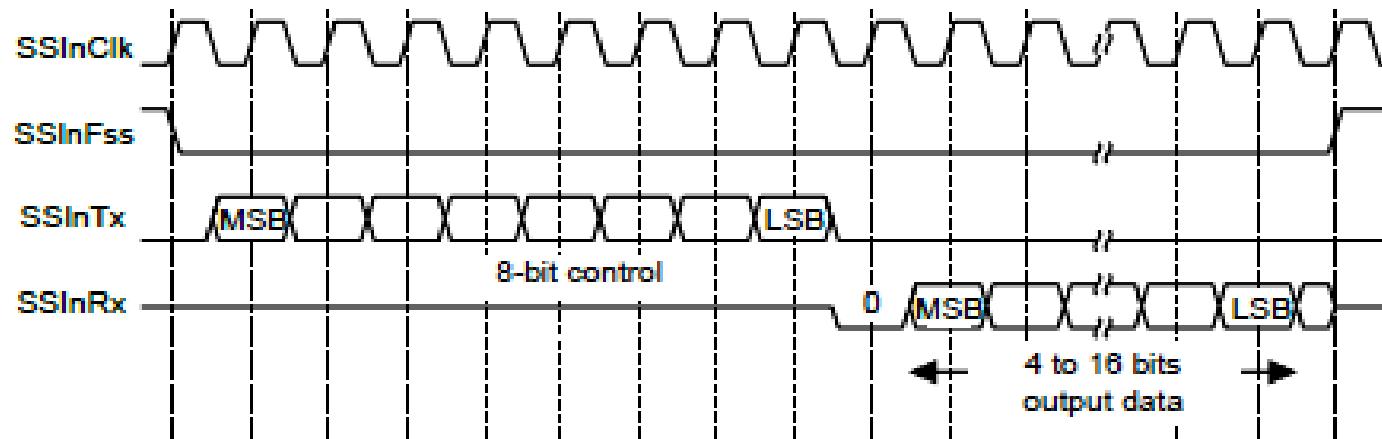


Figure 8.8 The operational sequence for MACROWIRE frame.

- Each serial transmission begins with an **8-bit control word** that is transmitted from the **SSI** to the off-chip slave device. After the message has been sent, the off-chip slave decodes it and waits one serial clock after the last bit of the 8-bit control message has been sent. Then the slave device responds with the required data. The returned data is **4 to 16** bits in length, making the total frame length anywhere from 13 to 25 bits.

### 8.3.3.3 MICROWIRE FRAME

- The data transmission begins after the control byte is sent. The falling edge of **SSInFss** causes the value contained in the transmit FIFO to be transferred to the serial shift register of the transmit logic and the **MSB** of the 8-bit control frame to be shifted out onto the **SSInTx** pin. **SSInFss** remains **Low** for the duration of the frame transmission. The **SSInRx** pin remains tri-stated during this transmission.
- The off-chip serial slave device latches each control bit into its serial shifter on each rising edge of **SSInClk**. After the last bit is latched by the slave device, the control byte is decoded during a one clock wait-state, and the slave responds by transmitting data back to the **SSI**.
- Each bit is driven onto the **SSInRx** line on the falling edge of **SSInClk**. The **SSI** in turn latches each bit on the rising edge of **SSInClk**. At the end of the frame, for single transfers, the **SSInFss** signal is pulled **High** one clock period after the last bit has been latched in the receive serial shifter, causing the data to be transferred to the receive FIFO.

### 8.3.3.3 MICROWIRE FRAME

- For continuous transfers, data transmission begins and ends in the same manner as a single transfer. However, the SSInFss line is continuously asserted Low and transmission of data occurs back-to-back.
- In most applications, the microcontroller is selected as a **master** device and some peripherals are selected to work as slave devices.
- In that case, the transmission line **Master Output/Slave Input (MOSI)** is connected to the **SSInTx** pin on the master device side, and the **Master Input/Slave Output (MISO)** is connected to the **SSInRx** pin in the slave device side.

### 8.3.4 SSI MODULE COMPONENTS AND SIGNAL DESCRIPTIONS

- As shown in Figure 8.4, all **SSI** module components and their control functions are globally illustrated by this block diagram.
- However, in order to get more details about these components and control signals, we need to discuss them one by one with more details based on each group of registers and their functions.

### 8.3.4.1 SSI CONTROL SIGNALS AND GPIO SSI CONTROL REGISTERS

- Most peripheral devices in TM4C123 system are related to GPIO Ports.
- All control signals and inputs/outputs of peripherals are transferred by using related GPIO pins. This is also true to **SSI** modules. All SSI control signals are connected to related GPIO Ports/pins, as shown in Table 8.3.

Table 8.3 SSI control signals and GPIO pins distributions

SSI Pin	GPIO Pin	Pin Type	Pin Function
<b>SSI0Clk</b>	<b>PA2 (2)</b>	I/O	SSI Module 0 Clock
<b>SSI0Fss</b>	<b>PA3 (2)</b>	I/O	SSI Module 0 Frame
<b>SSI0Rx</b>	<b>PA4 (2)</b>	I	SSI Module 0 Receive
<b>SSI0Tx</b>	<b>PA5 (2)</b>	O	SSI Module 0 Transmit
<b>SSI1Clk</b>	<b>PF2 (2)</b> <b>PD0 (2)</b>	I/O	SSI Module 1 Clock
<b>SSI1Fss</b>	<b>PF3 (2)</b> <b>PD1 (2)</b>	I/O	SSI Module 1 Frame
<b>SSI1Rx</b>	<b>PF0 (2)</b> <b>PD2 (2)</b>	I	SSI Module 1 Receive
<b>SSI1Tx</b>	<b>PF1 (2)</b> <b>PD3 (2)</b>	O	SSI Module 1 Transmit
<b>SSI2Clk</b>	<b>PB4 (2)</b>	I/O	SSI Module 2 Clock
<b>SSI2Fss</b>	<b>PB5 (2)</b>	I/O	SSI Module 2 Frame
<b>SSI2Rx</b>	<b>PB6 (2)</b>	I	SSI Module 2 Receive
<b>SSI2Tx</b>	<b>PB7 (2)</b>	O	SSI Module 2 Transmit
<b>SSI3Clk</b>	<b>PD0 (1)</b>	I/O	SSI Module 3 Clock
<b>SSI3Fss</b>	<b>PD1 (1)</b>	I/O	SSI Module 3 Frame
<b>SSI3Rx</b>	<b>PD2 (1)</b>	I	SSI Module 3 Receive
<b>SSI3Tx</b>	<b>PD3 (1)</b>	O	SSI Module 3 Transmit

### 8.3.4.1 SSI CONTROL SIGNALS AND GPIO SSI CONTROL REGISTERS

- To enable the corresponding GPIO pins to work as **SSI** module pins to transfer related SSI control signals, the **AFSEL** bits in the **GPIOAFSEL** register should be set to **1** to choose the **SSI** function.
- The number in parentheses is the encoding that must be programmed into the **PMCx** field in the **GPIOPCTL** register to assign the SSI signal to the specified GPIO pin. The following configurations should be performed:
  - Enable the clock to the appropriate GPIO modules via the **RCGCGPIO** register.
  - The related **AFSEL** bits (lower 8-bit) in the **GPIOAFSEL** register must be set to **1**.
  - The related bits in the **GPIODEN** register must be set to **1** to enable the digital function and disable the analog function.
  - The related **PMCx** field in the **GPIOPCTL** register must be assigned with appropriate codes to enable the GPIO pins to work as desired **SSI** signal pins.
  - Optionally the related **AMSEL** bits in the **GPIOAMSEL** register should be clear to **0** to activate the analog isolation circuits to enable the selected pin to work as digital function pin.



### 8.3.4.2 SSI MODULE BIT RATE GENERATION AND CLOCK CONTROL

- Two **clock sources** can be used as the input clock to the **SSI** module (**SSInClk**); the system clock or Precision Internal Oscillator (**PIOSC**).
- The SSI Clock Configuration (**SSICC**) register is used to select one of these two clock sources. The following conditions must be met to ensure that the system clock can provide suitable source to the **SSInClk**:
  - If the **PIOSC** is used for the **SSI** baud clock, the system clock frequency must be at least **16 MHz** in **Run mode**.
  - When **SSI** works in the master mode, the system clock or the **PIOSC** must be at least two times faster than the **SSInClk**, but the **SSInClk** cannot be faster than 25 MHz.
  - When **SSI** works in the slave mode, the system clock or the **PIOSC** must be at least **12** times faster than the **SSInClk**, but the **SSInClk** cannot be faster than 6.67 MHz.
- The **SSI** includes a programmable **bit rate clock divider** and a **prescaler** to generate the serial output clock. Bit rates are supported to 2 MHz and higher, although maximum bit rate is determined by peripheral devices.



### 8.3.4.2 SSI MODULE BIT RATE GENERATION AND CLOCK CONTROL

- Figure 8.9 shows the bit field and functions of the **SSI Clock Configuration (SSICC) Register**. The lowest 4 bits or bit field **CS** in this register is used to select one of two clock sources.
- When **CS** = **0x0** (0000), the system clock is selected as the clock input to the SSI module. When **CS** = **0x5** (0101), the **PIOOSC** is selected as the clock source. All other values are reserved and not used for this field.
- Two control registers, **SSI Clock Prescale (SSICPSR)** and **SSI Control 0 (SSICR0)**, are used to provide two dividers, **CPSDVSR** and **SCR**, to divide the input clock source (**SysClk**) to get the desired serial transmission/reception clock with appropriate frequency.
- The frequency of the serial output clock **SSIInClk** can be defined as:
  - $\text{SSIInClk} = \text{SysClk} / (\text{CPSDVSR} \times (1 + \text{SCR}))$

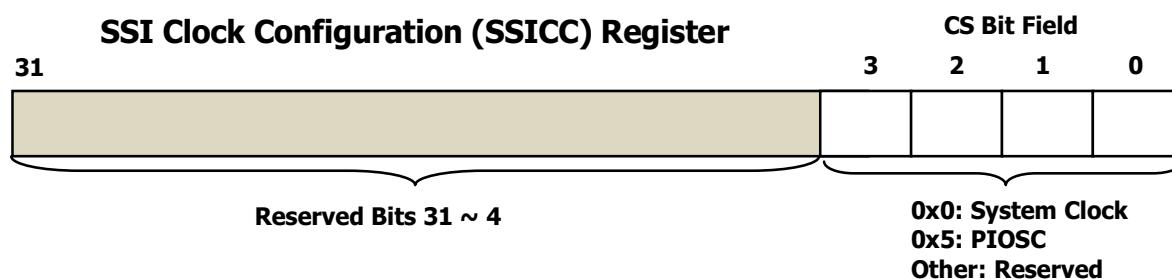


Figure 8.9 The bit field and functions of the SSICC Register.

### 8.3.4.2 SSI MODULE BIT RATE GENERATION AND CLOCK CONTROL

- The **CPSDVSR** is the lowest 8 bits in the **SSICPSR** register. This 8-bit field must contain an even **prescale value** with a range of **2 ~ 254**.
- The **SCR** is an 8-bit field (bits **15 ~ 8**) in the **SSICR0** register and it can provide a dividing value of **0 ~ 255**. Therefore the denominator on the above equation is ranged **(2 ~ 254) × 256 = 512 ~ 65024**.
- This means that the frequency of the serial output clock **SSIInClk** can be ranged from **SysClk/65024 ~ SysClk/512**.
- Figure 8.10 shows the bit field and functions of the **SSI Clock Prescale (SSICPSR)** Register. The lowest 8 bits or bit field **CPSDVSR** provides a divider to divide the input system clock source to get the desired serial output clock with an appropriate frequency. This value must be an even number ranged from **2 to 254**.

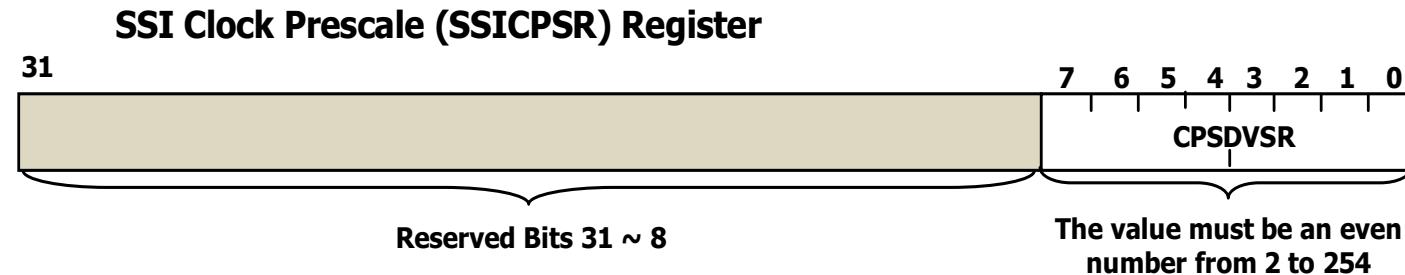


Figure 8.10 The bit field and functions of the SSICPSR Register.

### 8.3.4.2 SSI MODULE BIT RATE GENERATION AND CLOCK CONTROL

- Figure 8.11 shows the bit field and functions of the **SSI Control 0 (SSICR0)** Register.
- Table 8.4 (next slide) shows the bit value and its function for **SSICR0** register.

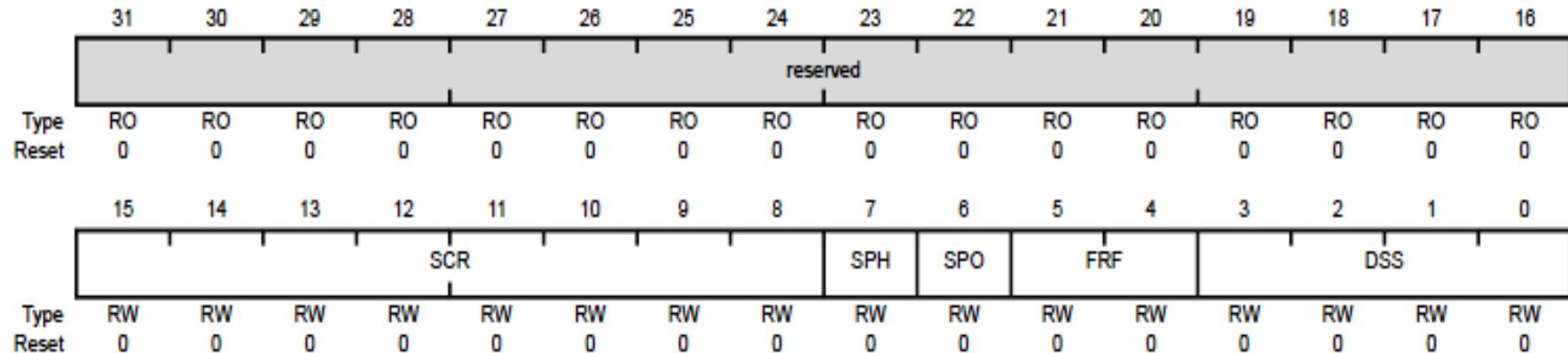


Figure 8.11 The bit field and functions of the SSICR0 Register.

### 8.3.4.2 SSI MODULE BIT RATE GENERATION AND CLOCK CONTROL

Table 8.4 Bit value and its function for SSICR0 register.

Bit	Name	Reset	Function
<b>31:16</b>	<b>Reserved</b>	0x0	Reserved
<b>15:8</b>	<b>SCR</b>	0x0	SSI Serial Clock Rate. This bit field is used to generate the transmit and receive bit rate of the SSI (0 – 255).
<b>7</b>	<b>SPH</b>	0	SSI Serial Clock Phase (Only work for <b>Freescale SPI Frame</b> ). <b>0</b> : Data is captured on the <b>First</b> clock edge transition. <b>1</b> : Data is captured on the <b>Second</b> clock edge transition.
<b>6</b>	<b>SPO</b>	0	SSI Serial Clock Polarity (Only work for <b>Freescale SPI Frame</b> ). <b>0</b> : The inactive level for the <b>SSInClk</b> pin is Low. <b>1</b> : The inactive level for the <b>SSInClk</b> pin is High. If this bit is set, the software must also configure the GPIO port pin corresponding to the <b>SSInClk</b> signal as a pull-up in the <b>GPIO Pull-Up Select (GPIOPUR)</b> register.
<b>5:4</b>	<b>FRF</b>	0x0	SSI Frame Format Select. <b>0x0</b> : Freescale SPI Frame Format, <b>0x1</b> : Texas Instruments Synchronous Serial Frame Format <b>0x2</b> : MICROWIRE Frame Format, <b>0x3</b> : Reserved
<b>3:0</b>	<b>DSS</b>	0x0	SSI Data Size Select. <b>0x0-0x2</b> : Reserved <b>0x3</b> : 4-bit data, <b>0x4</b> : 5-bit data, <b>0x5</b> : 6-bit data, <b>0x6</b> : 7-bit data, <b>0x7</b> : 8-bit data <b>0x8</b> : 9-bit data, <b>0x9</b> : 10-bit data, <b>0xA</b> : 11-bit data, <b>0xB</b> : 12-bit data, <b>0xC</b> : 13-bit data <b>0xD</b> : 14-bit data, <b>0xE</b> : 15-bit data, <b>0xF</b> : 16-bit data

### 8.3.4.3 SSI MODULE CONTROL/STATUS AND FIFO CONTROL

- As we discussed in section 8.3.2, the following control registers belong to SSI module control and status group:
  - **SSI Control 0 Register (SSICR0)**: Setup protocol mode, clock rate and data size.
  - **SSI Control 1 Register (SSICR1)**: Setup master/slave mode.
  - **SSI Status Register (SSISR)**: Provide SSI working status (busy, FIFO empty or full).
  - **SSI Data Register (SSIDR)**: 16-bit register used to store data to be written or read.
  - **Transmit FIFO ( $T_x$ FIFO)**: Store  $8 \times 16$ -bit data to be transmitted.
  - **Receive FIFO ( $R_x$ FIFO)**: Store  $8 \times 16$ -bit data to be received.
- The **SSICR0** register has been discussed in the last section. Let's start our discussion from the **SSICR1** register.



### 8.3.4.3.1 SSI CONTROL 1 REGISTER (SSICR1)

- This register is used to provide additional control for the **SSI** data transmission and reception functions.
- Figure 8.12 shows the bit field and functions of this register.
- Table 8.5 (next slide) shows the bit value and its function of this register.
- Two points to be noted when using this register: 1) the **EOT** bit is only valid when the SSI works as a master device (**MS** = **0x0**), and 2) the **SSE** bit must be cleared to **0** to disable the **SSI** when it is in the initialization or configuration process.

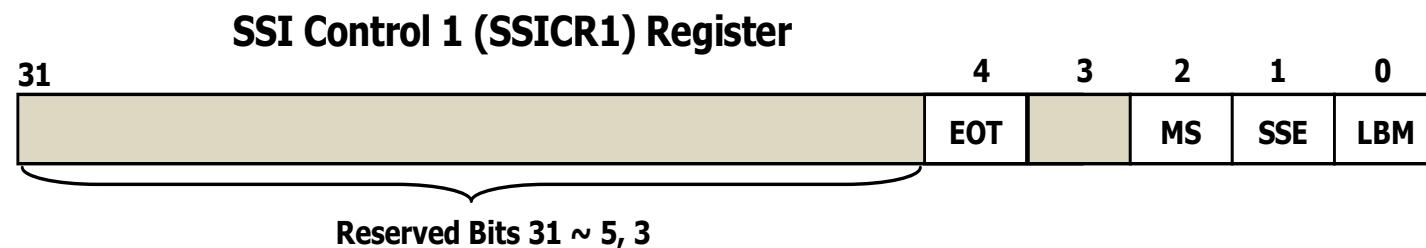


Figure 8.12 The bit field and functions of the SSICR1 Register.

### 8.3.4.3.1 SSI CONTROL 1 REGISTER (SSICR1)

Table 8.5 Bit value and its function for SSICR1 register.

Bit	Name	Reset	Function
31:5	Reserved	0x0	Reserved
4	<b>EOT</b>	0	End Of Transmission. This bit is only valid for Master mode devices and operations ( <b>MS = 0x0</b> ). <b>0:</b> The <b>TXRIS</b> interrupt indicates that the transmit FIFO is half-full or less. <b>1:</b> The End of Transmit interrupt mode for the <b>TXRIS</b> interrupt is enabled.
3	Reserved	0	Reserved.
2	<b>MS</b>	0	SSI Master/Slave Select. Select Master or Slave mode and can be modified only when the SSI is disabled ( <b>SSE=0</b> ). <b>0:</b> The SSI is configured as a Master. <b>1:</b> The SSI is configured as a Slave.
1	<b>SSE</b>	0	SSI Synchronous Serial Port Enable. <b>0:</b> SSI operation is disabled. <b>1:</b> SSI operation is enabled.  This bit must be cleared before any control registers are reprogrammed.
0	<b>LBM</b>	0	SSI Loopback Mode. <b>0:</b> Normal serial port operation enabled. <b>1:</b> Output of the transmit serial shift register is connected internally to the input of the receive serial shift register.

### 8.3.4.3.2 SSI STATUS REGISTER (SSISR)

- This register provides **SSI working status** such as **busy** or **idle** and **FIFO working status** such as empty or full.
  - Figure 8.13 shows the bit field and functions of this register.
  - Table 8.6 (next slide) shows the bit value and its function of this register.
  - The lowest **5** bits on this register monitor and indicate the SSI running status and FIFO operational status.

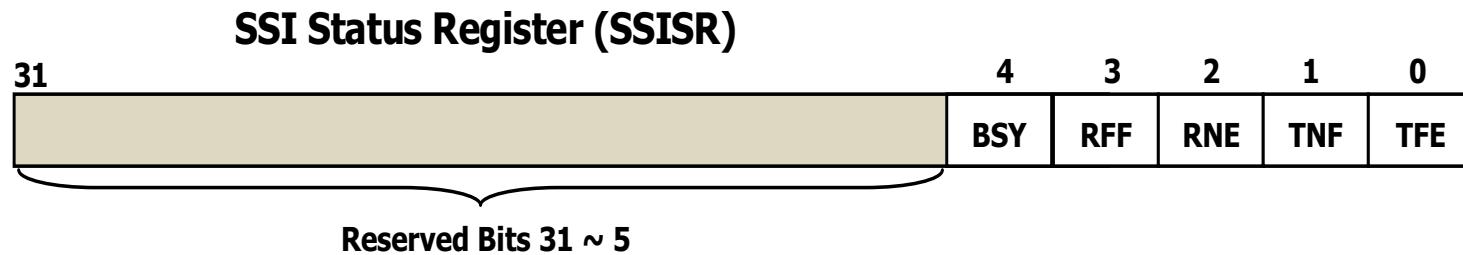


Figure 8.13 The bit field and functions of the SSISR Register.

### 8.3.4.3.2 SSI STATUS REGISTER (SSISR)

Table 8.6 Bit value and its function for SSISR register.

Bit	Name	Reset	Function
31:5	Reserved	0x0	Reserved
4	<b>BSY</b>	0	SSI Busy Bit. <b>0:</b> The SSI is idle. <b>1:</b> The SSI is busy or the transmit FIFO is not empty.
3	<b>RFF</b>	0	SSI Receive FIFO Full. <b>0:</b> The Receive FIFO is not Full. <b>1:</b> The Receive FIFO is Full.
2	<b>RNE</b>	0	SSI Receive FIFO Not Empty. <b>0:</b> The Receive FIFO is Empty. <b>1:</b> The Receive FIFO is not Empty.
1	<b>TNF</b>	1	SSI Transmit FIFO Not Full. <b>0:</b> The Transmit FIFO is Full. <b>1:</b> The Transmit FIFO is not Full.
0	<b>TFE</b>	1	SSI Transmit FIFO Empty. <b>0:</b> The Transmit FIFO is not Empty. <b>1:</b> The Transmit FIFO is Empty

### 8.3.4.3.3 SSI DATA REGISTER (SSIDR)

- This register is used to transmit or receive data to/from the transmit/receive FIFO. Although this is a 32-bit register, only **lower16-bit** (bits **15 ~ 0**) is used to store a data item to match the data-width used in both FIFOs.
- Each FIFO can hold up to **eight 16-bit** data.
- When the **SSIDR** register is read, the entry in the receive FIFO that is pointed to by the current FIFO read pointer is accessed. When a data value is moved by the SSI receive logic from the incoming data frame, it is placed into the entry in the receive FIFO pointed to by the current FIFO write pointer (Figure 8.14a).

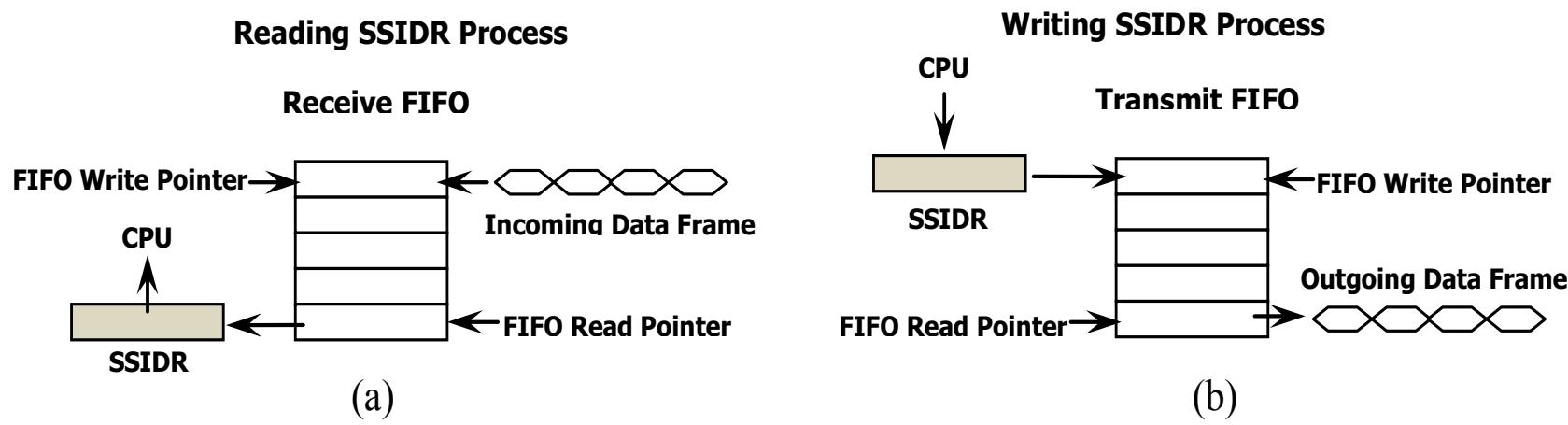


Figure 8.14 The receive FIFO and transmit FIFO operational processes.

### 8.3.4.3.3 SSI DATA REGISTER (SSIDR)

- When a data item is **written** to the **SSIDR** register by processor, the entry in the **transmit FIFO** that is pointed to by the **write pointer** is written to.
- Data values are moved from the transmit FIFO one value at a time by the transmit logic. Each data value is loaded into the transmit serial shifter, then serially shifted out onto the **SSInTx** pin at the programmed bit rate (Figure 8.14b).

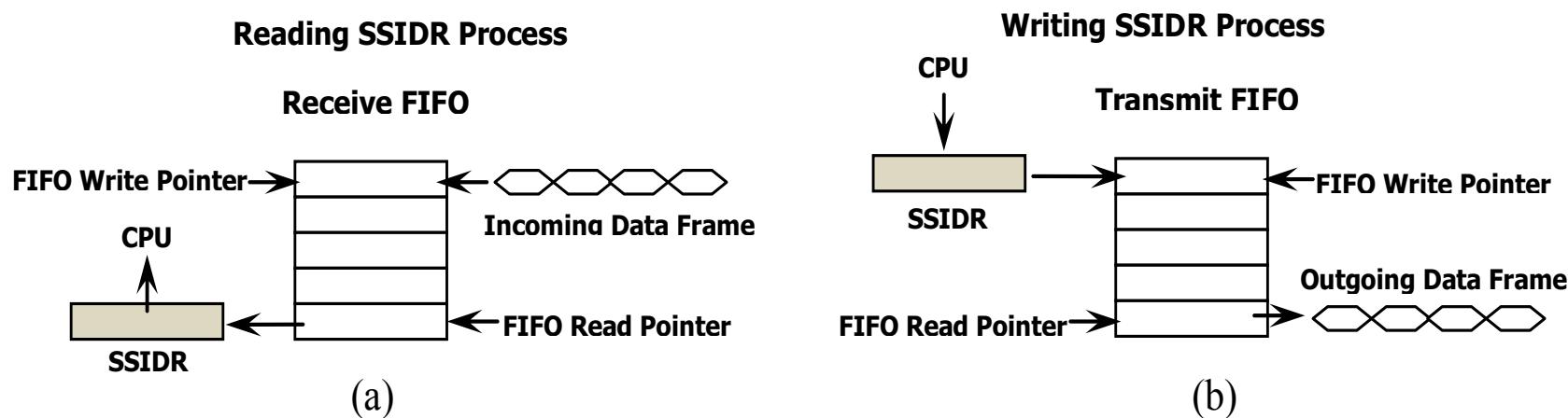


Figure 8.14 The receive FIFO and transmit FIFO operational processes.

### 8.3.4.3.3 SSI DATA REGISTER (SSIDR)

- When a data size of less than **16** bits is selected, the user must right-justify data written to the transmit FIFO. The transmit logic ignores the unused bits.
- Received data that are less than **16** bits are automatically right-justified in the receive buffer.
- When the **SSI** is programmed for using the **MICROWIRE** data frame format, the default size for the transmit data is **eight bits** with the most significant byte being ignored.
- The receive data size is controlled by the programmer. The transmit FIFO and the receive FIFO are not cleared even when the **SSE** bit in the **SSICR1** register is cleared to enable the software to fill the transmit FIFO before enabling the **SSI**.



#### 8.3.4.3.4 FIFO OPERATIONS

- Both transmit FIFO and receive FIFO can hold up to **eight 16-bit** data items with the First In First Out (**FIFO**) operational sequence.
- Both transmit FIFO (**TxFIFO**) and receive FIFO (**RxFIFO**) works as a temporary storage or buffer to store data to be transmitted to or received from the transmit/receive pins **SSInTx** and **SSInRx**. The operational sequence can be summarized as:
  - **Transmit FIFO:** When transmits begin, the CPU sends data to the FIFO by writing the SSI Data (**SSIDR**) register (Figure 8.14b), and data is stored in the FIFO until it is read out by the transmission logic. When configured as a master or a slave, parallel data is first written into the transmit FIFO and then it is serially converted and transmitted to the attached slave or master, respectively, through the **SSInTx** pin.
  - In slave mode, the SSI transmits data each time the master initiates a transaction. If the transmit FIFO is empty and the master initiates, the slave transmits the **8th** most recent value in the transmit FIFO. If less than 8 values are in the transmit FIFO since the SSI module clock was enabled using the **Rn** bit in the Synchronous Serial Interface Run Mode Clock Gating Control (**RCGSSI**) register, a **0** is transmitted. Care should be taken to ensure that valid data is in the FIFO as needed. The SSI can be configured to generate an interrupt or a  $\mu$ DMA request when the FIFO is empty.

### 8.3.4.3.4 FIFO OPERATIONS

- Figure 8.15 shows bit fields and related control values on the **RCGCSSI** register. This register is used to enable and provide clock to the desired SSI module. If the related bit **Rn** is **0**, the corresponding **SSI Module n** is disabled. Otherwise if **Rn** is **1**, the **SSI module n** is enabled.

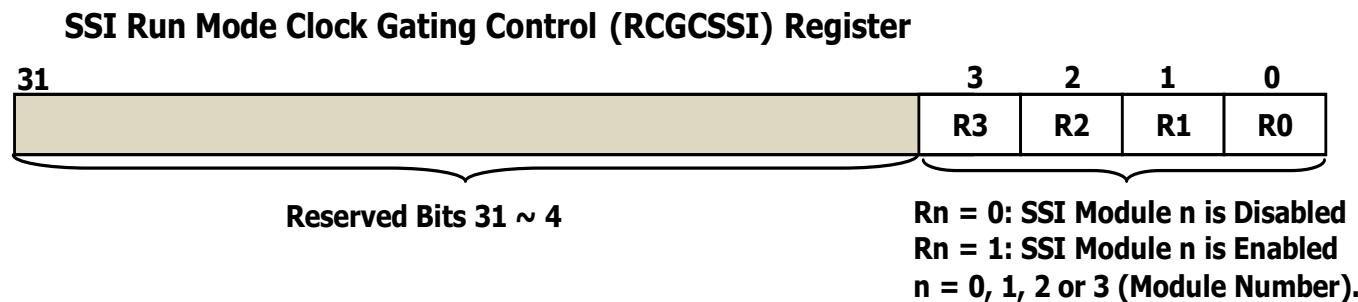


Figure 8.15 The bit field and value in RCGCSSI register.

- Receive FIFO:** Like transmit FIFO, the receive FIFO is used to store the data received from the **SSInRx** pin. The received data from the serial interface is stored in the FIFO until read out by the CPU, which accesses the read FIFO by reading the **SSIDR** register.
- When configured as a master or slave, serial data received through the **SSInRx** pin is shifted and registered prior to parallel loading into the slave or master receive FIFO, respectively.

#### 8.3.4.4 SSI MODULE INTERRUPT AND DMA CONTROL

- As we discussed in section 8.3.2, the following control registers control and monitor the SSI interrupts and DMA operations:
  - **SSI Interrupt Mask Register (SSIIM)**: Enable (unmask)/disable (mask) SSI interrupt.
  - **SSI Raw Interrupt Status Register (SSIRIS)**: Provide SSI raw interrupt status.
  - **SSI Interrupt Clear Register (SSIICR)**: Clear SSI interrupts.
  - **SSI Masked Interrupt Status Register (SSIMIS)**: Provide masked SSI interrupt status.
  - **SSI DMA Control Register (SSIDMACTL)**: Provide DMA control.
- The SSI can generate interrupts when the following conditions are met:
  - Transmit FIFO service/Receive FIFO service (when the transmit/receive FIFO is half full or less).
  - Receive FIFO time-out or overrun.
  - End of transmission.
  - Receive DMA transfer complete.
  - Transmit DMA transfer complete.

#### 8.3.4.4 SSI MODULE INTERRUPT AND DMA CONTROL

- All of these interrupt events can be **ORed** together before being sent to the NVIC interrupt controller, thus the SSI generates a single interrupt request to the controller regardless of the number of active interrupts.
- The **receive FIFO** has a time-out period, which is **32** clock periods of the SSI clock **SSIInClk**, whether or not it is currently active. This time-out starts when the **RxFIFO** goes from **EMPTY**.
- If this time-out period is less than **32** clocks, the time-out period is reset. As a result, the ISR should clear the receive FIFO time-out interrupt just after reading out the **RxFIFO** by writing **1** to the **RTIC** bit in the **SSIICR** register.
- The End-of-Transmission (**EOT**) interrupt indicates that the data has been transmitted completely and is only valid for **Master Mode** devices and operations. This interrupt can be used to indicate the time when it is safe to turn off the SSI module clock or enter sleep mode. In addition, because transmitted data and received data complete at exactly the same time, the interrupt can also indicate that the data is ready to be read immediately, without waiting for the receive FIFO time-out period to complete.

#### 8.3.4.4 SSI MODULE INTERRUPT AND DMA CONTROL

- The operational sequence of the SSI interrupts are:
  - The related interrupt sources are setup by configuring the corresponding bits in the related registers, such as **SSI Status Register (SSISR)** used to control the status of the FIFO, either empty or full, **SSI Control 1 (SSICR1)** register used to monitor the **EOT** status, to allow them to work as interrupt sources.
  - Set related bits in the **SSI Interrupt Mask (SSIIM)** register to enable selected SSI interrupts. When interrupt events occurred, the related bits in the **SSI Raw Interrupt Status (SSIRIS)** register are set to **1** to indicate this situation.
  - Whether these interrupts indicated in the **SSIRIS** register can be sent to the NVIC interrupt controller or not, it depends on the bit values in the **SSIIM** register. If a bit is **1**, which means that the related interrupt is enabled, it can be transferred to the NVIC. Otherwise if a bit is cleared to **0**, it has been masked and cannot be sent out.
  - If interrupts have been accepted and handled in the ISR, the **SSI Interrupt Clear Register (SSIICR)** should be used to clear those interrupts, such as SSI receive time-out or overrun interrupt. Some other interrupts can be automatically cleared when the interrupt sources are recovered or modified, such as the FIFO's status being modified.
  - The **SSI Masked Interrupt Status Register (SSIMIS)** provides the interrupt status of masked interrupt sources.

#### 8.3.4.4.1 SSI INTERRUPT MASK REGISTER (SSIIM)

- Figure 8.16 shows the bit field and functions of the **SSIIM** register. Table 8.7 shows the bit value and its function of this register.

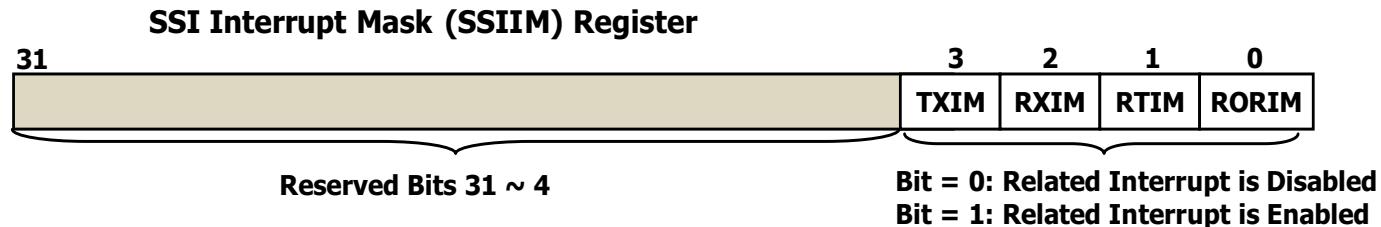


Figure 8.16 The bit field and value in SSIIM register.

Table 8.7 Bit value and its function for SSIIM register.

<b>Bit</b>	<b>Name</b>	<b>Reset</b>	<b>Function</b>
<b>31:4</b>	<b>Reserved</b>	0x0	Reserved
<b>3</b>	<b>TXIM</b>	0	SSI Transmit FIFO Interrupt Mask. <b>0:</b> The transmit FIFO interrupt is masked (Disabled). <b>1:</b> The transmit FIFO interrupt is not masked (Enabled).
<b>2</b>	<b>RXIM</b>	0	SSI Receive FIFO Interrupt Mask. <b>0:</b> The receive FIFO interrupt is masked (Disabled). <b>1:</b> The receive FIFO interrupt is not masked (Enabled).
<b>1</b>	<b>RTIM</b>	0	SSI Receive Time-Out Interrupt Mask. <b>0:</b> The receive FIFO time-out interrupt is masked (Disabled). <b>1:</b> The receive FIFO time-out interrupt is not masked (Enabled).
<b>0</b>	<b>RORIM</b>	0	SSI Receive Overrun Interrupt Mask. <b>0:</b> The receive FIFO overrun interrupt is masked (Disabled). <b>1:</b> The receive FIFO overrun interrupt is not masked (Enabled).

### 8.3.4.4.2 SSI RAW INTERRUPT STATUS REGISTER (SSIRIS)

- This register is used to monitor and indicate which raw interrupt status has been modified and generated. Similar to **SSIIM** register, the lowest 4 bits on this register are used to monitor 4 different interrupt sources or events.
- Figure 8.17 shows the bit field and functions of the **SSIRIS** register. Table 8.8 (next slide) shows the bit value and its function of this register.
- After a raw interrupt status has been changed or a raw interrupt has been set in the register, the bit value on the corresponding bit in the **SSIIM** register determines whether this raw interrupt can be sent to the NVIC or not. A value of **1** on the related bit in the **SSIIM** indicates that the selected interrupt can be sent to the NVIC interrupt controller.

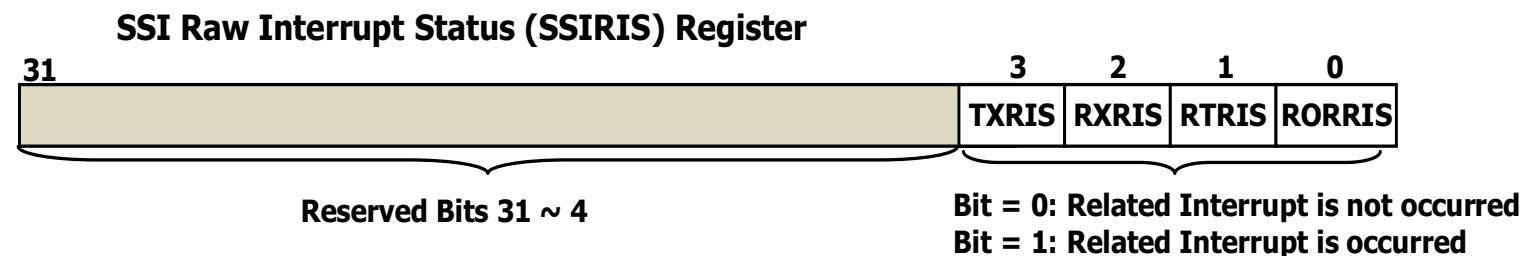


Figure 8.17 The bit field and value in SSIRIS register.

## 8.3.4.4.2 SSI RAW INTERRUPT STATUS REGISTER (SSIRIS)

Table 8.8 Bit value and its function for SSIRIS register.

Bit	Name	Reset	Function
<b>31:4</b>	<b>Reserved</b>	0x0	Reserved
<b>3</b>	<b>TXRIS</b>	1	<p>SSI Transmit FIFO Raw Interrupt Status.</p> <p><b>0:</b> No interrupt.</p> <p><b>1:</b> If the <b>EOT</b> bit in the <b>SSICR1</b> register is clear, the transmit FIFO is half empty or less. If the <b>EOT</b> bit is set, the transmit FIFO is empty, and the last bit has been transmitted out of the serializer.</p> <p>This bit is cleared when the transmit FIFO is more than half full (if the <b>EOT</b> bit is clear) or when it has any data in it (if the <b>EOT</b> bit is set).</p>
<b>2</b>	<b>RXRIS</b>	0	<p>SSI Receive FIFO Raw Interrupt Status.</p> <p><b>0:</b> No interrupt.</p> <p><b>1:</b> The receive FIFO is half full or more.</p> <p>This bit is cleared when the receive FIFO is less than half full.</p>
<b>1</b>	<b>RTRIS</b>	0	<p>SSI Receive Time-Out Raw Interrupt Status.</p> <p><b>0:</b> No interrupt.</p> <p><b>1:</b> The receive time-out interrupt occurred.</p> <p>This bit is cleared by writing 1 to the <b>RTIC</b> bit in the <b>SSIICR</b> register.</p>
<b>0</b>	<b>RORRIS</b>	0	<p>SSI Receive Overrun Raw Interrupt Status.</p> <p><b>0:</b> No interrupt.</p> <p><b>1:</b> The receive FIFO has an overflow.</p> <p>This bit is cleared by writing 1 to the <b>RORIC</b> bit in the <b>SSIICR</b> register.</p>

### 8.3.4.4.3 SSI DMA CONTROL REGISTER (SSIDMACTL)

- This register is used to control the availability of the DMA for the transmit FIFO or receive FIFO.
- The lowest two bits, **TXDMAE** and **RXDMAE** (bits **1 ~ 0**), are used to enable (**1**) or disable (**0**) the DMA as a conveyer for the transmit FIFO or the receive FIFO. The bits **31 ~ 2** in this register are reserved.
- The DMA module should be enabled when using this mode to work with the SSI modules.

### 8.3.4.5 SSI MODULE TRANSMIT/RECEIVE LOGIC CONTROL

- The **SSI transmit/receive logic** control block is used to assist SSI module to effectively transmit/receive data via serial communication pins/lines. The main function of this block includes:
  - For **transmit operations**, perform **parallel-to-serial conversion** by using the transmit shift register to convert parallel data stored in the transmit FIFO to the serial format.
  - **Move or transmit** serial data to the transmission pin and line **SSInTx** based on the **SSInClk** and **SSInFss** frame.
  - For **receive operations**, perform **serial-to-parallel conversion** by using the receive shift register to convert serial data coming from transmission pins **SSInRx** to the parallel format that can be stored in the receive FIFO based on the **SSInClk** and **SSInFss** frame.
  - Besides the data mode conversions, this block also performs the voltage level conversion to transfer the **TTL** logic level to the standard serial communication voltage levels.
- At this point, we have completed our discussions about the SSI signals and components. Now let's take a look at the initialization and configuration process for the SSI modules before they can be implemented in the real applications.

### 8.3.4.6 SSI MODULES INITIALIZATION AND CONFIGURATIONS

- Before the **SSI module** can be used for any peripherals, it must be initialized and configured based on the data transmission/reception requirements for the selected peripheral device.
- During the initialization and configuration process, the selected SSI module must not be enabled. In other words, prior to this initialization process, the SSI module must be disabled.
- This initialization and configuration process can be divided into three parts:
  - **Initialize and configure the GPIO Ports and pins related to SSI modules.**
  - **Initialize and configure SSI module.**
  - **Initialize and configure SSI clock source and bit rate.**
- Let's have a closer look at these initializations one by one in the following sections.



### 8.3.4.6.1 SSI MODULE RELATED GPIO PORTS INITIALIZATION

- In order to enable GPIO pins to work as **SSI** function pins, the following initializations and configurations should be performed:
  1. Enable the clock to the appropriate GPIO modules via the **RCGCGPIO** register.
  2. The related **AFSEL** bits (lower 8-bit) in the **GPIOAFSEL** register must be set to **1**.
  3. The related **PMCx** field in the **GPIOPCTL** register must be assigned with appropriate codes to enable the GPIO pins to work as desired **SSI** signal pins.
  4. The related bits in the GPIO Digital Enable (**GPIODEN**) register must be set to **1** to enable the digital function and disable the analog function. In addition, the drive strength, drain select and pull-up/pull-down functions must be configured.
  5. Optionally the related **AMSEL** bits in the **GPIOAMSEL** register should be clear to **0** to activate the analog isolation circuits to enable the selected pin to work as digital function pin.
- When performing step **4**, the pull-up mode is used to avoid unnecessary toggles on the SSI pins since this toggle can take the slave to a wrong state.
- If the **SSIClk** signal is programmed to steady state **High** through the **SPO** bit (**SPO = 1**) in the **SSICR0** register, the software must also configure the GPIO pin corresponding to the **SSInClk** signal as a pull-up in the **GPIOPUR** register.

### 8.3.4.6.2 SSI MODULE INITIALIZATION AND CONFIGURATION

- To enable and initialize the **SSI module**, the following steps are necessary:
  1. Enable the **SSI module** using the **RCGCSSI** register (Figure 8.15 in section 8.3.4.3.4).
  2. Ensure that the **SSE** bit in the **SSICR1** register is clear before making any initialization and configuration changes (Figure 8.12 in section 8.3.4.3.1).
  3. Select the **SSI** to work as a master or slave by configuring the **SSICR1** register.
    - If the SSI works as a master device, set the **SSICR1** register to **0x0000.0000**.
    - If the SSI works as a slave device with output enabled, set the **SSICR1** register as **0x0000.0004**.
    - If the SSI works as a slave device with output disabled, set the **SSICR1** register as **0x0000.000C**.
- Next let's have the SSI clock source and bit rate to be configured and initialized.

### 8.3.4.6.3 SSI MODULE CLOCK SOURCE & BIT RATE INITIALIZATION

- To initialize and configure **SSI module clock source** and select the **bit rate**, the following operations are necessary:
  - Select the **SSI clock source** by configuring the **SSICC** register (Figure 8.9 in 8.3.4.2).
  - Define the **clock prescale divisor** by configuring the **SSICPSR** register to select the desired bit-field value **CPSDVSR = [2 ~ 254]** (Figure 8.10 in section 8.3.4.2).
  - Configure the **SSICRo** register with the following parameters:
    - Serial **Clock Rate (SCR)** = **[0 ~ 255]**
    - Desired **Clock Phase/Polarity** if using Freescale SPI mode (**SPH** and **SPO**)
    - The **Protocol Mode** - Freescale SPI, TI SSF, MICROWIRE (FRF)
    - The Data Size (**DSS**)
  - Optionally, configure the **SSI module for μDMA** use with the following steps:
    - Configure a μDMA for SSI use.
    - Enable the SSI Module's **TxFIFO** or **RxFIFO** by setting the **TXDMAE** or **RXDMAE** bit in the **SSIDMACTL** register (refer to section 8.3.4.4.3).
  - Enable the configured SSI by setting the **SSE** bit in the **SSICR1** register.



### 8.3.4.6.3 SSI MODULE CLOCK SOURCE & BIT RATE INITIALIZATION

- A **SSI module 2** initialization example is shown below with the assumptions:
  - The system clock source (**SysClk**) is **20 MHz**.
  - The SSI module 2 is selected and worked as a **master** device.
  - Use **Freescale SPI frame (SPO = 0, SPH = 0)**.
  - Use **1 MBPS** bit rate and **8** data bits.
- The bit rate is calculated as: **SSInClk = SysClk / (CPSDVSR × (1 + SCR))** or
  - **$1 \times 10^6 = 20 \times 10^6 / (\text{CPSDVSR} \times (1 + \text{SCR}))$**
- In this case, if **CPSDVSR = 0x2**, **SCR** should be **0x9**. Then:
  - Clear the **SSE** bit in the **SSICR1** register to disable the SSI module.
  - Write the **SSICC** register **0x0000.0000** to select the system clock source.
  - Write the **SSICR1** register **0x0000.0000** to define the SSI to work as a master.
  - Write the **SSICPSR** register **0x0000.0002** to define the **CPSDVSR** bit-field **0x2**.
  - Write the **SSICR0** register **0x0000.0903** to set **SCR** to **0x9**, **SPO:SPH = 00**, **FRF = 00** (Freescale SPI frame) and **DSS = 0x7** (8 data bits).
  - Enable the configured SSI module by setting the **SSE** bit in the **SSICR1** register.

## 8.3.5 BUILD THE ON-BOARD LCD INTERFACE PROGRAMMING PROJECT

- In this example project, we use the **SSI Module 2 (SSI2)** to interface an on-board **16×2 LCD** device installed in the EduBASE ARM® Trainer to display some interesting staff.
- First let's have a closer look at the hardware configuration and connection for this LCD via SSI2.

### 8.3.5.1 SSI MODULE INTERFACE FOR LCD IN EDUBASE ARM TRAINER

- A High-Speed CMOS shift register **74VHCT595** is used as an interface to the LCD from the **SSI2** in the TM4C123GH6PM MCU system.
- A functional block diagram of this interface circuit connected to the LCD installed in the EduBASE ARM® Trainer is shown in Figure 8.18 (next slide).



### 8.3.5.1 SSI MODULE INTERFACE FOR LCD IN EDUBASE ARM TRAINER

- It can be found from Figure 8.18 that the SSI module 2 is connected to the **HCOMS** shift register **74VHCT595** via GPIO Ports B, exactly **PB7**, **PB6** and **PB4** pins.

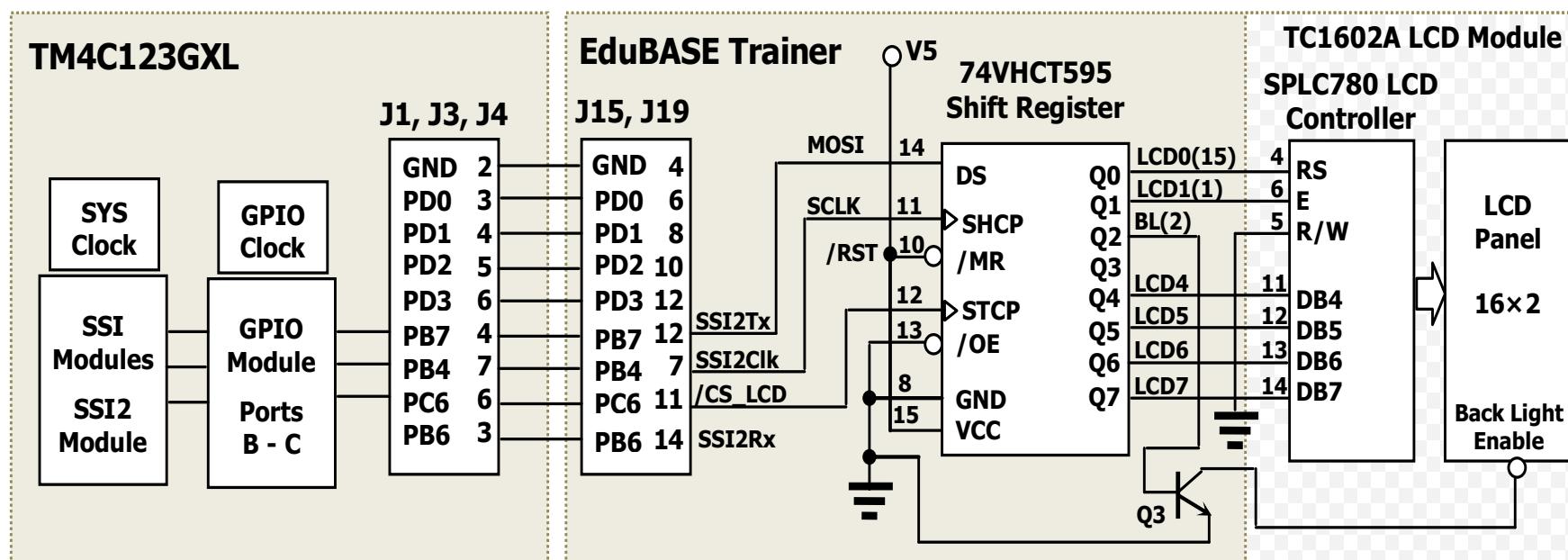


Figure 8.18 Hardware configuration of the LCD interfacing circuit.

### 8.3.5.1 SSI MODULE INTERFACE FOR LCD IN EDUBASE ARM TRAINER

- The functions of these pins are (Figure 8.18):
  - **PB7: SSI2Tx** - Transmit serial signals to the peripheral **74VHCT595**.
  - **PB4: SSI2Clk** – Provide output clock signal to coordinate the serial data transfers.
  - **PB6: SSI2Rx** – Receive serial data from the peripheral ([not used in this example](#)).
  - **PB5: SSI2Fss** – Provide the serial data transmit frame ([not used in this example](#)).
  - **PC6: /CS\_LCD** – Provide the LCD Chip Select signal. Here it works as a trigger clock to start a serial-to-parallel data conversion to convert 8-bit serial data stored in the serial shift register to the 8-bit parallel data to be stored in the 8-bit output registers in the **74VHCT595** register.
- In this configuration, the **SSI2** output is connected to the **LCD** via a serial shift register **74VHCT595** and a **LCD Controller SPLC780**.
- The data size to be transmitted from the **SSI2** to the LCD is **4 bits**. Therefore an 8-bit data must be broken into two sections with **4** bits for each section.
- The higher 4-bit (**DB7 ~ DB4**) are **LCD data** and the lower 3-bit are **LCD control signals**, **RS** (Register Select), **E** (Enable) and **BL** (Back Light Enable). The **R/W** control signal is connected to the ground to make the LCD controller work in the **writing mode**.

### 8.3.5.2 THE SERIAL SHIFT REGISTER 74VHCT595

- **74VHCT595** is an **8-bit serial-in-serial-out or serial-in-parallel-out** register with output latches. In this example, we use it as a **serial-in-parallel-out** register to convert the serial input coming from the **SSI2** to the parallel output, the latter is feed into the LCD controller to display the result on the LCD panel.
- The operational principle and sequence of this register is:
  - Data is shifted into the serial input pin (**DS**) on the positive-going edges of the shift register clock input (**SHCP**).
  - The data in each flip-flop register is transferred to the storage register on a positive-going edge of the storage register clock input (**STCP**). If both clocks are connected together, the shift register will always be one clock pulse ahead of the storage register.
  - The shift register has a serial input (**DS**) and a serial standard output (**Q7S**) for cascading.
  - It is also provided with asynchronous master reset (**MR** – active LOW) for all 8 shift flip-flop registers. The storage register has 8 parallel 3-state bus driver outputs. Data in the storage register appears at the output whenever the output enable input (**OE**) is **LOW**.
- Figure 8.19 (next slide) shows the functional block diagram of this register.

### 8.3.5.2 THE SERIAL SHIFT REGISTER 74VHCT595

- Figure 8.19 shows the functional block diagram of this register.

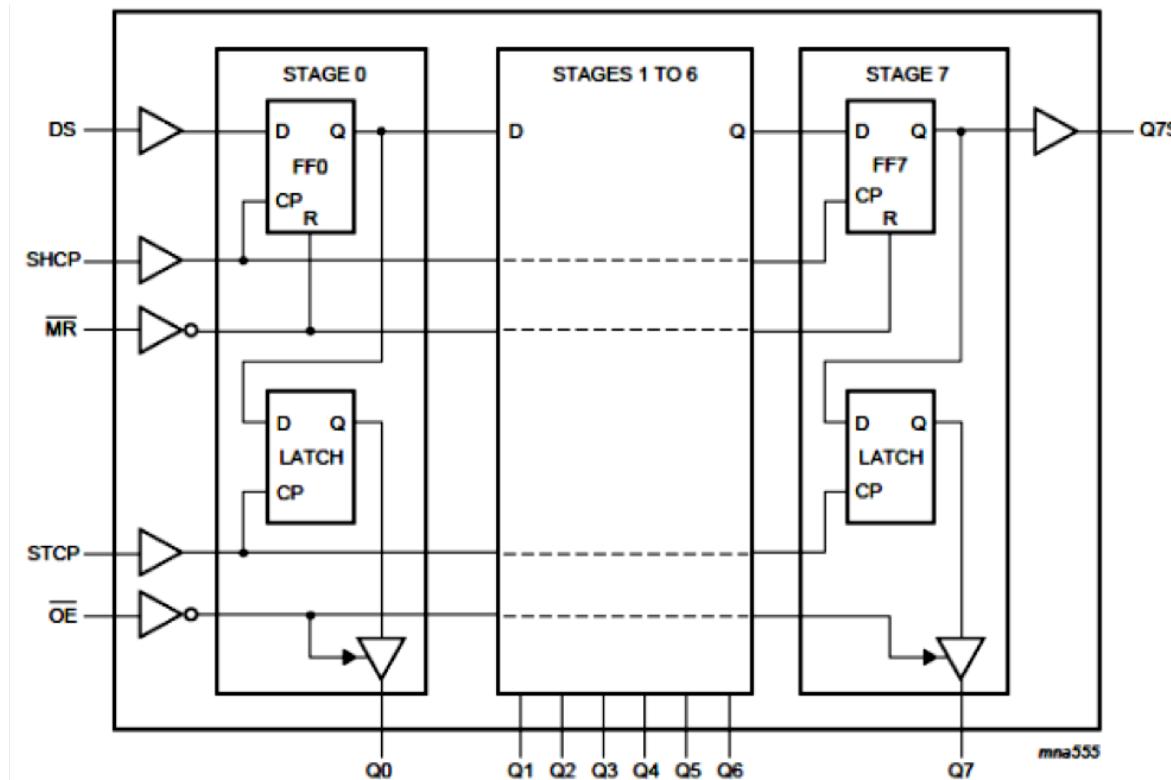


Figure 8.19 The functional block diagram of 74VHCT595 shift register.

### 8.3.5.2 THE SERIAL SHIFT REGISTER 74VHCT595

- The **STAGES 1 TO 6** in Figure 8.19 indicates that this stage is composed of 6 shift flip-flops, with total 8 flip-flops. All 8 parallel outputs are latched and controlled by the Output Enable (**OE**) signal.
- In our example application, each time after **SSI2** transmits 8 bits data to this shift register via **DS (SSI2Tx – PB7)** pin and **SHCP (SSI2Clk – PB4)** pin, a positive-going-edge (Low-to-High) signal should be applied on the **STCP (/CS\_LCD – PC6)** pin to transfer 8-bit serial data in the shift register into the 8-bit parallel output buffer or latches.
- The **OE** pin is connected to the ground to enable 8 latched outputs to be directly transferred to the output pins (refer to Figure 8.18).

### 8.3.5.3 THE LCD MODULE TC1602A AND LCD CONTROLLER SPLC780

- The **TC1602A** LCD Module is controlled and driven by a LCD Controller **SPLC780**, which is a dot-matrix liquid crystal display (LCD) controller and driver to control and display alpha-numeric, Japanese kana characters, and symbols in a **16 × 2** LCD device.
- It can be configured to drive a dot-matrix liquid crystal display under the control of a **4**- or **8**-bit microprocessor.
- Since all the functions such as display RAM, character generator, and liquid crystal driver, required for driving a dot-matrix liquid crystal display are internally provided on one chip, a minimal system can be interfaced with this controller/driver.
- In TM4C123GH6PM MCU system, the **SPLC780** is configured to interface with the MCU with **4-bit** style, therefore we will concentrate on this style in our following discussions.
- An internal structure block diagram of the **SPLC780** is shown in Figure 8.20 (next slide).

### 8.3.5.3 THE LCD MODULE TC1602A AND LCD CONTROLLER SPLC780

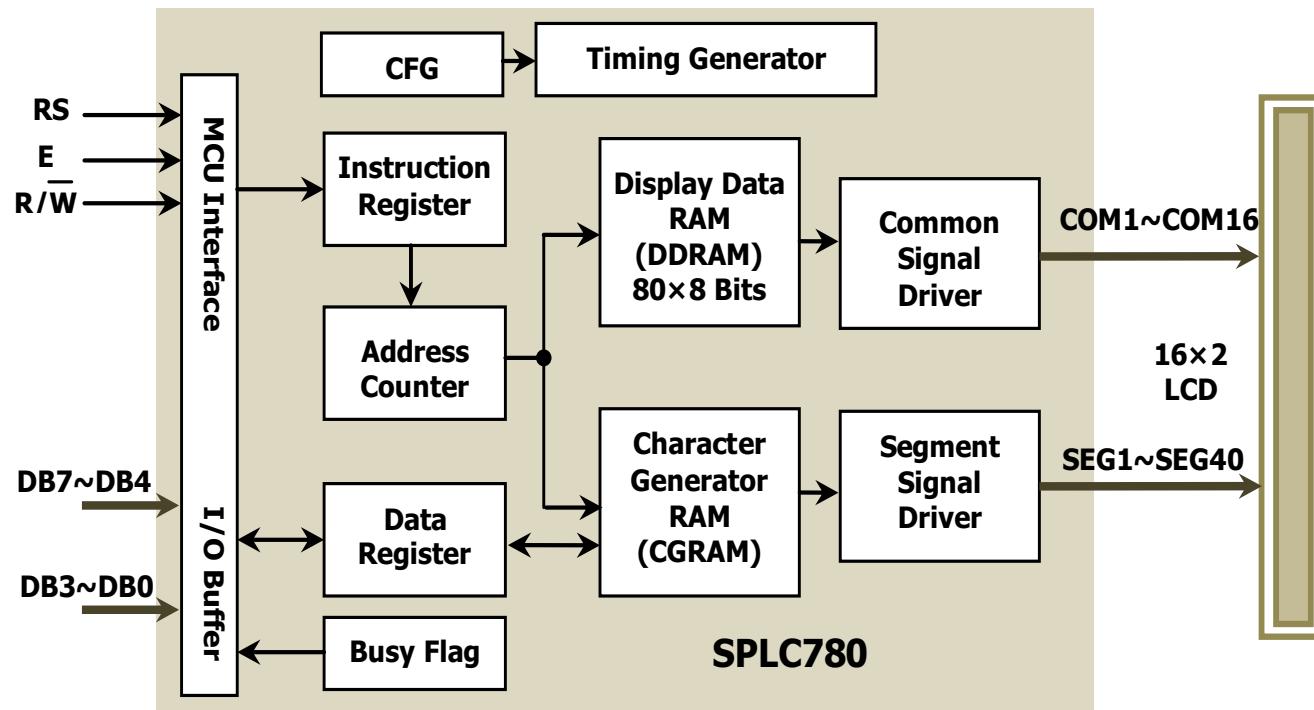


Figure 8.20 Functional block diagram of the LCD Controller SPLC780.

- The **SPLC780** provides a full controllability to the LCD. Some most important and popular components with the associated controlling and interfacing abilities to the LCD are listed below:

### 8.3.5.3 THE LCD MODULE TC1602A AND LCD CONTROLLER SPLC780

- **Instruction Register (IR) and Data Register (DR)**

The **SPLC780** has two 8-bit registers, instruction register (**IR**) & data register (**DR**).

The **IR** stores instruction codes, such as display clear and cursor shift, and address information for display data RAM (**DDRAM**) and character generator RAM (**CGRAM**). The **IR** can only be written from the MPU.

The **DR** temporarily stores data to be written into **DDRAM** or **CGRAM** and temporarily stores data to be read from **DDRAM** or **CGRAM**.

Data written into the **DR** from the MPU is automatically written into **DDRAM** or **CGRAM** by an internal operation. The **DR** is also used for data storage when reading data from **DDRAM** or **CGRAM**.

When address information is written into the **IR**, data is read and then stored into the **DR** from **DDRAM** or **CGRAM** by an internal operation. Data transfer between the MPU is then completed when the MCU reads the **DR**.

After the read, data in **DDRAM** or **CGRAM** at the next address is sent to the **DR** for the next read from the MCU. By the register selector (**RS**) signal, these two registers can be selected as shown in Table 8.9.

Table 8.9 Register selection.

RS	R/W	Function
0	0	Write an instruction to the SPLC780
0	1	Read an instruction from the SPLC780
1	0	Write a data item to the SPLC780
1	1	Read a data item from the SPLC780



### 8.3.5.3 THE LCD MODULE TC1602A AND LCD CONTROLLER SPLC780

- **Busy Flag (BF)**

When the busy flag is **1**, the **SPLC780** is in the internal operation mode, and the next instruction will not be accepted. When **RS = 0** and **R/W = 1** (Table 8.9), the busy flag is output to **DB7**. The next instruction must be written after ensuring that the busy flag is **0**.

- **Address Counter (AC)**

The address counter (**AC**) assigns addresses to both **DDRAM** and **CGRAM**. When an address of an instruction is written into the **IR**, the address information is sent from the **IR** to the **AC**. Selection of either **DDRAM** or **CGRAM** is also determined concurrently by the instruction. After writing into (reading from) **DDRAM** or **CGRAM**, the **AC** is automatically incremented by **1** (decremented by **1**). The **AC** contents are then output to **D<sub>0</sub>** to **D<sub>6</sub>** when **RS = 0** and **R/W = 1** (Table 8.9).

- **Display Data RAM (DDRAM)**

**DDRAM** stores display data represented in 8-bit character codes. Its extended capacity is **80×8** dots or bits, or **80** characters. If each character is composed of **5×8** dots, each line on the LCD can display **80/5 = 16** characters. The area in **DDRAM** that is not used for display can be used as general data RAM.

Figure 8.21 (next slide) shows the relationships between **DDRAM** addresses and **positions** on the 2-line by 16 characters LCD.



### 8.3.5.3 THE LCD MODULE TC1602A AND LCD CONTROLLER SPLC780

- The DDRAM address (**ADD**) is set in the address counter (**AC**) as hexadecimal. In TM4C123GH6PM MCU system, a  $16 \times 2$  LCD is used, which means that this LCD can display 2 lines characters with 16 characters being displayed at each line.

Display Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
DDRAM Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

Figure 8.21 Relationship between the display positions and DDRAM addresses.

- Character Generator ROM (CGROM)**

The **CGROM** generates **5×8** dots or **5×10** dots character patterns from 8-bit character codes. It can generate **208**  $5 \times 8$  dots character patterns and **32**  $5 \times 10$  dots character patterns. User-defined character patterns are also available by mask-programmed ROM.

- Character Generator RAM (CGRAM)**

- In the **CGRAM**, the user can rewrite character patterns by programming. For **5×8** dots, **eight** character patterns can be written, and for **5×10** dots, **four** character patterns can be written.

### 8.3.5.3.1 INTERFACING CONTROL SIGNALS BETWEEN MCU & SPLC780

- In summary, **three** important signals are widely used in the interfacing process between a **MCU** and the **SPLC780**:
  - **Register Selection RS** signal: When **RS = 0**, it means that currently an **instruction** is transferred from the MCU to the SPLC780. If **RS = 1**, it means that currently a **data** item is reading from or writing into the SPLC780.
  - **Read/Write R/W** signal: When this signal is **0**, it means that a **writing** operation is performed from the MCU to the SPLC780. If this bit is **1**, it means that a **reading** operation is performed by reading a data item including the Address Counter (AC) from the SPLC780 to the MCU.
  - **Enable E** signal: When this signal is **0**, it means that the SPLC780 is **disabled** and no matter what kind of instructions or data are read from or written into the SPLC780, no information can be obtained from the SPLC780 since it is disabled. If this bit is **1**, it means that the SPLC780 is **enabled** and it can accept any control signal with the appropriate responses.



### 8.3.5.3.1 INTERFACING CONTROL SIGNALS BETWEEN MCU & SPLC780

- The **SPLC780** can send data in either **two 4-bit** operations or **one 8-bit** operation, thus allowing interfacing with **4-bit** or **8-bit** MCUs.
  - For **4-bit** interface data, only **four higher bits (DB4 ~ DB7)** are used for transfer. Bus lines **DB0 ~ DB3** are disabled. The data transfer between the SPLC780 and the MCU is completed after the **4-bit** data has been transferred twice.
  - As for the order of data transfer, the four high order bits (**DB4 ~ DB7**) are transferred before the four low order bits (**DB0 ~ DB3**) can be transferred.
  - The **busy flag** must be checked after the **4-bit** data has been transferred twice. Two more 4-bit operations then transfer the busy flag and address counter data.
  - For **8-bit** interface data, all eight bus lines (**DB0 ~ DB7**) are used.
- Since we are using TM4C123GH6PM MCU system with the EduBASE ARM® Trainer, in which a **4-bit** interfacing style is used, therefore we will concentrate on this **4-bit** interfacing technique.

### 8.3.5.3.1 INTERFACING CONTROL SIGNALS BETWEEN MCU & SPLC780

- The operational sequence of interfacing between the MCU and the SPLC780 is:
  1. The LCD must be **initialized** first by performing a reset operation. Regularly this reset can be performed by the SPLC780 itself via an internal reset process when it is powered up. However, if this reset is not as good as desired, the user must perform an external reset operation by programming the SPLC780 to do that reset operation. This reset process includes the following operations:
    - a. **Clear** the display area in the LCD.
    - b. **Set** the display function by setting the related bit with the appropriate values:
      - **DL = 0:** 4-bit interface data, **DL = 1:** 8-bit interface data.
      - **N = 0:** 1-Line display, **N = 1:** 2-Line display.
      - **F = 0:**  $5 \times 8$  dots character font, **F = 1:**  $5 \times 10$  dots character font.
    - c. Set the display **On/Off** control by setting the related bit with the appropriate values:
      - **D = 0:** Display Off, **D = 1:** Display On.
      - **C = 0:** Cursor Off, **C = 1:** Cursor On.
      - **B = 0:** Blinking Off, **B = 1:** Blinking On.



### 8.3.5.3.1 INTERFACING CONTROL SIGNALS BETWEEN MCU & SPLC780

- d. Select the entry mode by setting the related bit with the appropriate values:
  - o **I/D = 0**: Address Counter (**AC**) is **decremented** by **1** after each operation,  
**I/D = 1**: Address Counter (**AC**) is **incremented** by **1** after each operation.
  - o **S = 0**: No shift, **S = 1**: Shift character.
  - o **S/C = 0**: Cursor move, **S/C = 1**: Display shift.
  - o **R/L = 0**: Shift to the left, **R/L = 1**: Shift to the right.
2. After the reset process, perform the **Instruction Writing operation** to set the internal **DDRAM address** from which the data can be written into or read out for SPLC780.
3. To display characters in the LCD, perform the **Data Writing operations** to send the data to the **DDRAM** in the SPLC780.
4. After each 8-bit character writing (**two 4-bit writings**), an **Instruction Read (IR)** should be performed to check the **Busy Flag (BF)** before the next data can be sent to the SPLC780.



### 8.3.5.3.1 INTERFACING CONTROL SIGNALS BETWEEN MCU & SPLC780

- All of these initialization parameters and their values are shown in Table 8.10.

Table 8.10 LCD initialization functions.

Bit	Value	Functions	Bit	Value	Functions
<b>DL</b>	0	4-bit interface data	<b>C</b>	0	Cursor Off
	1	8-bit interface data		1	Cursor On
<b>N</b>	0	1-Line display	<b>B</b>	0	Blinking Off
	1	2-Line display		1	Blinking On
<b>F</b>	0	5×8 dots character font	<b>I/D</b>	0	Decrement by 1
	1	5×10 dots character font		1	Increment by 1
<b>D</b>	0	Display Off	<b>S</b>	0	No shift
	1	Display On		1	Shift character
<b>S/C</b>	0	Cursor move	<b>R/L</b>	0	Shift to the left
	1	Display shift		1	Shift to the right

### 8.3.5.3.1 INTERFACING CONTROL SIGNALS BETWEEN MCU & SPLC780

- A block diagram of this operational sequence is shown in Figure 8.22.

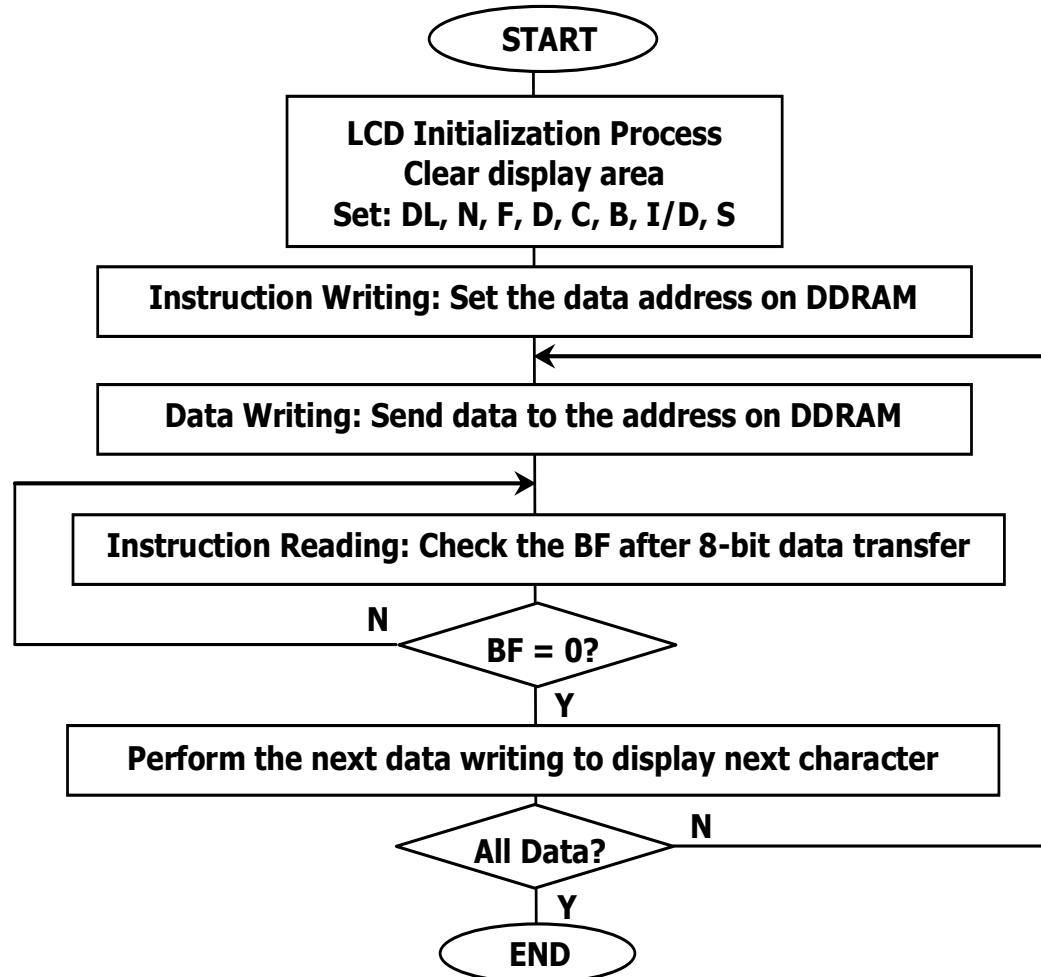


Figure 8.22 Block diagram of the operational sequence of interfacing to SPLC780.

### 8.3.5.3.2 CONTROL AND INTERFACE PROGRAMMING FOR SPLC780

- When the **SPLC780** is connected to a MCU by using **4-bit** interface, an **8-bit** instruction or data must be broken into **two parts** to be transferred.
- The **higher 4-bit** instruction or data is transferred first via **DB7~DB4** in the SPLC780. Then the **lower 4-bit** is transferred via **DB7~DB4** again. The relationship for the lower 4-bit in the **74VHCT595** and **SPLC780** is:
  - **Q7 = DB7**
  - **Q6 = DB6**
  - **Q5 = DB5**
  - **Q4 = DB4**
- Tables 8.11 and 8.12 (next slides) list the general **LCD instructions** and the actual LCD instructions specially applied in the example project for **4-bit** data transfer format implemented in SPLC780 LCD Controller (**X** = don't care).



### 8.3.5.3.2 CONTROL AND INTERFACE PROGRAMMING FOR SPLC780

Table 8.11 Most popular LCD Instructions for 4-bit data transfer.

Items	Code						Description
Instruction	RS	R/W	DB7	DB6	DB5	DB4	
<b>1 Clear Display</b>	0	0	0	0	0	0	Clear display area & set DDRAM address 0 in the Address Counter (AC).
	0	0	0	0	0	1	
<b>2 Return Home</b>	0	0	0	0	0	0	Return the display to the original position & set DDRAM address to 0.
	0	0	0	0	1	X	
<b>3 Set Function</b>	0	0	0	0	1	DL	Set to 4-bit operation. In this case, operation is handled as 8 bits by initialization, and only this instruction is performed with one write.
<b>4 Set Function</b>	0	0	0	0	1	DL	Set 4-bit operation & selects 2-line display & 5×8 dot font. 4-bit operation starts from this step and resetting is necessary. (Number of display lines and character fonts cannot be changed after this step).
	0	0	N	F	X	X	
<b>5 Display On/Off</b>	0	0	0	0	0	0	Turns on display (D=1) and cursor (C=1). Turn off blinking (B=0). Entire display is in space mode because of initialization.
<b>6 Set Entry Mode</b>	0	0	0	0	0	0	Sets mode to increment the address by one (I/D=1) & to shift the cursor to the right at the time of write to the DD/CGRAM. Display is not shifted (S=0).
	0	0	0	1	I/D	S	
<b>7 Set DDRAM Address</b>	0	0	1	ADD	ADD	ADD	Set DDRAM address (DB7=1). ADD is the address bit.
<b>8 Write Data to DDRAM</b>	1	0	<b>8-BIT WRITE-IN DATA</b>				Write data into DDRAM & display them in the LCD.
	1	0	<b>(ASCII CODE)</b>				
<b>9 Read Data from DDRAM</b>	1	1	<b>8-BIT READ-OUT DATA</b>				Read data from DDRAM.
	1	1	<b>(ASCII CODE)</b>				

### 8.3.5.3.2 CONTROL AND INTERFACE PROGRAMMING FOR SPLC780

Table 8.12 The actual LCD Instructions for 4-bit data transfer in the example project.

Items	Code						Description
Instruction	RS	R/W	DB7	DB6	DB5	DB4	
1 <b>Clear Display</b>	0	0	0	0	0	0	Clear display area & set DDRAM address 0 in the Address Counter (AC).
	0	0	0	0	0	1	
2 <b>Return Home</b>	0	0	0	0	0	0	Return the display to the original position & set DDRAM address to 0.
	0	0	0	0	1	X	
3 <b>Set Function</b>	0	0	0	0	1	0	Set to 4-bit operation. In this case, operation is handled as 8 bits by initialization, and only this instruction is performed with one write.
4 <b>Set Function</b>	0	0	0	0	1	0	Set 4-bit operation & selects 2-line display & 5×8 dot font. 4-bit operation starts from this step and resetting is necessary. (Number of display lines and character fonts cannot be changed after this step).
	0	0	1	0	X	X	
5 <b>Display On/Off</b>	0	0	0	0	0	0	Turns on display (D=1) and cursor (C=1). Turn on blinking (B=1). Entire display is in space mode because of initialization.
6 <b>Set Entry Mode</b>	0	0	0	0	0	0	Sets mode to increment the address by one (I/D=1) & to shift the cursor to the right at the time of write to the DD/CGRAM. Display is not shifted (S=0).
	0	0	0	1	1	0	
7 <b>Set DDRAM Address</b>	0	0	1	ADD	ADD	ADD	Set DDRAM address (DB7=1). ADD is the address bit.
8 <b>Write Data to DDRAM</b>	0	0	<b>8-BIT WRITE-IN DATA</b>				Write data into DDRAM & display them in the LCD.
	1	0	<b>(ASCII CODE)</b>				
9 <b>Read Data from DDRAM</b>	1	1	<b>8-BIT READ-OUT DATA</b>				Read data from DDRAM.
	1	1	<b>(ASCII CODE)</b>				

### 8.3.5.3.2 CONTROL AND INTERFACE PROGRAMMING FOR SPLC780

- One important point to be noted is the operational steps **3** and **4**.
- After the LCD initialization process, the **SPLC780** uses the default data transfer format, 8-bit data format to be set. Therefore in step **3**, even the SPLC780 is configured as **4-bit** mode, the SPLC780 still be considered as **8-bit** mode.
- In order to enable the SPLC780 to know this **4-bit** configuration, the SPLC780 needs to be reconfigured by rewriting **DB7~DB4 = 0010** again in step **4**.
- Then the lower **4-bit** are written to the SPLC780 by performing the second writing operation. This rewriting is very important and necessary. Otherwise the SPLC780 may encounter some abnormal operations later.
- Another point is the **ENABLE** signal **E** on the SPLC780. In the instruction sequence listed above, we did not list the **ENABLE** signal. In fact, this control signal is very important and all operations listed in both tables need this signal to be active or **HIGH** when any of instruction is written and executed.



### 8.3.5.3.3 LCD PROGRAMMING INSTRUCTION STRUCTURE AND SEQUENCE

- The **SSI2** output is connected to the LCD via a serial shift register **74VHCT595** and a LCD Controller **SPLC780**.
- The higher 4 bits (**DB7 ~ DB4**) are **LCD data** and the lower 3 bits are **LCD control signals, RS, E and BL**. The **R/W** signal is connected to the ground to make the LCD controller work in the writing mode.
- Each **SSI2** serial data output is an 8-bit serial data sequence transmitted from the **SSI2 Data Register (SSIDR)** to the **SSI2Tx** pin via **TxFIFO**. Each 8-bit data should contains LCD data (upper 4-bit **DB7 ~ DB7**) and LCD commands (lower 3-bit **RS, E** and **BL**). Figure 8.23 shows an 8-bit serial data configuration that is sent to the **SSI2Tx** pin and transmitted to the **DS** serial input on **74VHCT595**.

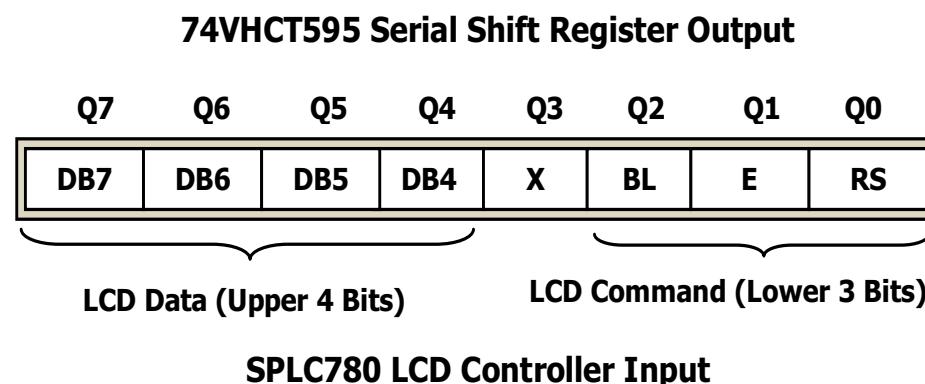


Figure 8.23 8-bit serial data configuration to be transmitted to LCD Controller.

### 8.3.5.3.3 LCD PROGRAMMING INSTRUCTION STRUCTURE AND SEQUENCE

- Refer to Table 8.12. For example, if we want to **clear display** on LCD panel, we need to:
  - First send **DB7-DB4 = 0000** and then
  - **DB7-DB4 = 0001** as **LCD data** to the LCD controller **in two times**.
- In both times, we should also include the LCD command **BL:E:RS = 110** with those LCD data together to send out. The resulted instruction of this 8-bit code is **00000110 (0x06)** for the first 8-bit code and **00010110 (0x16)** for the second 8-bit code.
- In our actual project, we can build some **subroutines** to perform these data **splitting** operations, and therefore we can directly send **00000001 (0x01)** to the subroutine to allow that subroutine to break this 8-bit data into two pieces of **4-bit** data (**0000** and **0001**) and send them out with the corresponding LCD command bits (**BL**, **E**, **RS**) together.



### 8.3.5.3.3 LCD PROGRAMMING INSTRUCTION STRUCTURE AND SEQUENCE

- Generally the LCD programming process includes three sections:
  - **LCD Initialization** (if internal initialization is not as good as desired, the programming initialization built by using the user's program is necessary).
  - **LCD Function Setup and Configuration.**
  - **LCD Displaying Data Output and Display.**
- Now let's have a closer look at the LCD initialization and function setup instruction structure and sequence. First let's handle the LCD initialization process.



### 8.3.5.3.3.1 THE LCD INITIALIZATION PROCESS

- The operational sequence of the LCD manually initialization process is shown in Figure 8.24.

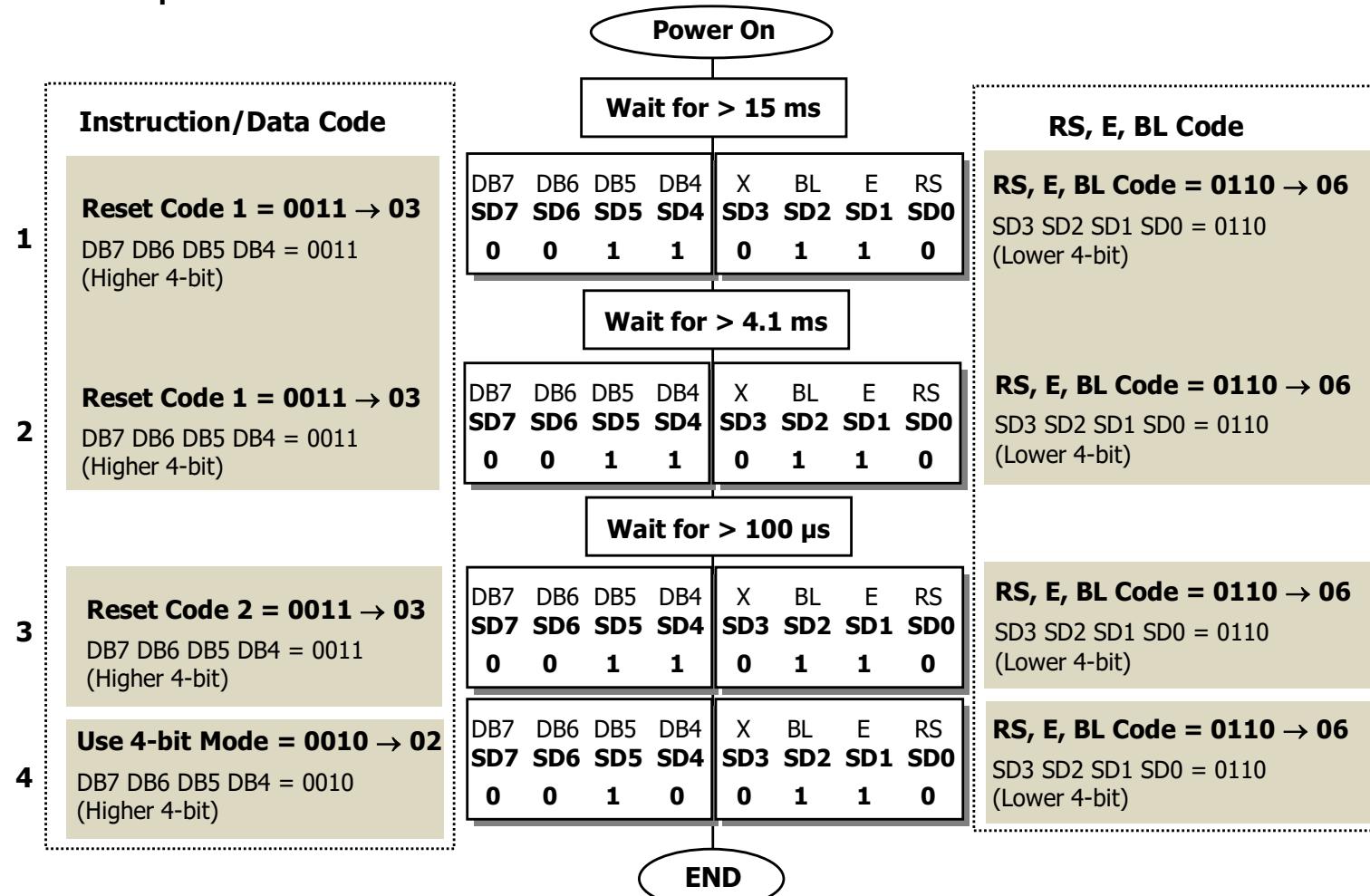


Figure 8.24 The coding process for the LCD initialization operation.

### 8.3.5.3.3.1 THE LCD INITIALIZATION PROCESS

- **Two reset codes** are used for this LCD initialization process.
- **Reset Code 1** is sent to the SPLC780 two times. The combined single 8-bit code is **0x36** (**DB7-DB4 = 0011**, **BL:E:RS = 0110**). Some time delays are needed between these two reset codes to be transferred to the SPLC780.
- Similarly if we combine **Reset Code 2** together to form a single reset code, it is **0x36** (step 3 in Figure 8.24). After these reset, the LCD should be initialized properly and can be configured to perform our desired data display process.
- **Step 4** in Figure 8.24 is to set the data mode for the LCD module (step 4 in Table 8.12). The higher 4-bit is **0010** and the lower 4-bit is **0110**. Combined these two pieces of nibbles together, we can get **0x26**.
- The reason we combine these two codes together to get a single 8-bit code is that we will use this kind of 8-bit code to do the LCD initialization in our program. With the help of some subroutines, we can decompose these combinations into two separate 4-bit codes and transfer them in 8-bit format with the higher 4-bit as **DB7 ~ DB4** and lower 4-bit as **RS**, **E** and **BL** codes.

### 8.3.5.3.3.1 THE LCD INITIALIZATION PROCESS

- One point to be noted when transmitting these **two-piece codes** by using subroutines is that the LCD controller can only accept these instruction/data when it is enabled. In order to make the LCD controller enabled to accept these instructions/data, we need to simulate an **Enable (E)** pulse by using the software codes in our subroutines.
- For example, to send the **Reset Code 1** to the SPLC780, one needs to use the codes shown in Figure 8.25 to send **0x36** with a simulated E pulse.

```
1 LCD_write(0x30, 0);                                // call LCD_Write() function...
2
3 void LCD_write(char data, unsigned char cmd)
4 {
5     data &= 0xF0;                                    // clear lower nibble for data
6     cmd &= 0x0F;                                    // clear upper nibble for control
7     SSI2_Write (data | cmd | BL);                  // RS = 0, R/W = 0, E = 0
8     SSI2_Write (data | cmd | EN | BL);            // RS = 0, R/W = 0, E = 1
9     delay();                                     
10    SSI2_Write (data | BL);                      // RS = 0, R/W = 0, E = 0
11    SSI2_Write (BL);                            // RS = 0, R/W = 0, BL = 1
}
```

A simulated E pulse

Figure 8.25 A piece of codes to create a simulated E pulse.

### 8.3.5.3.3.1 THE LCD INITIALIZATION PROCESS

- The codes in lines 6 ~ 8 are used to send a combined instruction/command code **0x34** (**DB7 ~ DB4 = 0011**, **BL:E:RS = 100**) to the LCD controller.
- In order to allow this code to be accepted by the SPLC780, the bit **E** must be **High**.
- The code in line 7 does this job. The codes in lines 6 and 8 are used to simulate a **Low** before and after the **High pulse** of **E** in line 7.
- The function **SSI2\_Write()** is used to send serial data to the SPLC789 via 74VHCT595.

### 8.3.5.3.3.2 THE LCD FUNCTION SETUP AND CONFIGURATION

- The LCD function setup and configuration process are equivalent to perform operations in steps 1 ~ 6 shown in Table 8.12.
- The operational sequence of the LCD **function setup** and **configuration** process is shown in Figure 8.26 (next slide).
- For example, to transfer the Function Set Code **0x28** at step 1 in Figure 8.26, this 8-bit code **0x28** is broken into two pieces, **0x2** and **0x8**, as shown in coding steps 4 and 5 in Figure 8.25.
- The **higher 4-bit 0x2** is sent first with the **RS**, **E** and **BL** code **0x7** together to form an 8-bit code **0x27**.
- Then the **lower 4-bit 0x8** is transferred with the **RS**, **E** and **BL** code **0x7** to form an 8-bit code **0x87** in the second time.
- These breaking & combining jobs as well as a simulated **E** pulse are performed by the software codes in a subroutine similar to one shown in Figure 8.25.

## 8.3.5.3.3.2 THE LCD FUNCTION SETUP AND CONFIGURATION

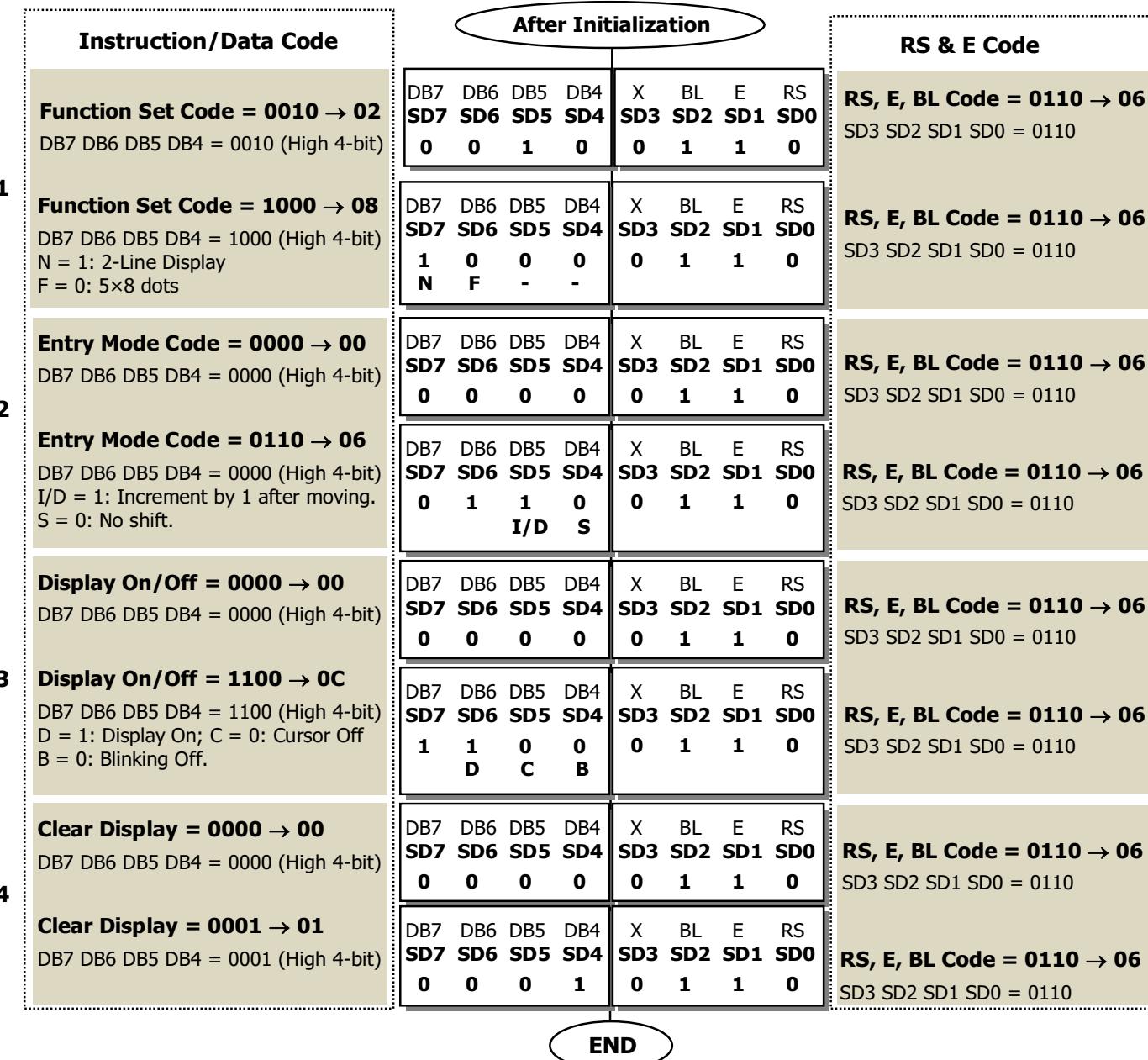


Figure 8.26 Operational sequence of LCD function set and configuration.

### 8.3.5.3.3.2 THE LCD FUNCTION SETUP AND CONFIGURATION

- **Step 2** in Figure 8.26 is used to set the entry mode code, which is **0006** or **ox06**. This 8-bit code is also broken to two pieces of 4-bit codes: **0000** and **0110** and sent in two times with the **BL**, **E** and **RS** signal together to the SPLC780 via the 74VHCT595.
- **Step 3** is to setup the display mode with the code of **00001111 = ox0F**, and **step 4** is to clear the display area with the code of **00000001 = ox01**.
- These 8-bit codes are also broken into two pieces of 4-bit codes and sent out in two times with the **BL**, **E** and **RS** signal together.

### 8.3.5.4 BUILD THE EXAMPLE LCD INTERFACING PROJECT

- The hardware configuration of this project is shown in Figure 8.18.
- In the EduBASE ARM® Trainer, it has been configured to use the **SSI module 2** to serially transmit instructions and data to **LCD Controller SPLC780** via the serial shift register **74VHTC595**. The GPIO pins used in this project include:
  - **PB7: SSI2Tx** - Transmit instructions/data to the peripheral 74VHCT595.
  - **PB4: SSI2Clk** – Provide SSI2 with a clock to conduct the serial data transmission.
  - **PB6: SSI2Rx** – Receive serial data from the peripheral (not used in this example).
  - **PB5: SSI2Fss** – Provide the serial data transmit frame (not used in this example).
  - **PC6: /CS\_LCD** – Provide the **LCD Chip Select** signal. Here it works as a trigger clock to start a serial-to-parallel data conversion to convert 8-bit serial data stored in the serial shift register to 8-bit parallel data to be stored in the 8-bit output registers in the 74VHCT595 register.



### 8.3.5.4 BUILD THE EXAMPLE LCD INTERFACING PROJECT

- In our example, each time after **SSI2** transmits 8 bits data to the shift register via **DS (SSI2Tx – PB7)** pin and clock **SHCP (SSI2Clk – PB4)** pin, a positive-going-edge (**Low-to-High**) signal should be applied on the **STCP (/CS\_LCD – PC6)** pin to transfer 8-bit serial data in the shift register into the 8-bit parallel output buffer or latches.
- The **OE** pin is connected to the ground to enable 8 latched outputs to be directly transferred to the output pins (Figure 8.18).
- We try to use the Direct Register Access (**DRA**) model to build this project. The entire coding process includes the following four sections:
  1. Initialize and configure SSI2 related GPIO Ports (**Ports B** and **C**).
  2. Initialize and configure SSI2 module and related registers (**SSICC**, **SSICPSR**, **SSICR0** and **SSICR1**).
  3. Build four subroutines to break, combine and transmit 8-bit instructions/data to the LCD controller to display desired letters and numbers on the LCD panel.
  4. Build some time delay functions to delay different periods of time to make the LCD data transmissions stable and reliable.

#### 8.3.5.4.1 CREATE A DIRECT REGISTER ACCESS LCD PROJECT DRALCD

- Perform the following operations to create a new project **DRALCD**:
  1. Open the Windows Explorer to create a new folder named **DRALCD** under the **C:\ARM Class Projects\Chapter 8** folder.
  2. Open the Keil® ARM-MDK µVersion5 and go to **Project|New µVersion Project** menu item to create a new µVersion Project. On the opened wizard, browse to our new folder **DRALCD** that is created in step 1 above. Enter **DRALCD** into the **File name** box and click on the **Save** button to create this project.
  3. On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series** and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
  4. Next the Software Components wizard is opened, and you need to setup the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, click on **OK** button since we need these two components to build our project.
- Since this project is a little complex, therefore both a header file and a C file are needed.

### 8.3.5.4.2 CREATE THE HEADER FILE DRALCD.H

- Create a new header file **DRALCD.h** and enter the codes shown in Figure 8.27 into this header file.

```
1 //*****  
2 // DRALCD.h - Header Files for the LCD Project - DRALCD  
3 //*****  
4 #include <stdint.h>  
5 #include <stdbool.h>  
6 #include "TM4C123GH6PM.h"  
7  
8 #define RS      1          // 74VHCT595 Q0 bit for RS (Reg Select)  
9 #define EN      2          // 74VHCT595 Q1 bit for E (Enable LCD)  
10 #define BL     4          // 74VHCT595 Q2 bit for BL (Backlight)  
11  
12 void delay_ms(int time);  
13 void delay_us(int time);  
14 void LCD_cd_Write(char data, unsigned char control);  
15 void LCD_Comd(unsigned char cmd);  
16 void LCD_Data(char data);  
17 void LCD_Init(void);  
18 void SSI2_Write(unsigned char data);
```

Figure 8.27 The codes for the header file DRALCD.h.

### 8.3.5.4.3 CREATE THE C SOURCE FILE DRALCD.C

- Create a new C file **DRALCD.c** and enter the first part codes shown in Figure 8.28 into this C file.

```
1 //*****  
2 // DRALCD.c - Main Application File for LCD Project – The First Part Codes  
3 //*****  
4 #include "DRALCD.H"  
5  
6 int main(void)  
7 {  
8     LCD_Init();                                // initialize LCD controller  
9     LCD_Command(1);                           // clear screen, move cursor to home  
10    LCD_Data('W'); LCD_Data('E'); LCD_Data('L'); LCD_Data('C'); LCD_Data('O');  
11    LCD_Data('M'); LCD_Data('E'); LCD_Data(' '); LCD_Data('T'); LCD_Data('O');  
12    LCD_Data(' '); LCD_Data('J'); LCD_Data('C'); LCD_Data('S'); LCD_Data('U'); ;LCD_Data('!');  
13 }  
14 void LCD_Init(void)                         // initialize SSI2 then initialize LCD controller  
15 {  
16     SYSCTL->RCGCGSS | = 0x04;                // enable clock to SSI2  
17     SYSCTL->RCGCGPIO |= 0x02|0x04;           // enable clock to Port B and Port C  
18     // PB7 & PB4 for SSI2Tx and SSI2Clk  
19     GPIOB->AMSEL &= ~0x90;                  // turn off analog of PB7 & PB4  
20     GPIOB->AFSEL |= 0x90;                   // set PB7 & PB4 for alternate functions  
21     GPIOB->PCTL &= ~0xF00F0000;            // clear functions for PB7 & PB4  
22     GPIOB->PCTL |= 0x20020000;             // PB7 & PB4 for SSI2 function  
23     GPIOB->DEN |= 0x90;                    // PB7 & PB4 as digital function pins  
24     // PC6 for SSI2 slave select and CS_LCD (STCP) clock  
25     GPIOC->AMSEL &= ~0x40;                 // disable PC6 analog function  
26     GPIOC->DIR |= 0x40;                   // set PC6 as output for CS_LCD signal  
27     GPIOC->DEN |= 0x40;                    // set PC6 as digital pins
```

Figure 8.28 The first part source codes for the project DRALCD.

### 8.3.5.4.3 CREATE THE C SOURCE FILE DRALCD.C

```
1 //*****  
2 // DRALCD.c - Main Application File for LCD Project – The Second Part Codes  
3 //*****  
4  
28 SSI2->CR1 = 0; // make SSI2 as master and disable SSI2  
29 SSI2->CC = 0; // use system clock (16 MHz)  
30 SSI2->CPSR = 16; // clock prescaler divide by 16 gets 1 MHz SSI2Clk clock  
31 SSI2->CR0 = 0x0007; // clock rate div by 1, phase/polarity 0/0, freescale, data size 8  
32 SSI2->CR1 = 2; // enable SSI2  
33 delay_ms(20); // LCD controller reset sequence  
34 LCD_cd_Write(0x30, 0); // send reset code 1 two times to SPLC780  
35 delay_ms(5);  
36 LCD_cd_Write(0x30, 0);  
37 delay_ms(1);  
38 LCD_cd_Write(0x30, 0); // send reset code 2 to SPLC780  
39 delay_ms(1);  
40 LCD_cd_Write(0x20, 0); // use 4-bit data mode  
41 delay_ms(1);  
42 LCD_Comd(0x28); // set 4-bit data, 2-line, 5x7 font  
43 LCD_Comd(0x06); // move cursor right  
44 LCD_Comd(0x0C); // turn on display, cursor off - no blinking  
45 LCD_Comd(0x01); // clear screen, move cursor to home  
46 }  
47 void SSI2_Write(unsigned char data)  
48 {  
49     GPIOC->DATA &= ~0x40; // clear STCP (CS_LCD) in 74VHCT595 to Low (PC6)  
50     SSI2->DR = data; // write serial data into 74VHCT595  
51     while (SSI2->SR & 0x10); // wait for 74VHCT595 serial data shift done  
52     GPIOC->DATA |= 0x40; // set CS_LCD (STCP) to High to simulate a positive-going-edge  
53 }
```

Figure 8.28 The second part source codes for the project DRALCD.

### 8.3.5.4.3 CREATE THE C SOURCE FILE DRALCD.C

```
1 //*****  
2 // DRALCD.c - Main Application File for LCD Project – The Third Part Codes  
3 //*****  
57 void LCD_cd_Write(char data, unsigned char control)  
58 {  
59     data &= 0xF0;                                // clear lower nibble for data  
60     control &= 0x0F;                             // clear upper nibble for control  
61     SSI2_Write (data | control | BL);           // RS = 0, R/W = 0  
62     SSI2_Write (data | control | EN | BL);       // pulse E  
63     delay_ms(0);  
64     SSI2_Write (data | BL);  
65     SSI2_Write (BL);  
66 }  
67 void LCD_Comd(unsigned char cmd)  
68 {  
69     LCD_cd_Write(cmd & 0xF0, 0);                // upper nibble first  
70     LCD_cd_Write(cmd << 4, 0);                  // then lower nibble  
71     if (cmd < 4)                                // command 1 and 2 needs up to 1.64ms  
72         delay_ms(2);  
73     else  
74         delay_ms(1);                           // all others 40 µs  
75 }
```

Figure 8.28 The third part source codes for the project DRALCD.

### 8.3.5.4.3 CREATE THE C SOURCE FILE DRALCD.C

```
1 //*****  
2 // DRALCD.c - Main Application File for LCD Project – The Fourth Part Codes  
3 //*****  
4  
5 void LCD_Data(char data)  
6 {  
7     LCD_cd_Write(data & 0xF0, RS);           // upper nibble first  
8     LCD_cd_Write(data << 4, RS);           // then lower nibble  
9     delay_us(40);  
10 }  
11  
12 void delay_ms(int time)                  // delay n milliseconds (16 MHz CPU clock)  
13 {  
14     int i, j;  
15     for(i = 0 ; i < time; i++)  
16         for(j = 0; j < 3180; j++) {}          // do nothing for 1 ms  
17 }  
18  
19 void delay_us(int time)                  // delay n microseconds (16 MHz CPU clock)  
20 {  
21     int i, j;  
22     for(i = 0 ; i < time; i++)  
23         for(j = 0; j < 3; j++) {}            // do nothing for 1 µs  
24 }
```

Figure 8.28 The fourth part source codes for the project DRALCD.

#### 8.3.5.4.4 SETUP ENVIRONMENT TO BUILD AND RUN THE PROJECT

- Perform the following operations to setup the environment for this project:
  - Select the debug adapter **Stellaris ICDI** in the **Debug** tab under the **Project|Options for Target ‘Target 1’** menu item.
- Now build the project by going to **Project|Build target** menu item. Then go to **Flash|Download** item to download the image file of the project into the flash memory. Then go to **Debug|Start/Stop Debug Session** and **Debug|Run** item to run the project.
- After the project running, you can find that the letter sequence: **WELCOME TO JCSU!** is displayed in the LCD panel.
- Can we display something on the second line in this LCD with desired starting position? Yes, but you need to refer to **step 7** in Table 8.12 and Figure 8.21 to figure out the coding process.
- You can use the subroutine **LCD\_Command(oxCo)** to select the first position on the second line in the LCD. How about other positions? Think about it.

#### 8.3.5.4.4 SETUP ENVIRONMENT TO BUILD AND RUN THE PROJECT

- Figure 8.30 shows the relationship between the LCD instruction and the DDRAM address.

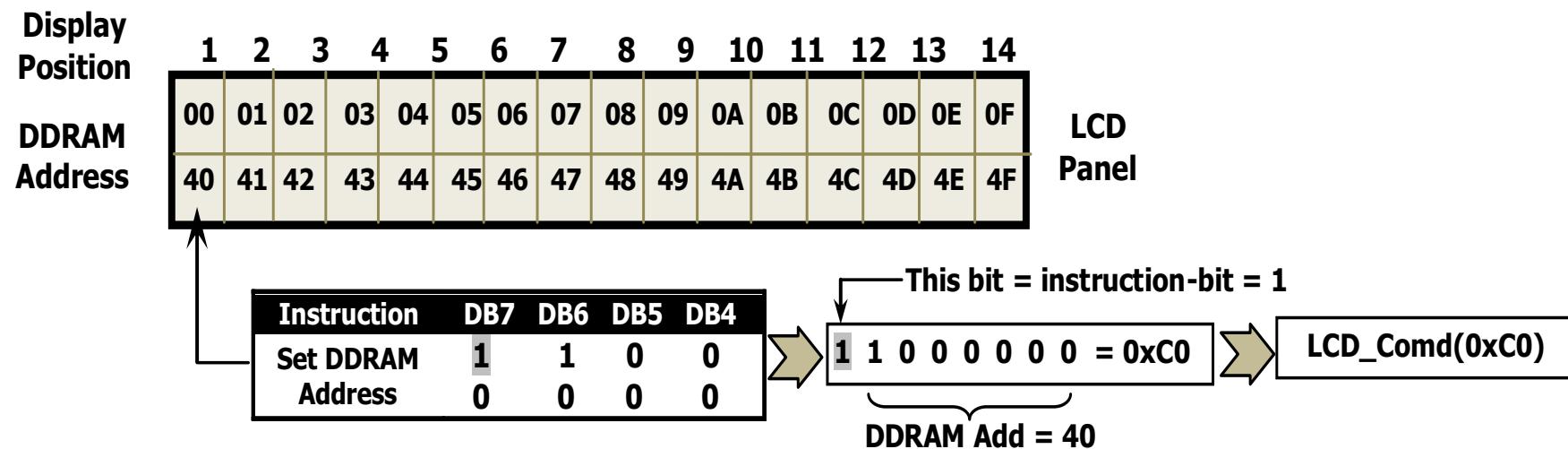


Figure 8.30 The relationship between the LCD instruction and the DDRAM address.

### 8.3.6 BUILD ON-BOARD 7-SEGMENT LED INTERFACE PROGRAMMING PROJECT

- In this section, we illustrate how to use the **SSI Module 2 (SSI2)** to interface four on-board 7-segment LEDs installed in the EduBASE ARM® Trainer.
- First let's have a closer look at the hardware configuration and connection for these LEDs via **SSI2**.

### 8.3.6.1 STRUCTURE OF 7-SEGMENT LEDs

- Figure 8.31a shows a one-digit 7-segment common cathode LED and its structure.

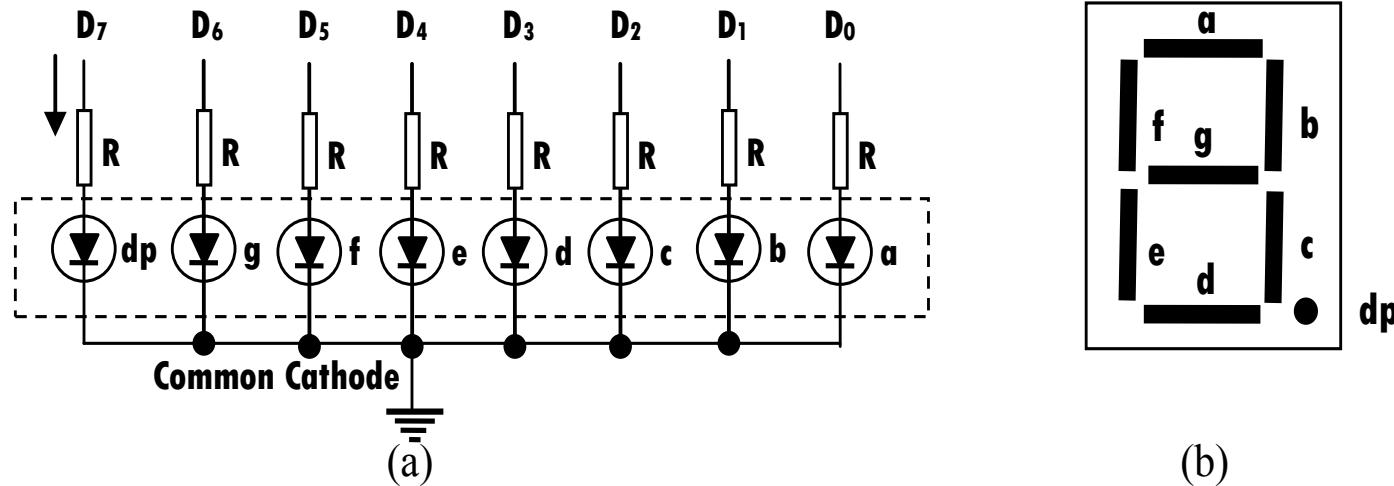


Figure 8.31 The structure of common cathode LED.

- To make any segment, such as **a**, **b**, **c**, etc to light or **ON**, just apply logic **HIGH** to the associated input bit  $D_i$  since the cathode terminals of all of these 7-segment LEDs are connected together to the ground.
- In the EduBASE ARM® Trainer, these 8 input bits ( $D_0 \sim D_7$ ) are connected to 8 latched output bits ( $Q_0 \sim Q_7$ ) on the **74VHCT595**.
- Figure 8.31b shows the layout of a single bit on 7-segment LED.

### 8.3.6.2 SSI MODULE INTERFACE FOR 7-SEGMENT LED IN EDUBASE ARM® TRAINER

- Four 7-segment LEDs are connected to the **SSI2** module via two cascaded serial shift registers **74VHCT595**.
- A functional block diagram of this interface circuit is shown in Figure 8.32.

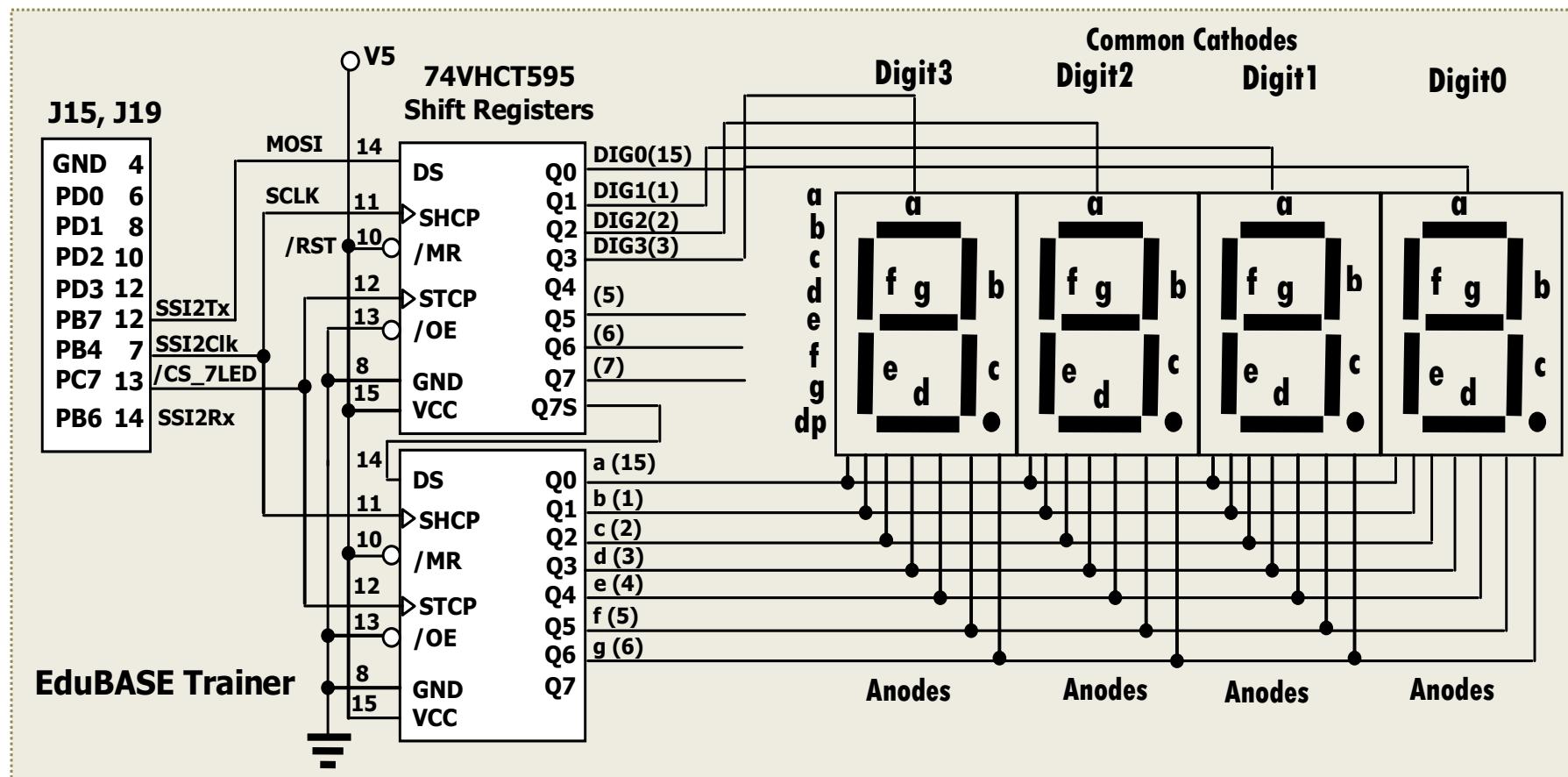


Figure 8.32 The hardware configuration of 74VHCT 595 and 7-segment LEDs.

### 8.3.6.2 SSI MODULE INTERFACE FOR 7-SEGMENT LED IN EDUBASE ARM® TRAINER

- It can be found from Figure 8.32 that **two** serial shift registers **74VHCT595** are **cascaded** or serial connected together to form two 8-bit parallel outputs to perform the LED digit-selection and 7-segment selection functions.
- The serial input and controls to the 74VHCT595 are still from **SSI module 2** control pins.
- The GPIO Port C pin 7 (**PC7**) now is used to provide the slave selection and triggering signal to convert 8-serial data into 8-parallel data in the 74VHCT595 when a positive-going-edge signal is provided by this pin.
- When working this mode, the first serial 8-bit data to be transferred by **SSI2** should be 7-segment control signal used to select the required segments to display desired number on all four LEDs. The second serial 8-bit data (only the lower 4-bit are used) should be used to select the desired LED digit.

### 8.3.6.2 SSI MODULE INTERFACE FOR 7-SEGMENT LED IN EDUBASE ARM® TRAINER

- To display desired numbers or letters on these LEDs, a **code conversion** job is necessary to translate the binary or hexadecimal codes output from the 74VHCT595 to the 7-segment codes to be displayed on those LEDs.
- Table 8.13 (next slide) shows the relationship between these code translations. In the real program, a subroutine can be developed and used to perform this coding conversion before the number or letter can be displayed on these LEDs.
- In addition to converting the data sent to the 74VHCT595 from binary or hexadecimal codes to 7-segment codes, a **LED digit selection** code should also be sent to the 74VHCT595 to select the desired **LED digit**.
- Table 8.14 shows a relationship between the codes to be sent to the 74VHCT595 and the selected LED digit.

Table 8.14 The codes sent to the second 74VHCT595 and the selected LED digits

LED DIGITS <i>Q<sub>n</sub> CODE</i>	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0	HEX CODE
DIGITO	X	X	X	X	1	1	1	0	\$FE
DIGIT1	X	X	X	X	1	1	0	1	\$FD
DIGIT2	X	X	X	X	1	0	1	1	\$FB
DIGIT3	X	X	X	X	0	1	1	1	\$F7

### 8.3.6.2 SSI MODULE INTERFACE FOR 7-SEGMENT LED IN EDUBASE ARM® TRAINER

Table 8.13 The relation between the 7-segment data code and normal code.

Number/ Letter	dp Q7	g Q6	f Q5	e Q4	d Q3	c Q2	b Q1	a Q0	7-Segment Code
0	0	0	1	1	1	1	1	1	3F
1	0	0	0	0	0	1	1	0	06
2	0	1	0	1	1	0	1	1	5B
3	0	1	0	0	1	1	1	1	4F
4	0	1	1	0	0	1	1	0	66
5	0	1	1	0	1	1	0	1	6D
6	0	1	1	1	1	1	0	1	7D
7	0	0	0	0	0	1	1	1	07
8	0	1	1	1	1	1	1	1	7F
9	0	1	1	0	1	1	1	1	6F
A	0	1	1	1	0	1	1	1	77
b	0	1	1	1	1	1	0	0	7C
C	0	0	1	1	1	0	0	1	39
d	0	1	0	1	1	1	1	0	5E
E	0	1	1	1	1	0	0	1	79
F	0	1	1	1	0	0	0	1	71
H	0	1	1	1	0	1	1	0	76
h	0	1	1	1	0	1	0	0	74
J	0	0	0	1	1	1	1	0	1E
L	0	0	1	1	1	0	0	0	38
S	0	1	1	0	1	1	0	1	6D
P	0	1	1	1	0	0	1	1	73
U	0	0	1	1	1	1	1	0	3E

### 8.3.6.2 SSI MODULE INTERFACE FOR 7-SEGMENT LED IN EDUBASE ARM® TRAINER

- For example, to display a number **0** in the **first digit** of 7-segment LED, **Digito**, the following data should be created and sent to the 74VHCT595:
  - A **0x3F** or **00111111B** is sent to 74VHCT595 to make **Q7 ~ Q0 = 00111111** on the second 74VHCT595. This will make segments (anodes) **a, b, c, d, e** and **f ON**, and segments **g** and **dp OFF**.
  - A **0xFE** or **11111110B** is sent to the first 74VHCT595 to make **Q3 ~ Q0 = 1110**. This will make the cathode of the **Digito** to **Low** to select this digit (the higher 4-bit can be any values). The key is to send a **Low** or **0** to the cathode of the selected LED digit to enable it to be **ON** to display it.
- Now let's build an example project to access and interface these 7-segment LEDs.



### 8.3.6.3 BUILD THE EXAMPLE LED INTERFACING PROJECT

- The hardware configuration of this project is shown in Figure 8.32.
- In the EduBASE ARM® Trainer, it has been configured to use the **SSI module 2** to serially transmit instructions and data to LEDs via two cascaded serial shift registers 74VHTC595. The GPIO pins used in this project include:
  - **PB7: SSI2Tx** - Transmit instructions/data to the peripheral 74VHCT595.
  - **PB4: SSI2Clk** – Provide SSI2 with a clock to conduct the serial data transmission.
  - **PB6: SSI2Rx** – Receive serial data from the peripheral (not used in this example).
  - **PB5: SSI2Fss** – Provide the serial data transmit frame (not used in this example).
  - **PC7: /CS\_7LED** – Provide the LED **Chip Select** signal. Here it works as a trigger clock to start a serial-to-parallel data conversion to convert 8-bit serial data stored in the serial shift register to 8-bit parallel data to be stored in the 8-bit output registers in the 74VHCT595 register.



### 8.3.6.3 BUILD THE EXAMPLE LED INTERFACING PROJECT

- In our example, each time after **SSI2** transmits 8 bits data to the shift register via **DS (SSI2Tx – PB7)** pin and clock **SHCP (SSI2Clk – PB4)** pin, a positive-going-edge (**Low-to-High**) signal should be applied on the **STCP (CS\_7LED – PC7)** pin to transfer 8-bit serial data in the shift register into the 8-bit parallel output buffer or latches.
- The **OE** pin is connected to **ground** to enable 8 latched outputs to be directly transferred to the output pins (Figure 8.32).
- We like to use the Direct Register Access (**DRA**) model to build this project. The entire coding process includes the following four sections:
  1. Initialize and configure **SSI2** related GPIO Ports (Ports **B** and **C**).
  2. Initialize and configure **SSI2** module and related registers (**SSICC**, **SSICPSR**, **SSICRo** and **SSICR1**).
  3. Build one subroutine to break, combine and transmit 8-bit instructions/data to the LED to display desired letters and numbers on the selected LEDs.
  4. Build some time delay functions to delay different periods of time to make the LED data transmissions stable and reliable.

### 8.3.6.3.1 CREATE A DIRECT REGISTER ACCESS LED PROJECT DRALED

- Perform the following operations to create a new project DRALED:
  1. Open the Windows Explorer to create a new folder **DRALED** under the **C:\ARM Class Projects\Chapter 8** folder.
  2. Open the Keil® ARM-MDK µVersion5 and go to **Project|New µVersion Project** menu item to create a new µVersion Project. On the opened wizard, browse to our new folder **DRALED** that is created in step 1 above. Enter **DRALED** into the **File name** box and click on the **Save** button to create this project.
  3. On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series** and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
  4. Next the Software Components wizard is opened, and you need to setup the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, and click on the **OK** button since we need these two components to build our project.



### 8.3.6.3.2 CREATE THE C SOURCE FILE DRALED.C

- Create a C file **DRALED.c** and enter codes shown in Figure 8.33 into this file.

```
1 //*****  
2 // DRALED.c - Main Application File for the LED Project (First Part)  
3 //*****  
4 #include <stdint.h>  
5 #include <stdbool.h>  
6 #include "TM4C123GH6PM.h"  
7  
8 void delay_ms(int time);  
9 void LED_Init(void);  
10 void SSI2_Write(unsigned char data);  
11  
12 int main(void)  
13 {  
14     LED_Init();                                // initialize SSI2 that connects to the shift registers  
15     while(1)  
16     {  
17         SSI2_Write(0x5B);                      // write num 2 to the seven segments  
18         SSI2_Write(0xF7);                      // select digit 3  
19         delay_ms(4);  
20  
21         SSI2_Write(0x3F);                      // write num 0 to the seven segments  
22         SSI2_Write(0xFB);                      // select digit 2  
23         delay_ms(4);  
24  
25         SSI2_Write(0x06);                      // write num 1 to the seven segments  
26         SSI2_Write(0xFD);                      // select digit 1  
27         delay_ms(4);  
28         SSI2_Write(0x6D);                      // write num 5 to the seven segments  
29         SSI2_Write(0xFE);                      // select digit 0  
30         delay_ms(4);  
31     }  
32 }
```

Figure 8.33 The C source file for the DRALED project – First Part.

### 8.3.6.3.2 CREATE THE C SOURCE FILE DRALED.C

```
1 //*****
2 // DRALED.c - Main Application File for the LED Project (Second Part)
3 //*****
4
5 void LED_Init(void) // enable SSI2 and associated GPIO pins
6 {
7     SYSCTL->RCGCGPIO |= 0x02| 0x04; // enable clock to GPIOB & GPIOC
8     SYSCTL->RCGCSSI |= 0x04; // enable clock to SSI2
9
10    GPIOB->AFSEL |= 0x90; // PB7 & PB4 for alternate function
11    GPIOB->PCTL &= ~0xF00F0000; // clear functions for PB7 & PB4
12    GPIOB->PCTL |= 0x20020000; // PB7 & PB4 for SSI2 function - PB7 (SSI2Tx) & PB4 (SSI2Clk)
13    GPIOB->DEN |= 0x90; // PB7 & PB4 as digital pins
14    GPIOC->DIR |= 0x80; // set PC7 as output pin
15    GPIOC->DEN |= 0x80; // set PC7 as digital pin
16
17    SSI2->CR1 = 0; // disable SSI2 during configuration
18    SSI2->CC = 0; // use system clock
19    SSI2->CPSR = 16; // clock prescaler divide by 16 gets 1 MHz clock
20    SSI2->CR0 = 0x0007; // clock rate div by 1, phase/polarity 0/0, mode freescale, data size 8
21    SSI2->CR1 = 2; // enable SSI2 as master
22
23 }
24
25 void SSI2_Write(unsigned char data)
26 {
27     GPIOC->DATA &= ~0x80; // select the slave and clear STCP to Low
28     SSI2->DR = data; // output the serial data
29     while (SSI2->SR & 0x10) {} // wait for transmit done by checking the BSY bit in SSISR
30     GPIOC->DATA |= 0x80; // set STCP to High to trigger serial-to-parallel conversion
31
32 }
33
34 void delay_ms(int time) // delay n milliseconds (16 MHz CPU clock)
35 {
36     int i, j;
37     for(i = 0 ; i < time; i++)
38         for(j = 0; j < 3180; j++)
39 }
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57 }
```

Figure 8.33 The C source file for the DRALED project – Second Part.

### 8.3.6.3.3 SETUP ENVIRONMENT TO BUILD AND RUN THE PROJECT

- Perform the following operations to setup the environment for this project:
  - Select the debug adapter **Stellaris ICDI** in the **Debug** tab under the **Project|Options for Target ‘Target 1’** menu item.
- Now build the project by going to **Project|Build target** menu item. Then go to **Flash|Download** item to download the image file of the project into the flash memory. Then go to **Debug|Start/Stop Debug Session** and **Debug|Run** item to run the project.
- After the project running, you can find that four numbers: **2015** are displayed in four LEDs.
- We leave a similar LED project for the readers. The function of that project is to repeatedly display each segment on each digit.



### 8.3.7 BUILD DIGITAL TO ANALOG CONVERTER PROGRAMMING PROJECT

- In this section, we illustrate how to use the **SSI Module 2 (SSI2)** to interface to a Digital-to-Analog Converter (**DAC-MCP4922**) installed in the EduBASE ARM® Trainer.
- The **DAC-MCP4922** is directly connected to the **SSI2** module without using any serial shift register. Because the DAC-MCP4922 provides a serial data input. A functional block diagram of this interface circuit is shown in Figure 8.34.

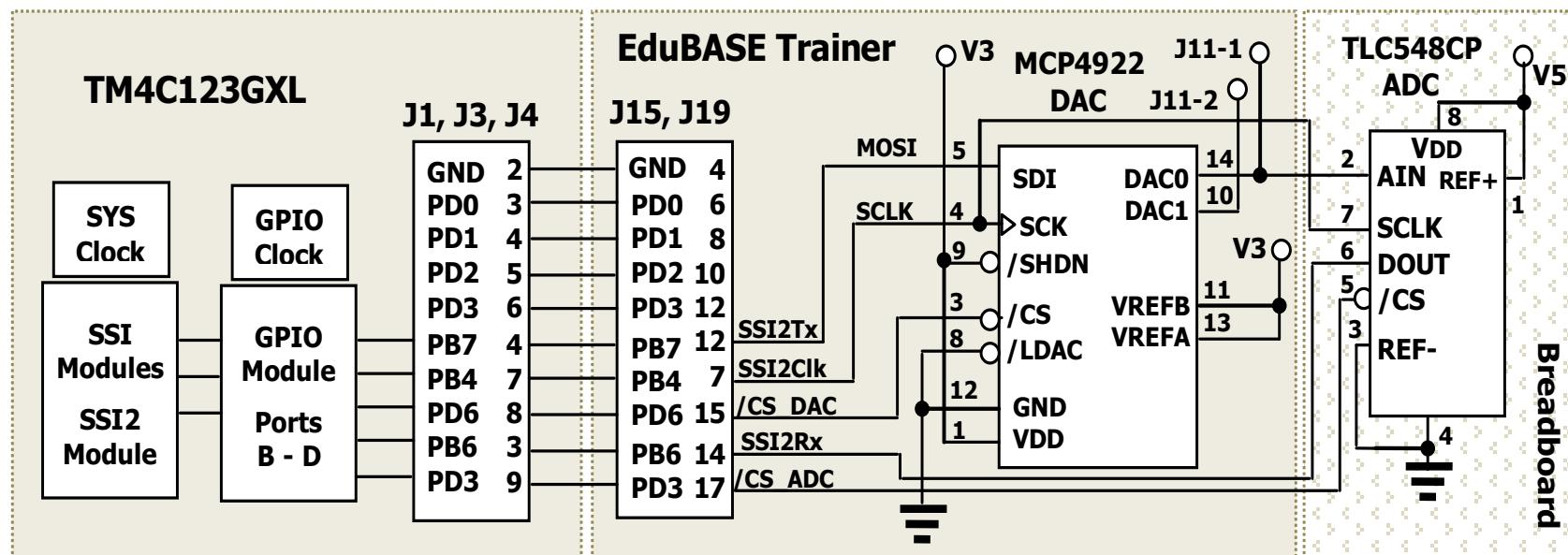


Figure 8.34 Hardware configuration of the DAC-MCP4922 and SSI2 module.

### 8.3.7.1 SSI MODULE INTERFACE FOR DAC-MCP4922 IN EDUBASE ARM® TRAINER

- It can be found from Figure 8.34 that two **SSI2** control signals, **SSI2Tx** and **SSI2Clk**, are connected to the **DAC-MCP4922** to control the DAC conversions and serial data transmission.
- Optionally a serial **ADC-TLC548** can be connected to the output of the **DAC0** to receive and check the DAC transferring results. An oscilloscope can also be connected to **DAC0 (J11-1)** and **DAC1 (J11-2)** pins to monitor and check the converted analog waveform outputs.
- Now let's have a closer look at these two peripherals, the Digital-to-Analog Converter (**DAC MCP4922**) and the Analog-to-Digital Converter (**ADC-TLC548**).



### 8.3.7.2 THE OPERATIONS AND PROGRAMMING FOR MCP4922 DAC

- The **MCP4922** is a dual **12-bit** buffered voltage output **Digital-to-Analog Converter (DAC)**. The devices operate from a single 2.7V to 5.5V supply with SPI compatible Interface. This **DAC** is a programmable device and needs to be programmed to perform the desired DAC operations.
- A block diagram of the **MCP4922** is shown in Figure 8.35 (next slide).
- The operational principle and sequence of this DAC are:
  - The **SDI** is the serial data input pin to accept serial digital data as input.
  - The **SCK** terminal is used to accept an external clock and the DAC performs its conversions based on this clock as a timing base.
  - The **/CS** is a **Chip-Select** signal used to start the DAC operations. Both DAC modules use the same clock with the same starting signal. The active level is **Low**.
  - The **/LDAC** signal is used to enable both buffered conversion results to be updated in two DAC outputs. The active level of this signal is **Low**.
  - The dual DAC can be shut-down for their outputs if the **/SHDN** signal is **Low**.
  - The dual DAC outputs can be obtained from **V<sub>OUTA</sub>** and **V<sub>OUTB</sub>** pins in serial format. Both outputs can be ranged up to **V<sub>REF</sub>** ( $\times 1 = 2.048V$ ) or doubled **V<sub>REF</sub>** ( $\times 2 = 4.096V$ ).

### 8.3.7.2 THE OPERATIONS AND PROGRAMMING FOR MCP4922 DAC

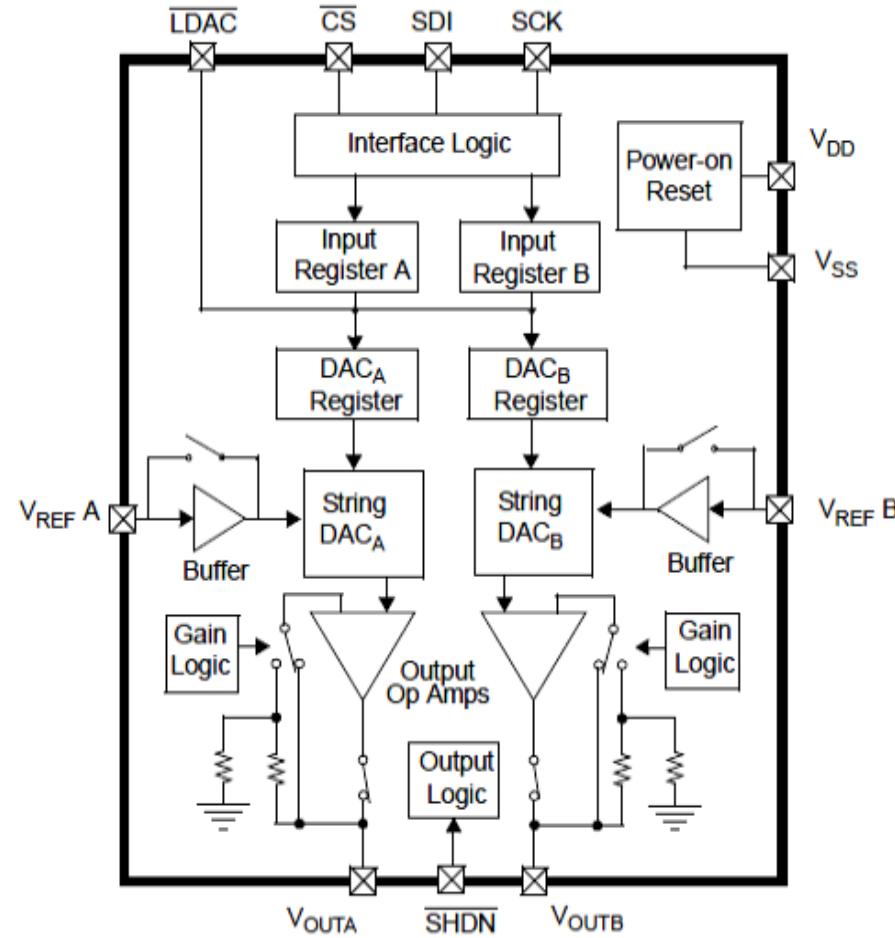


Figure 8.35 The functional block diagram of the DAC - MCP4922.

### 8.3.7.2 THE OPERATIONS AND PROGRAMMING FOR MCP4922 DAC

- In the EduBASE ARM® Trainer, the **/LDAC** is connected to **ground** and this is equivalent to setting the **/LDAC** pin to **Low**. This connection enables both outputs to be updated as a rising edge appeared on the **/CS** signal (Figure 8.34).
- The **/SHDN** pin is connected to the power supply **V3**, and this is equivalent to setting this pin to **High**. This configuration enables both DAC outputs to be available at any time.
- Both reference pins, **V<sub>REFA</sub>** and **V<sub>REFB</sub>**, are tied to the power supply V3. This connection makes both DAC use the same reference voltage.
- To configure the **MCP4922** to perform any DAC operation, the DAC needs first to be programmed. Each programming instruction is **16-bit** and is composed of commands part (**4 MSB bits**) and data part (**following 12-bit**).

### 8.3.7.2 THE OPERATIONS AND PROGRAMMING FOR MCP4922 DAC

- The **write** command is initiated by driving the **CS** pin to Low, followed by clocking the **four MSB Configuration bits** and the **12 data bits** into the **SDI** pin on the rising edge of **SCK**.
- The **CS** pin is then raised to **High** to cause the data to be latched into the selected DAC input registers. The MCP4922 utilizes a double-buffered latch to allow both **DACA** and **DACB** outputs to be synchronized with the **LDAC** pin if desired.
- Upon the **LDAC** pin achieving a **Low** state, the values held in the DAC input registers are transferred into the DAC output registers. The outputs will transition to the value and held in the **DACA** (DAC0) or **DACB** (DAC1) register.
- All writes to the MCP4922 are **16-bit instructions**. Any clocks past the 16th clock will be ignored.



### 8.3.7.2 THE OPERATIONS AND PROGRAMMING FOR MCP4922 DAC

- The Most Significant **4 bits** are **Configuration bits**. The remaining **12 bits** are data bits. No data can be transferred into the device with **CS** high.
- Figure 8.36 shows an example of the command-data structure to be written into the MCP4922 DAC.

A/B	BUF	GA	SHDN	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
bit 15								bit 0							

Figure 8.36 An example of the command-data structure for the MCP4922 DAC.

- The **4 MSBs** are configuration or command bits. The values for these configuration bits and related functions are shown in Table 8.15 (next slide).
- For example, if we want to configure the MCP4922 to transfer a 12-bit serial data with the following configurations:
  - Select the **DACA (DAC0)** channel as the converting channel.
  - Enable the buffered input and output.
  - Select the output gain as **1**.
  - Do not shut down the DAC channel to enable it active.

### 8.3.7.2 THE OPERATIONS AND PROGRAMMING FOR MCP4922 DAC

- The configuration bits should be: **0x7xxx = 0111xxxxxxxxxxxxxx**.
- The **lower 12 data bits** are x, which means that they do not matter and the real values depend on the application data.

Table 8.15 Bit value and its function for MCP4922 DAC.

Bit	Name	Function
15	<b>A/B</b>	DACA or DACB Selection Bit <b>0:</b> Write to DACA (DAC0) <b>1:</b> Write to DACB (DAC1)
14	<b>BUF</b>	VREF Input Buffer Control Bit <b>0:</b> Un-Buffered <b>1:</b> Buffered
13	<b>GA</b>	Output Gain Selection Bit <b>0:</b> 2x ( $V_{OUT} = 2 * V_{REF} * D/4096$ ) <b>1:</b> 1x ( $V_{OUT} = V_{REF} * D/4096$ )
12	<b>SHDN</b>	Output Shutdown Control Bit <b>0:</b> Shutdown the selected DAC channel. Analog output is not available at the channel that was shut down. VOUT pin is connected to 500 k <b>1:</b> Active mode operation. VOUT is available
11:0	<b>D11:D0</b>	DAC Input Data bits

### 8.3.7.3 THE ANALOG TO DIGITAL CONVERTER TLC-548

- Figure 8.37 shows the functional block diagram of the **ADC- TLC548**.

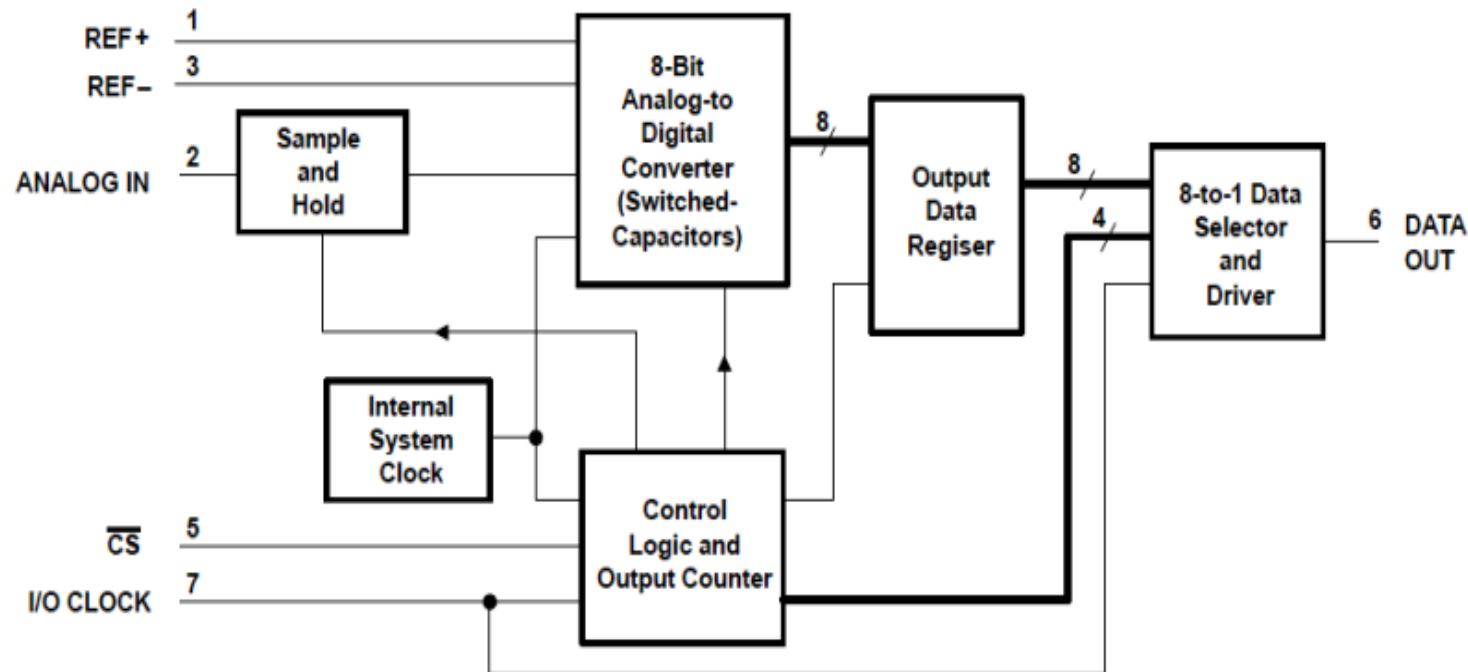


Figure 8.37 Functional block diagram of the ADC- TLC548.

### 8.3.7.3 THE ANALOG TO DIGITAL CONVERTER TLC-548

- The **TLC-548** is an **8-bit serial-in-serial-out** Analog-to-Digital Converter (**ADC**) with 3-wire serial interface, such as **SPI** and **SSI** interfaces.
- The conversion results of the **DAC** can be feed into the TLC-548 serial ADC input via **pin2 (AIN)**. The **CS\_ADC (PD3)** provides a Chip-Select (/CS) signal to start the ADC conversion and the ADC conversion process is controlled by the **SSI2Clk** clock in a timing sequence.
- The ADC conversion results can be received by the **SSI2Rx pin (PB6)** since the ADC output (**DOUT**) is connected to the **SSI2Rx** pin.
- The reference pin (**REF+**) in the TLC-548 is connected to the power supply and **REF-** is connected to ground. With this configuration, the ADC used the internal reference to allow the output up to be **4.096V**.
- The **serial 8-bit output** can be obtained from **DOUT** pin (**pin 6**) via the TLC548, which is controlled by the input clock sequence **SCLK**. The data can be shifted out bit by bit as each falling edge of the **SCLK** is coming.

### 8.3.7.4 BUILD THE EXAMPLE DAC INTERFACING PROJECT

- The hardware configuration of this project is shown in Figure 8.34.
- The GPIO pins used in this project include:
  - **PB7: SSI2Tx** - Transmit digital data to the peripheral DAC – MCP4922.
  - **PB4: SSI2Clk** – Provide SSI2 with a clock to conduct the serial data transmission.
  - **PB6: SSI2Rx** – Receive serial data from the peripheral ADC- TLC548.
  - **PB5: SSI2Fss** – Provide the serial data transmit frame (not used in this example).
  - **PD6: /CS\_DAC** – Provide the DAC Chip Select signal. Here it works as a trigger signal to start the Digital-to-Analog conversion to convert 12-bit serial digital data to 12-bit serial analog data to be sent to the serial input of the Analog-to-Digital Converter TLC548.
  - **PD3: /CS\_ADC** – Provide the ADC Chip Select signal. Here it works as a trigger signal to start the Analog-to-Digital conversion to convert 12-bit serial analog data back to 8-bit serial digital data to be received by the SSI2Rx pin.

### 8.3.7.4 BUILD THE EXAMPLE DAC INTERFACING PROJECT

- In our example application, each time after the **SSI2** transmits 12 bits digital data to the MCP4922 via **SSI2Tx (PB7)** pin and clock **SSI2Clk (PB4)** pin, a Low-level signal should be applied on the **/CS\_DAC (PD6)** pin to start a DAC operation.
- The **DAC conversion results** can be obtained serially from the first conversion channel **V<sub>OUTA</sub>** or **DACo** as each rising edge of the **SSI2Clk** coming (Figure 8.34).
- While this serial analog output can be optionally sent to the **serial input** of the **ADC- TLC548** to perform an A-to-D conversion to convert it back to the digital data.
- Then the digital data can be received by the **SSI2Rx** pin to enable us to collect and then confirm the correctness of these transmission and conversions. A point to be noted is that before this ADC can start, a Low-level signal must be applied on the **/CS\_ADC (PD3)** to enable the TLC548 to start its conversions.
- We try to use the Direct Register Access (DRA) model to build this project.

### 8.3.7.4 BUILD THE EXAMPLE DAC INTERFACING PROJECT

- We try to use the Direct Register Access (**DRA**) model to build this project. The entire coding process includes the following four sections:
  - Initialize and configure SSI2 related GPIO Ports (**Ports B** and **D**).
  - Initialize and configure SSI2 module and related registers (**SSICC**, **SSICPSR**, **SSICR0** and **SSICR1**).
  - Build a subroutine to perform data transmission from the **SSI module 2** to the **DAC-MCP4922** via **SSI2Tx**.
  - Build a subroutine to perform data receiving from the **ADC- TLC548** to the **SSI module 2** via **SSI2Rx**.
- Now let's first create a new DAC project **DRADAC**.



### 8.3.7.4.1 CREATE A DIRECT REGISTER ACCESS DAC PROJECT DRADAC

- Perform the following operations to create a new project DRADAC:
  1. Open the Windows Explorer to create a new folder **DRADAC** under the **C:\ARM Class Projects\Chapter 8** folder.
  2. Open the Keil® ARM-MDK µVersion5 and go to **Project|New µVersion Project** menu item to create a new µVersion Project. On the opened wizard, browse to our new folder **DRADAC** that is created in step 1 above. Enter **DRADAC** into the **File name** box and click on the **Save** button to create this project.
  3. On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series** and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
  4. Next the Software Components wizard is opened, and you need to setup the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, and click on the **OK** button since we need these two components to build our project.



### 8.3.7.4.2 CREATE THE HEADER FILE DRADAC.H

- Create a new header file named **DRADAC.h** and enter the codes shown in Figure 8.38 into this file.
- Since the codes are easy and straightforward, no explanation is needed.

```
1 //*****  
2 // DRADAC.h - Header File for DAC-MCP4922 (SSI2)  
3 //*****  
4 #include <stdint.h>  
5 #include <stdbool.h>  
6 #include "TM4C123GH6PM.h"  
7 #include <math.h>  
8 void DAC_Init(void);  
9 void DAC_Write(int value, int channel);  
10 void ADC_Read(void);
```

Figure 8.38 The header file for the project DRADAC.

### 8.3.7.4.3 CREATE THE C SOURCE FILE DRADAC.C

- Create a C file **DRADAC.c** & enter codes shown in Figure 8.39 into this C file.

```
1 //*****  
2 // DRADAC.c - Main Application File for DAC-MCP4922 (SSI2) – Part I  
3 //*****  
4 #include "DRADAC.h"  
5  
6 int main(void)  
7 {  
8     int idata = 0; float fdata;  
9  
10    DAC_Init();  
11    while(1)  
12    {  
13        // create a sawtooth waveform  
14        DAC_Write(idata++, 0);  
15        // create a sine waveform  
16        fdata = sinf(0.00314159 * idata) * 0x07FF + 0x800;      // 0x800 = DC offset  
17        DAC_Write((int)fdata, 1);  
18        ADC_Read();                                // receive the ADC result for sine wave  
19    }  
20  
21    void DAC_Init(void)                         // initialize SSI2 that connects to the DAC  
22    {  
23        SYSCTL->RCGCGPIO |= 0x02|0x08;          // enable clock to GPIOB & GPIOD  
24        SYSCTL->RCGCSSI |= 0x04;                  // enable clock to SSI2  
25  
26        GPIOB->AFSEL |= 0xD0;                    // PORTB 7, 6 & 4 for SSI2  
27        GPIOB->PCTL &= ~0xFF0F0000;              // PORTB 7, 6 & 4 for SSI2  
28        GPIOB->PCTL |= 0x22020000;  
29        GPIOB->DEN |= 0xDF;                      // PORTB 7, 6, 4 & 3 ~ 0 as digital pins  
30        GPIOB->DIR |= 0x0F;                      // PB3 ~ PB0 as output pins
```

Figure 8.39 The source file for the project DRADAC (Part I).

### 8.3.7.4.3 CREATE THE C SOURCE FILE DRADAC.C

```
1 //*****
2 // DRADAC.c - Main Application File for DAC-MCP4922 (SSI2) – Part II
3 //*****
4
5
6
7
8
9     GPIOD->DIR |= 0x48;           // use PD6 for DAC chip select
10    GPIOD->DEN |= 0x48;           // use PD3 for ADC chip select
11    GPIOD->DATA |= 0x48;          // set PD6 & PD3 to High to deselect DAC/ADC
12
13    SSI2->CR1 = 0;               // disable SSI2 and make it master
14    SSI2->CC = 0;                // use system clock
15    SSI2->CPSR = 16;             // clock prescaler divide by 16 gets 1 MHz clock
16    SSI2->CR0 = 0xF;              // clock rate div by 1, phase/polarity 0 0, mode freescale, data size 16
17    SSI2->CR1 = 2;               // enable SSI2
18 }
19
20 void DAC_Write(int value, int channel) // write a 16-bit value to DAC through SSI2
21 {
22     int config;
23
24     if (channel == 0) { config = 0x07000; }
25     else if (channel == 1) { config = 0xF000; }
26     GPIOD->DATA &= ~0x40;           // setup chip select to Low to select DAC
27     value = (value & 0x0FFF) | config; // configure 4 MSB with 12-bit data together
28     SSI2->DR = value;              // send 16-bit command-data to DAC
29     while (SSI2->SR & 0x10);       // wait for transmit done
30     GPIOD->DATA |= 0x40;            // set chip select to High to deselect DAC
31 }
32
33 void ADC_Read(void)
34 {
35     int adc;
36
37     GPIOD->DATA &= ~0x08;           // setup chip select to Low to select ADC
38     while (SSI2->SR & 0x10);       // wait for SSI2 data receiving done
39     adc = SSI2->DR;                // get the ADC data
40     GPIOB->DATA = adc;              // display ADC result on LEDs PB3 ~ PB0
41     GPIOD->DATA |= 0x08;             // set PD3 to High to deselect ADC
42 }
```

Figure 8.39 The source file for the project DRADAC (Part II).

#### 8.3.7.4.4 SETUP ENVIRONMENT TO BUILD AND RUN THE PROJECT

- Perform the following operations to setup the environment for this project:
  - Select the debug adapter **Stellaris ICDI** in the **Debug** tab under the **Project|Options for Target ‘Target 1’** menu item.
- Now build and run the project.
- As the project runs, you can use an oscilloscope to monitor **DAC0** and **DAC1** via **J11-1** and **J11-2** pins to get two waveforms, the saw-tooth and sinusoidal.
- The period may be different with our calculation values since this program repeatedly generates two waveforms in a **while()** loop and the period may be doubled. Also you can find that the intensity and color on the three-color LED on the TM4C123GXL EVB are periodically changed based on the ADC result.
- When checking two outputs with an oscilloscope, to get better results, you can comment out the sinusoidal signal to allow only the saw-tooth to input to the **ADC TLC548** and add a **10 ms** time delay in the **ADC\_Read()** subroutine. You cannot watch the input and output for the ADC since four LEDs cannot response to high frequency signals.

## 8.3.8 SSI API FUNCTIONS PROVIDED BY TIVAWARE™ PERIPHERAL DRIVER LIBRARY

- In this section, we introduce another way to access & interface the SSI modules to perform desired synchronous serial communication tasks via SSI modules.
- In the TivaWare™ Peripheral Driver Library, there are more than **20 API functions** used to interface to SSI modules. In this section, we concentrate on some most important and popular functions to access and interface to SSI modules to fulfill desired serial data communication tasks.
- These SSI API functions can be categorized into the following groups:
  1. The **initialization** and **configuration** functions:
    - **SSIClockSourceSet()**
    - **SSIClockSourceGet()**
    - **SSIConfigSetExpClk()**
  2. The **control** and **status** functions:
    - **SSIEnable()**
    - **SSIDisable()**
    - **SSIBusy()**



## 8.3.8 SSI API FUNCTIONS PROVIDED BY TIVAWARE™ PERIPHERAL DRIVER LIBRARY

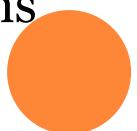
### 3. The **data processing** functions:

- **SSIDataPut()**
- **SSIDataPutNonBlocking()**
- **SSIDataGet()**

### 4. The **interrupt source and processing** functions:

- **SSIIntRegister()**
- **SSIIntEnable()**
- **SSIIntDisable()**
- **SSIIntClear()**
- **SSIIntStatus()**

- These API functions are contained in the **driverlib/ssi.c** file and defined in the **driverlib/ssi.h** file. The driverlib folder is located at **C:\ti\TivaWare\_C\_Series-2.0.1.11577**.
- Now let's have a quick introduction and discussion about these API functions based on their groups.



### 8.3.8.1 THE SSI MODULE INITIALIZATION AND CONFIGURATION FUNCTIONS

- Table 8.16 (next slide) shows some popular and important **SSI module initialization** and **configuration** API functions.
- The API function **SSICfgSetExpClk()** is a very important and useful function used to setup and configure the entire **SSI module**.
- Only after the SSI module is correctly configured by using this function, it can work properly to provide the normal functions.
- Some arguments used in this function are specified by using the system macros, such as **SSI\_FRF\_MOTO\_MODE\_0** and **SSI\_MODE\_MASTER**. These arguments must be used correctly to make the configurations correct.



### 8.3.8.1 THE SSI MODULE INITIALIZATION AND CONFIGURATION FUNCTIONS

Table 8.16 The SSI module initialization and configuration functions.

API Function	Parameter	Description
void <b>SSIClockSourceSet(</b> uint32_t ui32Base, uint32_t ui32Source) <b>)</b>	<b>ui32Base</b> is the base address of the SSI port. <b>ui32Source</b> is the baud clock source for the SSI.	Select clock source for the SSI. The possible clock source are the system clock ( <b>SSI_CLOCK_SYSTEM</b> ) or the precision internal oscillator ( <b>SSI_CLOCK_PIOSC</b> ). Changing the baud clock source changes the data rate generated by the SSI. Therefore, the data rate should be reconfigured after any change to the SSI clock source.
uint32_t <b>SSIClockSourceGet(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the SSI port.	Return the data clock source for the specified SSI. The returned clock source is either <b>SSI_CLOCK_SYSTEM</b> or <b>SSI_CLOCK_PIOSC</b> .
void <b>SSIConfigSetExpClk(</b> uint32_t ui32Base, uint32_t ui32SSIClk, uint32_t ui32Protocol, uint32_t ui32Mode, uint32_t ui32BitRate, uint32_t ui32DataWidth) <b>)</b>	<b>ui32Base</b> specifies the SSI module base address. <b>ui32SSIClk</b> is the rate of the clock supplied to the SSI. <b>ui32Protocol</b> specifies the data transfer protocol. <b>ui32Mode</b> specifies the mode of operation. <b>ui32BitRate</b> specifies the clock rate. <b>ui32DataWidth</b> specifies number of bits transferred per frame	Configure the synchronous serial interface. It sets the SSI protocol, mode of operation, bit rate, and data width.  The <b>ui32Protocol</b> parameter can be one of the following values: <b>SSI_FRF_MOTO_MODE_0</b> , <b>SSI_FRF_MOTO_MODE_1</b> , <b>SSI_FRF_MOTO_MODE_2</b> , <b>SSI_FRF_MOTO_MODE_3</b> , <b>SSI_FRF_TI</b> , or <b>SSI_FRF_NMW</b> .  The <b>ui32Mode</b> parameter defines the operating mode of the SSI. The SSI module can operate as a master or slave; if it is a slave, the SSI can be configured to disable output on its serial output line. The ui32Mode parameter can be one of the following values: <b>SSI_MODE_MASTER</b> , <b>SSI_MODE_SLAVE</b> , or <b>SSI_MODE_SLAVE_OD</b> .  The <b>ui32BitRate</b> parameter defines the bit rate for the SSI. This bit rate must satisfy the following clock ratio criteria: FSSI $\geq$ 2 $\times$ bit rate (master mode); this speed < 25 MHz. FSSI $\geq$ 12 $\times$ bit rate or 6 $\times$ bit rate (slave modes), depending on the capability of the specific microcontroller.  The <b>ui32DataWidth</b> parameter defines the width of the data transfers and can be a value between 4 and 16.

### 8.3.8.2 THE SSI MODULE CONTROL AND STATUS FUNCTIONS

- Table 8.17 shows some popular and important SSI module control and status API functions.
- Three API functions are involved in this group, **SSIEnable()**, **SSIDisable()** and **SSIBusy()**.
- The first two functions are used to control the availability of the SSI modules, and the third function is used to check the working status of the SSI modules.

Table 8.17 The SSI module control and status functions.

API Function	Parameter	Description
void <b>SSIEnable(uint32_t ui32Base)</b>	<b>ui32Base</b> specifies the SSI module base address.	Enable operation of the synchronous serial interface. The synchronous serial interface must be configured before it is enabled.
void <b>SSIDisable(uint32_t ui32Base)</b>	<b>ui32Base</b> is the base address of the SSI port.	Disable operation of the synchronous serial interface.
bool <b>SSIBusy(uint32_t ui32Base)</b>	<b>ui32Base</b> specifies the SSI module base address.	Allow the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If <b>false</b> is returned, then the transmit FIFO is empty and all bits of the last transmitted word have left the hardware shift register.

### 8.3.8.2 THE SSI MODULE CONTROL AND STATUS FUNCTIONS

- All SSI modules must be **initialized** and **configured** to the proper working mode **before** they can be used to transfer and receive any serial data.
- However, those initializations and configurations must be performed when the SSI modules are **disabled**. This means that no any initialization and configuration job can be performed if the SSI has been enabled.
- The **SSIBusy()** function can return a Boolean value to indicate whether the SSI is busy or idle. A returned **True** means that the SSI module is busy in working, and **False** indicates that the SSI module is idle and any new data can be sent to it to begin a new transfer.



### 8.3.8.3 THE SSI MODULE DATA PROCESSING FUNCTIONS

- Table 8.18 shows some popular and important SSI module **data processing** API functions.

Table 8.18 The SSI module data processing functions.

API Function	Parameter	Description
<code>void <b>SSIDataPut(</b>     uint32_t ui32Base,     uint32_t ui32Data)<b>)</b></code>	<b>ui32Base</b> specifies the SSI module base address. <b>ui32Data</b> is the data to be transmitted over the SSI.	Place the data into the transmit FIFO of the specified SSI. If there is no space available in the transmit FIFO, this function waits until there is space available before returning. The upper <b>32-N</b> bits of <b>ui32Data</b> are discarded by the hardware, where <b>N</b> is the data width as configured by <b>SSIConfigSetExpClk()</b> . If the SSI is configured for 8-bit data width, the upper 24 bits of <b>ui32Data</b> are discarded.
<code>int32_t <b>SSIDataPutNonBlocking(</b>     uint32_t ui32Base,     uint32_t ui32Data)<b>)</b></code>	<b>ui32Base</b> specifies the SSI module base address. <b>ui32Data</b> is the data to be transmitted over the SSI.	Place the data into the transmit FIFO of the specified SSI module. If there is no space in the FIFO, then this function returns a zero.
<code>void <b>SSIDataGet(</b>     uint32_t ui32Base,     uint32_t *pui32Data)<b>)</b></code>	<b>ui32Base</b> specifies the SSI module base address. <b>pui32Data</b> is a pointer to a storage location for data that was received over the SSI.	Get received data from the receive FIFO of the specified SSI and place it into the location specified by the <b>pui32Data</b> parameter. If there is no data available, this function waits until data is received before returning. Only the lower <b>N</b> bits of the value written to <b>pui32Data</b> contain valid data, where <b>N</b> is the data width as configured by <b>SSIConfigSetExpClk()</b> .

### 8.3.8.3 THE SSI MODULE DATA PROCESSING FUNCTIONS

- The **SSIDataPut()** and **SSIDataPutNonBlocking()** functions have the similar function and both of them place the transferring data into the transmit FIFO of the specified SSI module. However, the **SSIDataPut()** will keep waiting and place the data until a space in the transmit FIFO is available. The **SSIDataPutNonBlocking()** never wait if no space is available in the transmit FIFO and directly returns a zero if no space available.
- Similar situation is true to the **SSIDataGet()** and **SSIDataGetNonBlocking()** functions.
- For both **SSIDataPut()** and **SSIDataPutNonBlocking()** functions, the upper **32-N** bits data are ignored and only the lower **N** bits data are transmitted to the transmit FIFO.
- This **N** is the data width defined in the **SSIConfigSetExpClk()** function. Similarly, only the lower **N** bits data are considered as valid data items when using the **SSIDataGet()** and **SSIDataGetNonBlocking()** functions to receive data from the specified receive FIFO.

### 8.3.8.4 THE SSI MODULE INTERRUPT SOURCE AND PROCESSING FUNCTIONS

- Table 8.19 (next slide) shows some popular and important SSI module **interrupt processing** API functions.
- Before any interrupt can be handled and responded by the selected SSI module, the API function **SSIIntRegister()** must be executed to register the related interrupt handler. The second argument of this function is a pointer point to the handler subroutine or an **entry address** of the ISR.
- The argument **ui32IntFlags** for the **SSIIntEnable()**, **SSIIntDisable()** and **SSIIntClear()** API functions is a **bit mask** for the selected interrupt sources. A corresponding bit value of **1** indicates that the related interrupt source is selected, if a bit is **0**, and the related interrupt source is not included. Some system macros should be used for those interrupt sources.
- For the **SSIIntStatus()** function, the second argument **bMasked** is a Boolean variable. When it is **True**, it indicates that the masked interrupt status is required and returned. If this value is **False**, only the raw interrupt status is needed and returned.

## 8.3.8.4 THE SSI MODULE INTERRUPT SOURCE AND PROCESSING FUNCTIONS

Table 8.19 The SSI module interrupt processing functions.

API Function	Parameter	Description
<pre>void <b>SSIIntRegister(</b>     uint32_t ui32Base,     void (*pfnHandler)(void))</pre>	<p><b>ui32Base</b> specifies the SSI module base address.  <b>pfnHandler</b> is a pointer to the function to be called when a synchronous serial interface interrupt occurs.</p>	Register the handler to be called when an SSI interrupt occurs. This function enables the global interrupt in the interrupt controller; specific SSI interrupts must be enabled via <b>SSIIntEnable()</b> . It is the interrupt handler responsibility to clear the interrupt source via <b>SSIIntClear()</b> .
<pre>void <b>SSIIntEnable(</b>     uint32_t ui32Base,     uint32_t ui32IntFlags)</pre>	<p><b>ui32Base</b> specifies the SSI module base address.  <b>ui32IntFlags</b> is a bit mask of the interrupt sources to be enabled.</p>	Enable the indicated SSI interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor. The <b>ui32IntFlags</b> parameter can be any of the <b>SSI_TXFF</b> , <b>SSI_RXFF</b> , <b>SSI_RXTO</b> , or <b>SSI_RXOR</b> values.
<pre>void <b>SSIIntDisable(</b>     uint32_t ui32Base,     uint32_t ui32IntFlags)</pre>	<p><b>ui32Base</b> specifies the SSI module base address.  <b>ui32IntFlags</b> is a bit mask of the interrupt sources to be disabled.</p>	Disable the indicated SSI interrupt sources. The <b>ui32IntFlags</b> parameter can be any of the <b>SSI_TXFF</b> , <b>SSI_RXFF</b> , <b>SSI_RXTO</b> , or <b>SSI_RXOR</b> values.
<pre>void <b>SSIIntClear(</b>     uint32_t ui32Base,     uint32_t ui32IntFlags)</pre>	<p><b>ui32Base</b> specifies the SSI module base address.  <b>ui32IntFlags</b> is a bit mask of the interrupt sources to be cleared.</p>	Clear the specified SSI interrupt sources so that they no longer assert. This function must be called in the interrupt handler to keep the interrupts from being triggered again immediately upon exit. The <b>ui32IntFlags</b> parameter can consist of either or both the <b>SSI_RXTO</b> and <b>SSI_RXOR</b> .
<pre>uint32_t <b>SSIIntStatus(</b>     uint32_t ui32Base,     bool bMasked)</pre>	<p><b>ui32Base</b> specifies the SSI module base address.  <b>bMasked</b> is <b>false</b> if the raw interrupt status is required or <b>true</b> if the masked interrupt status is required.</p>	Return the interrupt status for the SSI module. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned.

### 8.3.8.5 BUILD AN EXAMPLE PROJECT TO INTERFACE SERIAL PERIPHERALS USING THE SSI MODULE

- In this section, we use the **SSI related API functions** to build an interfacing project to access and display desired letters and numbers on the LCD device.
- We can modify the project **DRALCD** we built in section 8.3.5.4 and use the related API functions to replace direct register access methods to access and interface LCD controller SPLC780 via 74VHCT595.



### 8.3.8.5.1 CREATE A NEW SOFTWARE DRIVER MODEL PROJECT SDLCD

- Perform the following operations to create a new project **SDLCD**:
  1. Open the Windows Explorer to create a new folder named **SDLCD** under the **C:\ARM Class Projects\Chapter 8** folder.
  2. Open the Keil® ARM-MDK µVersion5 and go to **Project|New µVersion Project** menu item to create a new µVersion Project. On the opened wizard, browse to our new folder **SDLCD** that is created in step 1 above. Enter **SDLCD** into the **File name** box and click on the **Save** button to create this project.
  3. On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series** and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
  4. Next the Software Components wizard is opened, and you need to setup the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, and click on the **OK** button since we need these two components to build our project.
- Since this project is longer, a header file and a C source file are needed.

### 8.3.8.5.2 CREATE THE HEADER FILE SDLCD.H

- Create a new header file **SDLCD.h** and enter codes shown in Figure 8.40 into this file.

```
1 //*****  
2 // SDLCD.h - Header File for the Main Application File SDLCD.c  
3 //*****  
4 #include <stdint.h>  
5 #include <stdbool.h>  
6 #include "inc/hw_memmap.h"  
7 #include "inc/hw_ssi.h"  
8 #include "inc/hw_types.h"  
9 #include "driverlib/ssi.h"  
10 #include "driverlib/gpio.h"  
11 #include "driverlib/sysctl.h"  
12 #define RS 1                      // Q0 bit for RS (Reg Select)  
13 #define EN 2                      // Q1 bit for E (Enable LCD)  
14 #define BL 4                      // Q2 bit for BL (Backlight)  
15 #define GPIO_PB7_SSI2TX    0x00011C02  
16 #define GPIO_PB4_SSI2CLK    0x00011002  
17 void delay_ms(int time);  
18 void delay_us(int time);  
19 void LCD_cd_Write(char data, unsigned char control);  
20 void LCD_Comd(unsigned char cmd);  
21 void LCD_Data(char data);  
22 void LCD_Init(void);  
23 void SSI2_Write(unsigned char data);
```

Figure 8.40 The codes for the header file **SDLCD.h**.

### 8.3.8.5.3 CREATE THE C SOURCE FILE SDLCD.C

- Create a new C file **SDLCD.c** and enter the codes shown in Figure 8.41 into this C file (next slide).
- In this project, we want to display “**WELCOME TO JCSU**” in the first line on the LCD, and “**GOOD JOB**” in the center of the second line on the LCD.
- Therefore the function **LCD\_Comd(0xC4)** is used to select the position or address of the DDRAM for the first letter in the second line to make the second line’s letters located at the center.
- Refer to Figure 8.30 to get more details about this start position or the address of the DDRAM and this function.
- Now we need to pay more attention to this piece of codes to see how it works. For those codes that are duplicated from the project **DRALCD.c**, we will skip them and only pay attention to those new added API functions.



### 8.3.8.5.3 CREATE THE C SOURCE FILE SDLCD.C

```
1 //*****
2 // SDLCD.c - Main Application File for LCD Project (Part I)
3 //*****
4 #include "SDLCD.H"
5
6 int main(void)
7 {
8     LCD_Init();                                // initialize LCD controller
9     LCD_Command(1);                           // clear screen, move cursor to home
10    LCD_Data('W'); LCD_Data('E'); LCD_Data('L'); LCD_Data('C'); LCD_Data('O');
11    LCD_Data('M'); LCD_Data('E'); LCD_Data(' '); LCD_Data('T'); LCD_Data('O');
12    LCD_Data(' '); LCD_Data('J'); LCD_Data('C'); LCD_Data('S'); LCD_Data('U'); ;LCD_Data('!');
13    LCD_Command(0xC4);
14    LCD_Data('G'); LCD_Data('O'); LCD_Data('O'); LCD_Data('D'); LCD_Data(' ');
15    LCD_Data('J'); LCD_Data('O'); LCD_Data('B');
16 }
17 void LCD_Init(void)
18 {
19     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
20     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
21     SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI2);
22     GPIOPinTypeSSI(GPIO_PORTB_BASE, GPIO_PIN_7|GPIO_PIN_4);
23     GPIOPinConfigure(GPIO_PB7_SSI2TX);
24     GPIOPinConfigure(GPIO_PB4_SSI2CLK);
25     GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_6);
26     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0xFF);
27     SSIDisable(SSI2_BASE);
28     SSICfgSetExpClk(SSI2_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
29                      SSI_MODE_MASTER, 1000000, 8);
30     SSIEnable(SSI2_BASE);
```

Figure 8.41 The source codes for the project SDLCD (Part I).

### 8.3.8.5.3 CREATE THE C SOURCE FILE SDLCD.C

```
1 //*****  
2 // SDLCD.c - Main Application File for LCD Project (Part II)  
3 //*****  
4  
31     delay_ms(20);                                // LCD controller reset sequence  
32     LCD_cd_Write(0x30, 0);                      // send reset code 1 two times to SPLC780  
33     LCD_cd_Write(0x30, 0);                      // send reset code 2 to SPLC780  
34     LCD_cd_Write(0x30, 0);  
35     LCD_cd_Write(0x20, 0);                      // use 4-bit data mode  
36     delay_ms(1);  
37     LCD_Comd(0x28);                            // set 4-bit data, 2-line, 5x7 font  
38     LCD_Comd(0x06);                            // move cursor right  
39     LCD_Comd(0x0C);                            // turn on display, cursor off - no blinking  
40     LCD_Comd(0x01);                            // clear screen, move cursor to home  
41 }  
42 void LCD_cd_Write(char data, unsigned char control)  
43 {  
44     data &= 0xF0;                                // clear lower nibble for data  
45     control &= 0x0F;                             // clear upper nibble for control  
46     SSI2_Write(data | control | BL);           // RS = 0, R/W = 0  
47     SSI2_Write(data | control | EN | BL);       // pulse E  
48     SSI2_Write(data | BL);  
49     SSI2_Write(BL);  
50 }  
51 void SSI2_Write(unsigned char data)  
52 {  
53     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0xBF); // output a Low on PC6 for STCP  
54     SSIDataPut(SSI2_BASE, data);                  // transmit data into FIFO  
55     while(SSIBusy(SSI2_BASE)) {};                // wait for transmit done  
56     GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0xFF); // output a High on PC6 for STCP  
57 }
```

Figure 8.41 The source codes for the project SDLCD (Part II).

### 8.3.8.5.3 CREATE THE C SOURCE FILE SDLCD.C

- Some key points about this C File are:

- In line 22, the API function **GPIOPinTypeSSI()** is called to setup the GPIO pins as the **SSI** function pins. This function only defined the operational type for the selected pins but not configure any of them. This function must be combined with another API function **GPIOPinConfigure()** together to configure a GPIO pin, as we did in lines 23 and 24. Both **PB7 (SSI2TX)** and **PB4 (SSI2CLK)** are defined and configured in these lines. Two macro, **GPIO\_PB7\_SSI2TX** and **GPIO\_PB7\_SSI2TX**, which are defined in the **pin\_map.h** header file, must be used for these configurations.
- Since **PC6** is used as the **/CS\_LCD** control signal to provide a positive-going-edge triggering signal for the 74VHCT595, the API function **GPIOPinTypeGPIOOutput()** is used to define **PC6** as an output pin in line 25.
- In line 26, the API function **GPIOPinWrite()** sets **PC6** to **High** when it is idle.
- The codes in lines 27 ~ 30 are used to configure SSI module 2 via related API functions. First the SSI2 is **disabled** by using the function **SSIDisable()** in line 27 since no any configuration can be performed until the SSI module is disabled.
- In line 28, API function **SSICfgSetExpClk()** configures and setups operational parameters for the SSI2 module. The fifth parameter, bit rate for the **SSI2Clk**, is defined as 1 MHz with **1000000**. The unit for this parameter is Hz.
- After the configuration, the SSI2 module is enabled in line 30 using the **SSIEnable()**.

### 8.3.8.5.3 CREATE THE C SOURCE FILE SDLCD.C

- The codes for the subroutine **SSI2\_Write()** are totally new.
- First in line 53, the API function **GPIOPinWrite()** is called to reset the **/CS\_LCD (PC6)** signal to Low to prepare a **Low-to-High** (positive-going-edge) signal to the **STCP** on the serial shift register 74VHCT595.
- In line 54, the API function **SSIDataPut()** sends data item to the transmit FIFO.
- Then a **while()** loop is used to wait for the data sent to the transmit FIFO to be transmitted in line 55. The API function **SSIBusy()** works as the loop condition for this **while()** loop and the subroutine will not continue to execute the next instruction until this loop condition becomes **False**, which means that the SSI2 has completed this data transmission and a False is returned from the function **SSIBusy()**.
- In line 56, the function **GPIOPinWrite()** sets the **/CS\_LCD (PC6)** to **High** to create a positive-going-edge signal to trigger the 74VHCT595 to start a serial-to-parallel data conversion and output the conversion result to the LCD controller SPLC780.
- The rest codes are identical with those codes in the project **DRALCD**.
- We leave some projects as home works for the readers, and you can modify projects, such as **DRAADC** and **DRALED**, by using the API functions to rebuild those projects. These projects are included in **Lab8\_3** and **Lab8\_4**.



## 8.4 INTER-INTEGRATED CIRCUIT (I<sub>2</sub>C) INTERFACE

- The **Inter-Integrated Circuit (I<sub>2</sub>C)** module is exactly a serial communication bus system to enable I<sub>2</sub>C devices or peripherals to perform high-speed and bi-directional data transfer via two wire design, a serial data line **SDA** and a serial clock line **SCL**.
- By using this bus system, the microcontroller and other I<sub>2</sub>C compatible devices can interface to external I<sub>2</sub>C devices such as serial memory (RAMs and ROMs), networking devices, LCDs, tone generators, external clock and so on.
- The I<sub>2</sub>C bus may also be used for system testing and diagnostic purposes in product development and manufacturing. The TM4C123GH6PM microcontrollers include and provide the ability to communicate (both transmit and receive) with other I<sub>2</sub>C devices on the bus.
- In the TM4C123GH6PM MCU system, **four (4) I<sub>2</sub>C modules (I<sub>2</sub>C<sub>0</sub> ~ I<sub>2</sub>C<sub>3</sub>)** are provided to support the I<sub>2</sub>C related data operations. Each I<sub>2</sub>C module can work as a master or slave to perform data transmit or receive operation via two-wire bus.



## 8.4 INTER-INTEGRATED CIRCUIT (I<sub>2</sub>C) INTERFACE

- The **I<sub>2</sub>C master** and **slave** modules provide the ability to communicate to other I<sub>2</sub>C devices over an I<sub>2</sub>C bus. The I<sub>2</sub>C bus is specified to support devices that can both transmit and receive (write and read) data.
- Devices on the I<sub>2</sub>C bus can be designated as either a master or a slave. The Tiva™ I<sub>2</sub>C modules support both sending and receiving data as either a master or a slave, and also support the simultaneous operation as both a master and a slave.
- Finally, the Tiva™ I<sub>2</sub>C modules can operate at four speeds: **Standard** (100 kbps), **Fast** (400 kbps), **Fast-Mode Plus** (1 Mbps) and **High-Speed** (3.33 Mbps).
- Both master and slave I<sub>2</sub>C modules can generate interrupts. The I<sub>2</sub>C master module generates interrupts when a transmit or a receive operation is completed or aborted due to an error; and on some devices when a clock low timeout has occurred.
- The I<sub>2</sub>C slave module generates interrupts when data has been sent or requested by a master; and on some devices, when a **START** or **STOP** condition is present.

## 8.4.1 I<sup>2</sup>C MODULE BUS CONFIGURATION AND OPERATIONAL STATUS

- Each **I<sup>2</sup>C module** is composed of both master and slave units and can be identified by a unique address.
- A master-initiated communication generates the clock signal, **SCL**. For proper operation, the **SDA** pin must be configured as an open-drain signal.
- Due to the internal circuitry that supports high-speed operation, the **SCL** pin must **not be configured** as an open-drain signal.
- Both **SDA** and **SCL** signals must be connected to a positive power supply voltage using a pull-up resistor. When both wires are in this status, it is called that the bus is **idle**. A typical I<sup>2</sup>C bus configuration is shown in Figure 8.42.

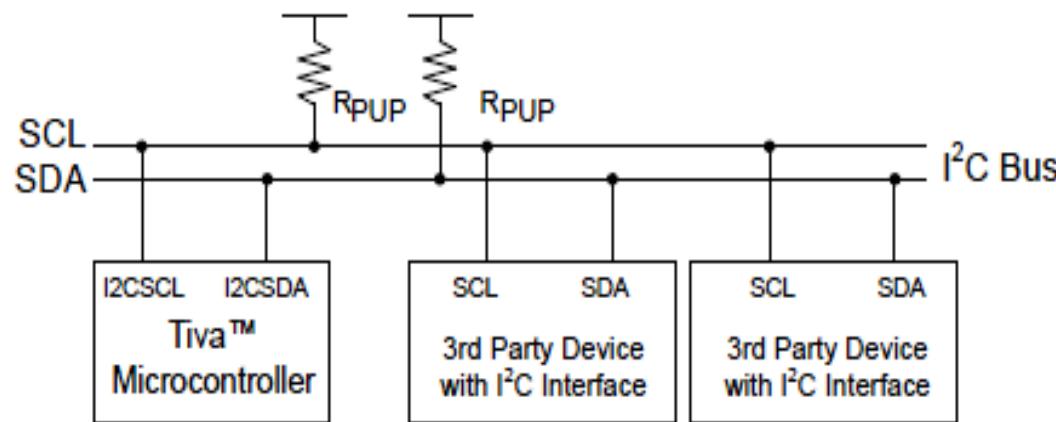


Figure 8.42 The I<sup>2</sup>C bus configuration and status.

## 8.4.1 I2C MODULE BUS CONFIGURATION AND OPERATIONAL STATUS

- The I<sub>2</sub>C bus uses only two wires, **SDA** and **SCL**, named **I<sub>2</sub>CSDA** and **I<sub>2</sub>CSCL** to perform bi-directional data communications on TM4C123GH6PM MCUs. **SDA** is the bi-directional serial data line and **SCL** is the bi-directional serial clock line. The bus is considered **idle** when both lines are High.
- Every transaction on the I<sub>2</sub>C bus is nine (**9**) bits long, consisting of **eight data** bits and a **single acknowledge** bit.
- The number of bytes per transfer that defined as the time period between a valid **START** and **STOP** condition, is unrestricted, but each data byte has to be followed by an acknowledge bit, and data must be transferred **MSB** first.
- The **START** and **STOP** are two conditions or two states defined in the protocol of the I<sub>2</sub>C module. A **High-to-Low** transition on the **SDA** line while the **SCL** is **High** is defined as a **START** condition, and a **Low-to-High** transition on the **SDA** line while **SCL** is **High** is defined as a **STOP** condition.
- The bus is considered **busy** after a **START** condition and free after a **STOP** condition.

## 8.4.1 I<sup>2</sup>C MODULE BUS CONFIGURATION AND OPERATIONAL STATUS

- Figure 8.43 shows an illustration for both **START** and **STOP** conditions.

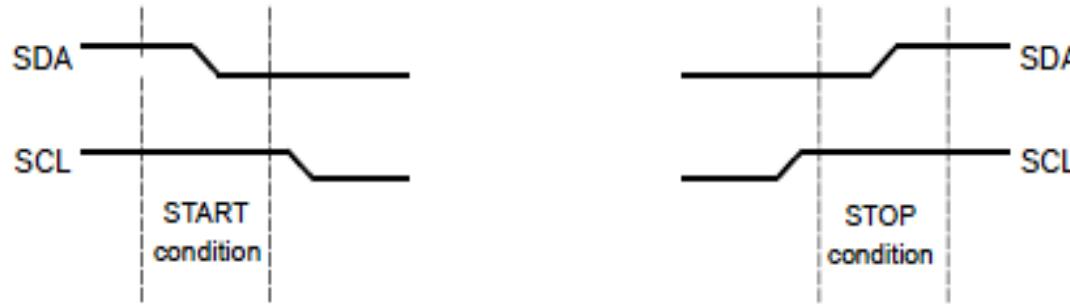


Figure 8.43 The definition of START and STOP conditions.

- Regularly, the data on the **SDA** line must be stable during the **High** period of the **SCL** line, and can only be changed during the **Low** period of the **SCL** line. In this way, the **START** and **STOP** state can be clearly distinguished with the normal data signals on the SDA line.
- The **STOP** bit determines if the cycle stops at the end of the data cycle or continues on to a repeated **START** condition. When a data communication operation is completed or aborted due to an error, the interrupt pin becomes active and the data may be read from the **I<sup>2</sup>C Master Data Register (I<sup>2</sup>CMDR)**.

## 8.4.2 I<sup>2</sup>C MODULE ARCHITECTURE AND FUNCTIONAL BLOCK DIAGRAM

- A functional block diagram of each **I<sup>2</sup>C module** is shown in Figure 8.44.

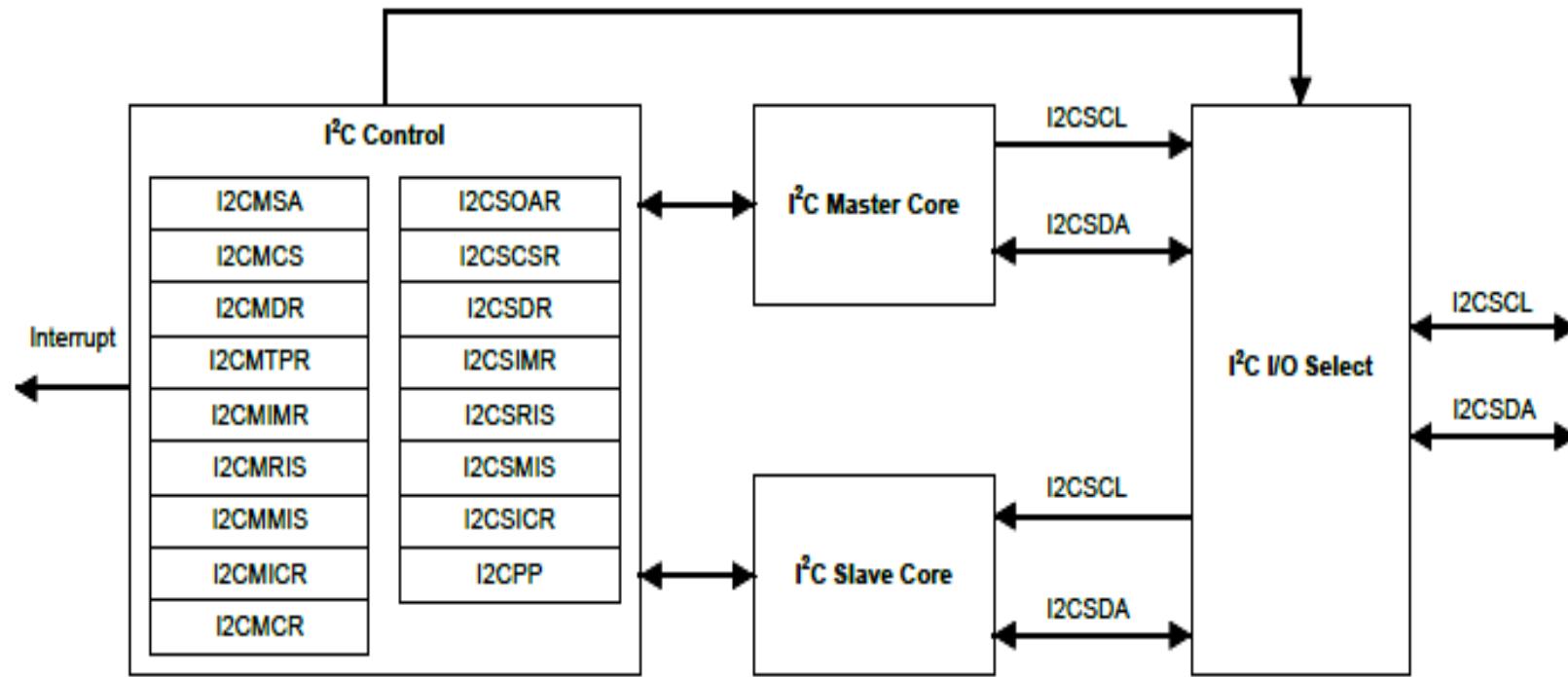


Figure 8.44 The functional block diagram of each I<sup>2</sup>C module.

- It can be found from Figure 8.44 that each I<sup>2</sup>C module contains **two cores**; the **I<sup>2</sup>C Master Core** and the **I<sup>2</sup>C Slave Core**. Each core controls and coordinates the data transmit and receive operations via related registers.

## 8.4.2 I2C MODULE ARCHITECTURE AND FUNCTIONAL BLOCK DIAGRAM

- All I<sub>2</sub>C **control registers** are divided into two groups; the **master** and the **slave** group with **I2CM** and **I2CS** as prefix for each group. Some important and popular registers are listed in Table 8.20.
- Each of these registers are **32-bit** registers, however some registers only used limited number of bits to perform their functions.

Table 8.20 Most popular and important I<sub>2</sub>C registers.

Master Control Register		Slave Control Register	
Register Name	Function	Register Name	Function
<b>I2C Master Slave Address Register (I2CMSTA)</b>	Hold the master and target slave address.	<b>I2C Slave Own Address Register (I2CSOAR)</b>	Hold the slave address.
<b>I2C Master Control/Status Register (I2CMCS)</b>	Hold the master control and status.	<b>I2C Slave Control/Status Register (I2CSCSR)</b>	Hold the slave control and status.
<b>I2C Master Data Register (I2CMDR)</b>	Hold the master data.	<b>I2C Slave Data Register (I2CSDR)</b>	Hold the slave data.
<b>I2C Master Configuration Register (I2CMCR)</b>	Configure the operational mode.	<b>I2C Slave ACK Control Register (I2CSACKCTL)</b>	Control the ACK or NACK signal for the slave device.

### 8.4.3 I2C MODULE DATA TRANSFER FORMAT AND FRAME

- In the I2C module, the data transfers follow the format shown in Figure 8.45.
- After the **START** condition, a **slave address** is first transmitted. This address is **7-bit long** followed by an **R/S** bit, which is a data direction bit in the **I2CMSCA** register.
- If the **R/S** bit is **0**, it indicates a sending or a transmit operation. If it is **1**, it indicates a receiving or a request for data.

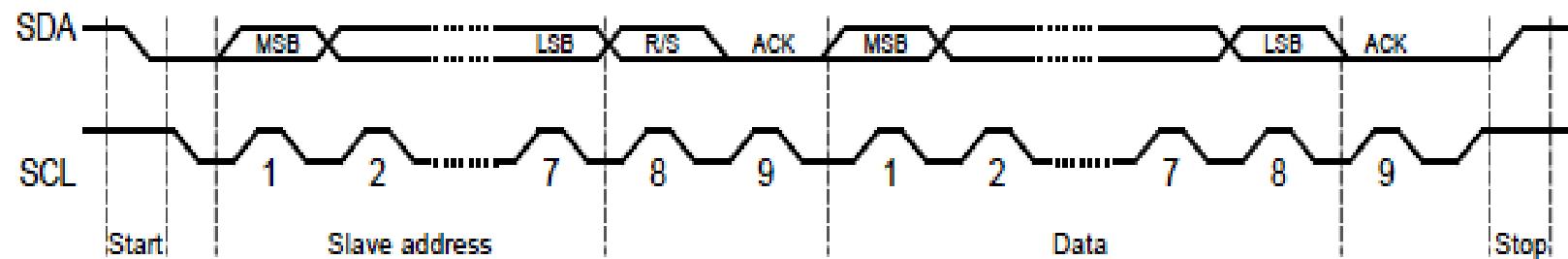


Figure 8.45 The I2C data transfer format and frame.

- A data transfer is always terminated by a **STOP** condition generated by the master, however, a master can initiate communications with another device on the bus by generating a repeated **START** condition and addressing another slave without first generating a **STOP** condition.

### 8.4.3 I2C MODULE DATA TRANSFER FORMAT AND FRAME

- The **first seven bits** of the first byte make up the slave address. The **eighth** bit **R/S** determines the direction of the message. The **ninth** bit is the **ACK** bit that is generated by the master.
- When the I<sub>2</sub>C module operates in **Master receiver** mode, the **ACK** bit is normally set causing the I<sub>2</sub>C bus controller to transmit an acknowledge automatically after each byte. This bit must be **cleared** when the I<sub>2</sub>C bus controller requires no further data to be transmitted from the slave transmitter.
- Both a slave address and the following data must be transferred starting with the **MSB** and each piece of address or each data must be composed of nine (**9**) bits.
- All bus transactions have a required acknowledge clock cycle that is generated by the master. During the acknowledge (**ACK**) cycle, the transmitter that can be the master or slave releases the **SDA** line as shown in Figure 8.45.
- To acknowledge the transaction, the receiver must pull down **SDA** during the acknowledge clock cycle.

#### 8.4.4 I2C MODULE OPERATIONAL SEQUENCE

- The I<sub>2</sub>C module can work as either a master or a slave with data transmitting and receiving mode.
- The master data transmission and receiving operations are controlled by the **I<sub>2</sub>C master group registers**, and the slave data transmission and receiving operations are controlled by the **I<sub>2</sub>C slave group registers**.
- The operational sequences for each different mode are listed below.

#### 8.4.4.1 I<sub>2</sub>C MODULE WORKS IN THE MASTER TRANSMIT MODE

1. When the I<sub>2</sub>C module works in the **master transmit mode**, the **ACK** signal is generated by the master and attached at the end of each transmitted data.
2. The **target slave address** is first sent after the **STRAT** condition by writing it into the **I<sub>2</sub>CMSA**. Then the transmitted data is written into the **I<sub>2</sub>CMDR** and the **I<sub>2</sub>CMCS** register is checked to wait for the **ACK** and the bus to be idle.
3. A **ox3 (H:ACK:STOP:START:RUN = ox011)** is written to the **I<sub>2</sub>CMCS** to inform the slave that a data transmission starts. The **x** means either **0** or **1** is ok, the **STOP** is **0** means that this is a multiple bytes data transfer and cannot stop right now.
4. Check the **I<sub>2</sub>CMCS** status for the **ACK** and wait the bus to be idle.
5. Write the next data into the **I<sub>2</sub>CMDR** to prepare to transmit the next data.
6. Check if the last data has been transmitted. If not, write **ox001** to the **I<sub>2</sub>CMCS** to continue to send the next data.
7. If the last data is encountered, write **ox101** to the **I<sub>2</sub>CMCS** to transmit the **STOP** status to inform the slave that the transmission is done.
8. Check the **I<sub>2</sub>CMCS** for the **ACK** signal and wait for the bus to be idle.
9. The transfer is done and the bus goes to idle status.



#### 8.4.4.1 I2C MODULE WORKS IN THE MASTER TRANSMIT MODE

- From this operation sequence, it can be found that the **I2CMCS** register is a double-function register:
  - When **reading**, it provides the **working status of the slave**.
  - When **writing**, it setups the **operational function for the master**.
- Figure 8.46 (next slide) shows the operational sequence for the I2C master working as a transmitter.



### 8.4.4.1 I2C MODULE WORKS IN THE MASTER TRANSMIT MODE

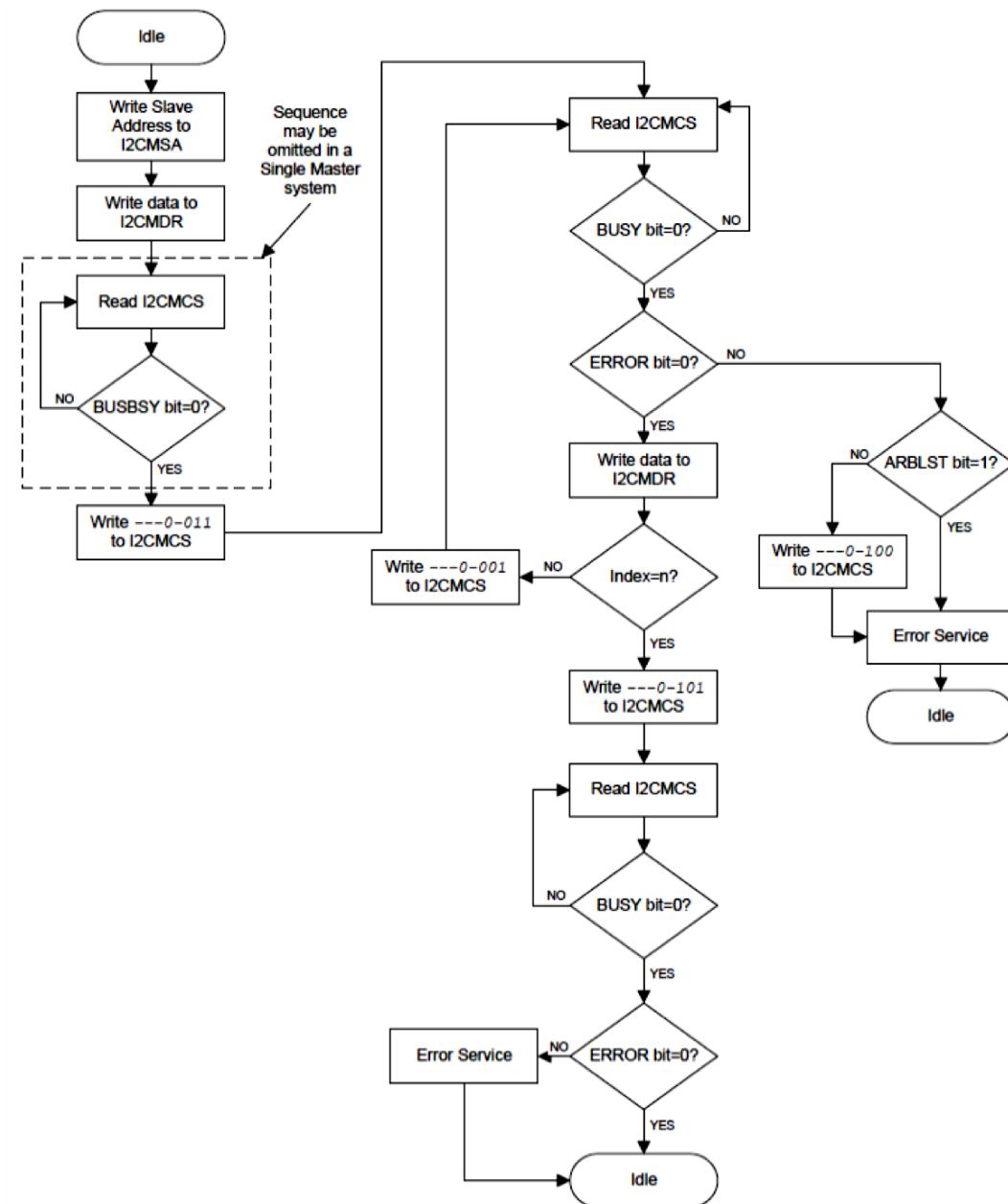


Figure 8.46 The operational sequence of the master working in the transmit mode.

#### 8.4.4.2 I2C MODULE WORKS IN THE MASTER RECEIVE MODE

1. When the I<sub>2</sub>C module operates in **Master receiver** mode, the **ACK** bit is normally set causing the I<sub>2</sub>C bus controller to transmit an acknowledge automatically after each byte. This bit must be cleared when the I<sub>2</sub>C bus controller requires no further data to be transmitted from the slave transmitter.
  2. Write the **target slave address** to the **I<sub>2</sub>CMSA** register to transmit it via the bus.
  3. The **I<sub>2</sub>CMCS** register is checked to wait for the **ACK** and the bus to be idle.
  4. Write **0xB (H:ACK:STOP:START:RUN = 01011)** to **I<sub>2</sub>CMCS** register to start the data receiving.
  5. Check the **I<sub>2</sub>CMCS** status for the **ACK** and wait the bus to be idle.
  6. When the **ACK** is obtained & the data is received, read the data from the **I<sub>2</sub>CMDR**.
  7. Check if the last data has been received. If not, write **01001 (0x9)** to the **I<sub>2</sub>CMCS** to continue to receive the next data.
  8. If the last data coming, write **00101 (0x5)** to the **I<sub>2</sub>CMCS** to inform the slave that the data receiving is done.
  9. Check the **I<sub>2</sub>CMCS** for the **ACK** signal and wait for the bus to be idle.
  10. Read the last data from the **I<sub>2</sub>CMDR**.
- The complete operational sequence of the master working in the receiving mode is shown in Figure 8.47.

## 8.4.4.2 I2C MODULE WORKS IN THE MASTER RECEIVE MODE

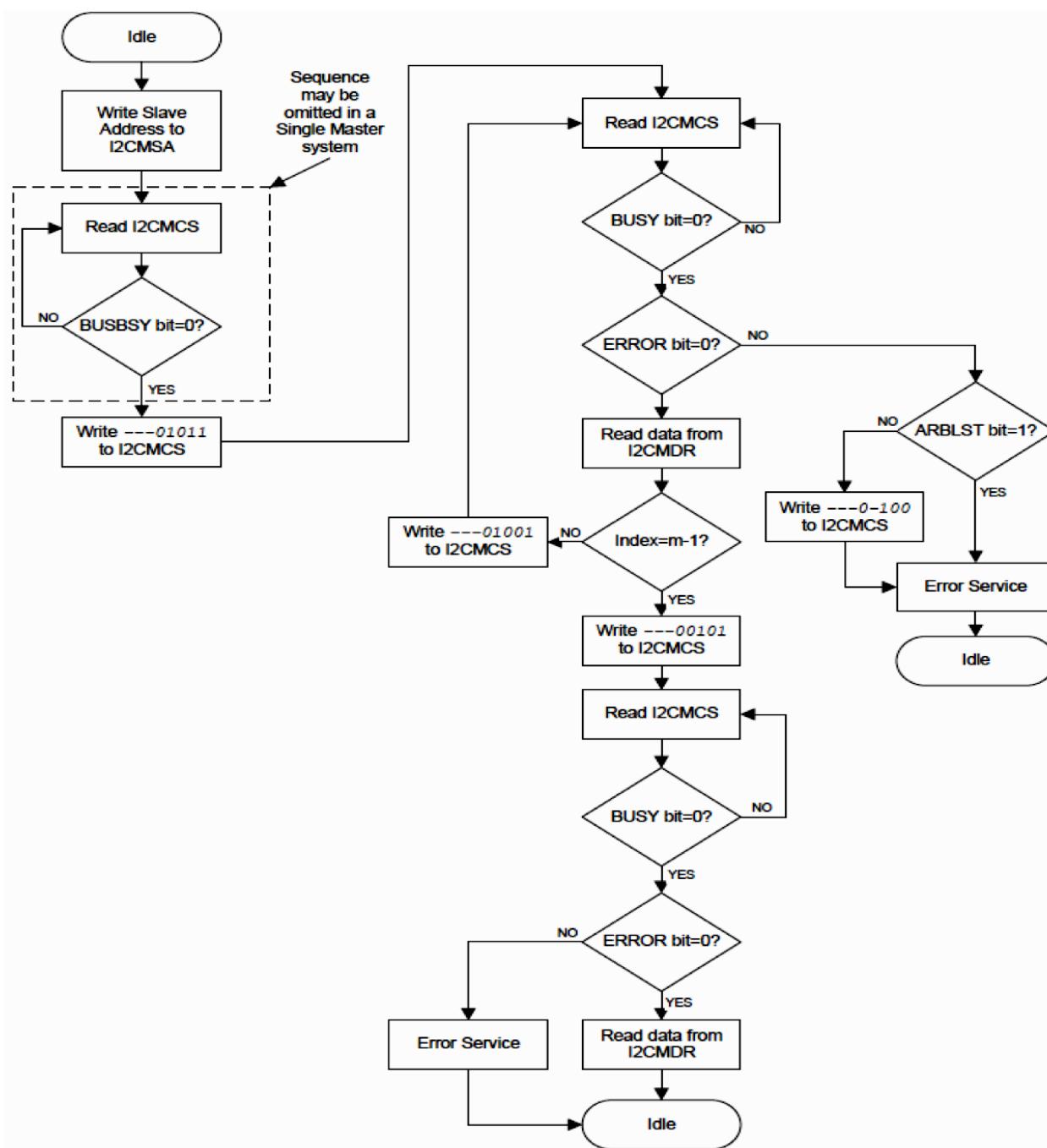


Figure 8.47 The operational sequence of the master working in the receive mode.

#### 8.4.4.3 I2C MODULE WORKS IN SLAVE TRANSMIT & RECEIVE MODES

- When operating in the slave mode, the **STARTRIS** and **STOPRIS** bits in the **I2C Slave Raw Interrupt Status (I2CSRIS)** register indicate detection of start and stop conditions on the bus and the **I2C Slave Masked Interrupt Status (I2CSMIS)** register can be configured to allow **STARTRIS** and **STOPRIS** to be promoted to controller interrupts when interrupts are enabled.
- When a receiver cannot receive another complete byte, it can hold the clock line **SCL** Low and force the transmitter into a wait state. The data transfer continues when the receiver releases the clock **SCL**.
- The operational sequence of the I2C module working in the slave mode is shown in Figure 8.48 (next slide). The operational sequence is:
  1. Write the **slave address** into the **I2CSOAR** to indicate that this is a slave device.
  2. Write **0x1** to the **I2CSCSR** to enable the I2C to work as a slave I2C device.
  3. Check the **I2CSCSR** to identify the operational mode for this slave. If **RREQ** bit in this register is **1**, this means that the slave is working in the receive mode and waiting for the data to be transmitted from the master.



#### 8.4.4.3 I2C MODULE WORKS IN SLAVE TRANSMIT & RECEIVE MODES

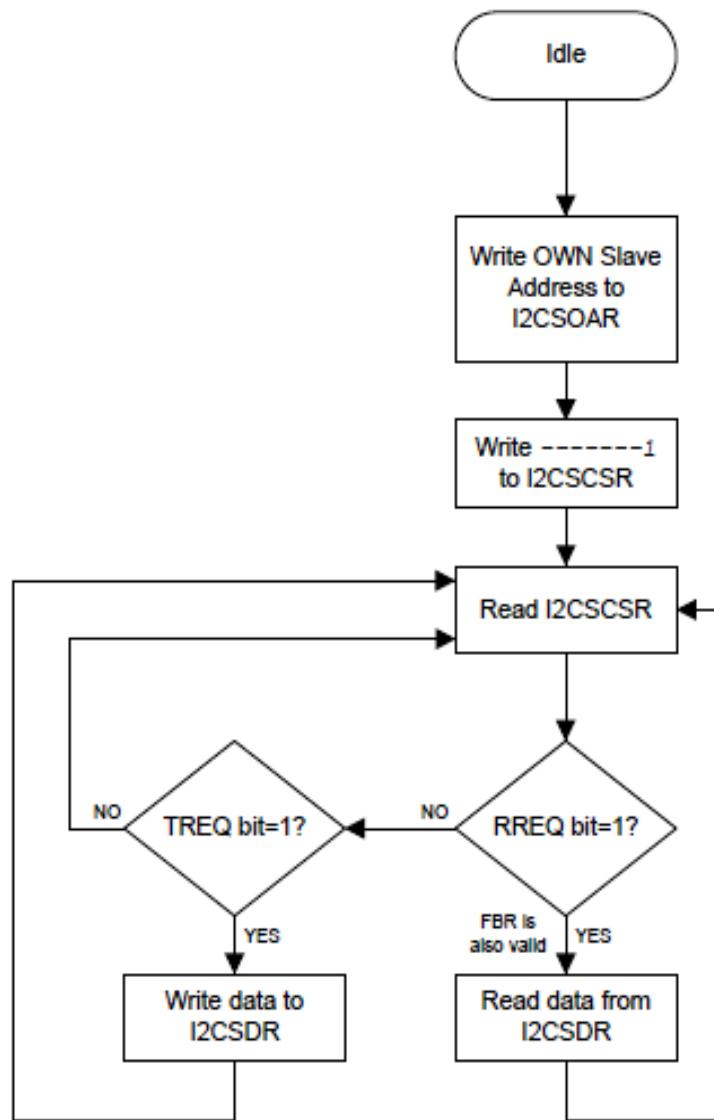


Figure 8.48 The operational sequence of the I2C module working in the slave mode.

#### 8.4.4.3 I2C MODULE WORKS IN SLAVE TRANSMIT & RECEIVE MODES

4. Also check the **FBR** bit in the **I2CSCCSR** to confirm that the first byte following the slave's own address has been received.
  5. Then the received data in the **I2CSDR** is read out by the slave device. The slave continues for the next data receiving process until all data received and a **STOP** status is received from the master by checking the **STOPRIS** bits in the **I2C Slave Raw Interrupt Status (I2CSRIS)** register.
  6. If the **RREQ** bit is **0**, the slave continues to check the **TREQ** bit in the **I2CSCCSR** to confirm whether the slave is working in the transmit mode. If the **TREQ** bit is **1**, it indicates that the slave is working in the data transmit mode. The slave writes the data to be transmitted into the **I2CSDR** to send it out.
  7. Then the slave device continues this data transmit until a **STOP** status is received from the master by checking the **STOPRIS** bits in the **I2C Slave Raw Interrupt Status (I2CSRIS)** register.
- Now we have a clear picture about the I2C operational sequence. Next let's have a closer look at the I2C major components.



## 8.4.5 I2C MODULE MAJOR OPERATIONAL COMPONENTS & CONTROL SIGNALS

- The following operational components are mainly implemented in the I2C module operations:
  - **Acknowledge**
  - All bus transactions have a required acknowledge clock cycle that is generated by the master. During the acknowledge cycle, the transmitter (either master or slave) releases the **SDA** line. To acknowledge the transaction, the receiver must pull down **SDA** during the acknowledge clock cycle.
  - When a slave receiver does not acknowledge the slave address, **SDA** must be left **High** by the slave so that the master can generate a **STOP** condition and abort the current transfer. If the master device is acting as a receiver during a transfer, it is responsible for acknowledging each transfer made by the slave. Because the master controls the number of bytes in the transfer, it signals the end of data to the slave transmitter by not generating an **ACK** on the last data byte. The slave transmitter must then release **SDA** to allow the master to generate the **STOP** or a repeated **START** condition.
  - If the slave is required to provide a manual **ACK** or **NACK**, the **I2C Slave ACK Control (I2CSACKCTL)** register allows the slave to **NACK** for invalid data or command or **ACK** for valid data or command. When this operation is enabled, the MCU slave module I2C clock is pulled low after the last data bit until this register is written with the indicated response.

## 8.4.5 I2C MODULE MAJOR OPERATIONAL COMPONENTS & CONTROL SIGNALS

- ***Repeated Start***
- The I<sub>2</sub>C master module has the capability to execute a repeated **START** (transmit or receive) after an initial transfer has occurred. This means that after a data has been transmitted, the master does not generate a **STOP** condition but instead writes another slave address to the **I2CMSA** register and then writes **0x3** to initiate the repeated **START**.
- ***Clock Low Timeout (CLTO)***
- The I<sub>2</sub>C slave sometimes can extend the transaction by pulling the clock low periodically to create a slow bit transfer rate. The I<sub>2</sub>C module has a 12-bit programmable counter that is used to track how long the clock has been held low. The upper 8 bits of the count value are software programmable through the **I2C Master Clock Low Timeout Count (I2CMCLKOCNT)** register. The lower four bits are **0x0** and are not user visible. The **CNTL** value programmed in the **I2CMCLKOCNT** register has to be greater than **0x01**. The application can program the eight MSBs of the counter to reflect the acceptable cumulative low period in transaction. The count is loaded at the **START** condition and counts down on each falling edge of the internal bus clock of the master. Note that the internal bus clock generated for this counter keeps running at the programmed I<sub>2</sub>C speed even if **SCL** is held low on the bus. Upon reaching terminal count, the master forces **ABORT** on the bus by issuing a **STOP** condition at the instance of **SCL** and **SDA** release.

## 8.4.5 I<sub>2</sub>C MODULE MAJOR OPERATIONAL COMPONENTS & CONTROL SIGNALS

- The **CLKRIS** bit in the **I<sub>2</sub>CMRIS** register is set when the clock timeout period is reached to allow the master to start corrective action to resolve the remote slave state. In addition, the **CLKTO** bit in the **I<sub>2</sub>CMCS** register is set; this bit is cleared when a **STOP** condition is sent or during the I<sub>2</sub>C master reset. The status of the raw **SDA** and **SCL** signals are readable by software through the **SDA** and **SCL** bits in the **I<sub>2</sub>C Master Bus Monitor (I<sub>2</sub>CMBMON)** register to help determine the state of the remote slave.
- In the event of a **CLTO** condition, application software must choose a way to try to recover it. Most applications may attempt to manually toggle the I<sub>2</sub>C pins to force the slave to let go of the clock signal (a common solution is to attempt to force a **STOP** on the bus).
- **Dual Address**
- The I<sub>2</sub>C interface supports dual address capability for the slave. The additional programmable address is provided and can be matched if enabled. In dual address mode, the I<sub>2</sub>C slave provides an **ACK** on the bus if either the **OAR** field in the **I<sub>2</sub>CSOAR** register or the **OAR2** field in the **I<sub>2</sub>CSOAR2** register is matched. The enable for dual address is programmable through the **OAR2EN** bit in the **I<sub>2</sub>CSOAR2** register and there is no disable on the legacy address.
- The **OAR2SEL** bit in the **I<sub>2</sub>CSCSR** register indicates if the address that was **ACKed** is the alternate address or not. When this bit is clear, it indicates either legacy operation or no address match.

## 8.4.5 I2C MODULE MAJOR OPERATIONAL COMPONENTS & CONTROL SIGNALS

- ***Arbitration***
- A master may start a transfer only if the bus is idle. It is possible for two or more masters to generate a **START** condition within minimum hold time of the **START** condition. In these situations, an arbitration scheme takes place on the **SDA** line while **SCL** is **High**. During arbitration, the first of the competing master devices places a **1** (High) on **SDA**, while another master transmits a **0** (Low), switches off its data output stage and retires until the bus is idle again.
- Arbitration can take place over several bits. Its first stage is a comparison of address bits, and if both masters are trying to address the same device, arbitration continues on to the comparison of data bits.
- ***Glitch Suppression in Multi-Master Configuration***
- When a multi-master configuration is used, the **GFE** bit in the **I2CMCR** can be set to enable glitch suppression on the **SCL** and **SDA** lines to assure proper signal values. The filter can be programmed to different filter widths using the **GFPW** bit in the **I2C Master Configuration Register 2 (I2CMCR2)**. The glitch suppression value is based on the buffered system clocks. Note that all signals will be delayed internally when glitch suppression is applied. For example, if **GFPW** is set to **0x7**, **31** clocks should be added onto the calculation for the expected transaction time.



## 8.4.6 I2C MODULE RUNNING SPEEDS (CLOCK RATES) AND INTERRUPTS

- Four I<sup>2</sup>C modules can operate at four speeds: **Standard** (100 kbps), **Fast** (400 kbps), **Fast-Mode Plus** (1 Mbps) and **High-Speed** (3.33 Mbps).
- These speeds can be selected by using the **I<sup>2</sup>C Master Timer Period (I<sup>2</sup>CMTPR)** register.
- The **I<sup>2</sup>C clock rate** is determined by four parameters: **CLK\_PRD**, **TIMER\_PRD**, **SCL\_LP**, and **SCL\_HP**. The definitions for these parameters are:
  - **CLK\_PRD** is the system clock period.
  - **SCL\_LP** is the low period of SCL (fixed at 6).
  - **SCL\_HP** is the high period of SCL (fixed at 4).
  - **TIMER\_PRD** is a programmed-value in the I<sup>2</sup>CMTPR register. This value is determined by using the equation below and solving for TIMER\_PRD.
- The **I<sup>2</sup>C clock period** is calculated as follows:
  - **SCL\_PERIOD = 2 × (1 + TIMER\_PRD) × (SCL\_LP + SCL\_HP) × CLK\_PRD**



## 8.4.6 I2C MODULE RUNNING SPEEDS (CLOCK RATES) AND INTERRUPTS

- For example, if **CLK\_PRD** = 50 ns, **TIMER\_PRD** = 1, **SCL\_LP** = 6, and **SCL\_HP** = 4. This yields a **SCL** period as 2000 ns = 2  $\mu$ s with a frequency of  $1/\text{SCL\_PERIOD} = 500 \text{ KHz}$ .
- Table 8.21 shows some examples of the timer periods (**TIMER\_PRD**) that should be used to generate **Standard**, **Fast mode**, and **Fast mode plus** **SCL** frequencies for various system clocks.

Table 8.21 Examples of I2C master timer period versus speed mode.

System Clock	Period	Standard Mode	Period	Fast Mode	Period	Fast mode Plus
4 MHz	0x01	100 Kbps	-	-	-	-
6 MHz	0x02	100 Kbps	-	-	-	-
12.5 MHz	0x06	89 Kbps	0x01	312 Kbps	-	-
16 MHz	0x07	100 Kbps	0x02	278 Kbps	-	-
20 MHz	0x09	100 Kbps	0x02	333 Kbps	-	-
25 MHz	0x0C	96.2 Kbps	0x03	312 Kbps	-	-
33 MHz	0x10	97.1 Kbps	0x04	330 Kbps	-	-
40 MHz	0x13	100 Kbps	0x04	400 Kbps	0x01	1000 Kbps
50 MHz	0x18	100 Kbps	0x06	357 Kbps	0x02	833 Kbps
80 MHz	0x27	100 Kbps	0x09	400 Kbps	0x03	1000 Kbps



### 8.4.6.1 I2C MODULE HIGH SPEED MODE

- The TM4C123GH6PM I<sub>2</sub>C peripheral provides supports for **High-speed** operation as both a master and a slave.
- High-Speed mode is configured by setting the **HS** bit in the **I<sub>2</sub>C Master Control/Status (I<sub>2</sub>CMCS)** register. High-Speed mode transmits data at a high bit rate with a **33.3%** duty cycle, but communication and arbitration are done at Standard, Fast mode, or Fast-mode plus speed, depending on which is selected by the user.
- When the **HS** bit in the **I<sub>2</sub>CMCS** register is set, current mode is enabled.
- The clock period is calculated using the equation below. But in the High-Speed mode, both **SCL\_LP** and **SCL\_HP** are fixed as **SCL\_LP = 2** and **SCL\_HP = 1**.
  - **SCL\_PERIOD = 2 × (1 + TIMER\_PRD) × (SCL\_LP + SCL\_HP) × CLK\_PRD**
- For example: if **CLK\_PRD = 25 ns** and **TIMER\_PRD = 1**, it yields a **SCL** period as **300 ns** with a frequency of  $1/T = 3.33 \text{ MHz}$ .



#### 8.4.6.1 I2C MODULE HIGH SPEED MODE

- Table 8.22 shows some examples of timer period and system clock in **High-Speed** mode. Note that the **HS** bit in the **I2CMTPR** register needs to be set for the **TPR** value to be used in High-Speed mode.

Table 8.22 Examples of master timer period in High-Speed mode.

System Clock	Period	Transmit Speed
40 MHz	0x01	3.33 Mbps
50 MHz	0x02	2.77 Mbps
80 MHz	0x03	3.33 Mbps

- When working as a master, the master is responsible for sending a **master-code-byte** in either **Standard** (100 Kbps) or **Fast-mode** (400 Kbps) before it begins transferring in **High-speed** mode.
- The **master-code-byte** must contain data in the form of **00001XXX** and is used to tell the slave to prepare for a **High-speed** transfer.
- The **master-code-byte** should never be acknowledged by a slave since it is only used to indicate that the upcoming data is going to be transferred at a higher speed data rate.



#### 8.4.6.1 I2C MODULE HIGH SPEED MODE

- To send the **master-code-byte**, software should place the value of the master-code-byte into the **I2CMCSA** register and write the **I2CMCS** register with a value of **0x13**.
- This places the I2C master peripheral in **High-speed** mode, and all subsequent transfers are carried out at High-speed data rate using the normal **I2CMCS** command bits, without setting the **HS** bit in the **I2CMCS** register.
- Again, setting the **HS** bit in the **I2CMCS** register is only necessary during the master-code-byte.
- When operating as a **High-speed** slave, there is no additional software required.



## 8.4.6.2 I2C MODULE INTERRUPTS GENERATION AND PROCESSING

- The I2C module can generate interrupts when the following conditions are observed:
  - **Master transaction completed.**
  - **Master arbitration lost.**
  - **Master transaction error.**
  - **Master bus timeout.**
  - **Slave transaction received.**
  - **Slave transaction requested.**
  - **Stop condition on bus detected.**
  - **Start condition on bus detected.**
- The I2C **master** and **slave** modules have **separate interrupt signals**.
- While both modules can generate interrupts for multiple conditions, only a **single** interrupt signal can be sent to the interrupt controller.



#### 8.4.6.2.1 I<sub>2</sub>C MASTER INTERRUPTS

- The I<sub>2</sub>C **master** module generates an interrupt when a transaction completes in either transmit or receive mode, when arbitration is lost, or when an error occurs during a transaction.
- To enable the I<sub>2</sub>C master interrupt, software must set the **IM** bit in the **I<sub>2</sub>C Master Interrupt Mask Register (I<sub>2</sub>CMIMR)**.
- When an interrupt condition is met, software must check the **ERROR** and **ARBLST** bits in the **I<sub>2</sub>C Master Control/Status (I<sub>2</sub>CMCS)** register to verify that an error didn't occur during the last transaction and to ensure that arbitration has not been lost. An error condition is asserted if the last transaction was not acknowledged by the slave. If an error is not detected and the master has not lost arbitration, the application can proceed with the transfer.
- The interrupt is cleared by writing a **1** to the **IC** bit in the **I<sub>2</sub>C Master Interrupt Clear Register (I<sub>2</sub>CMICR)**.
- If the application does not need to use interrupts, the raw interrupt status is always visible in the **I<sub>2</sub>C Master Raw Interrupt Status (I<sub>2</sub>CMRIS)** register.

### 8.4.6.2.2 I<sub>2</sub>C SLAVE INTERRUPTS

- The **slave** module can generate an interrupt when data has been received or requested. This interrupt is enabled by setting the **DATAIM** bit in the **I<sub>2</sub>C Slave Interrupt Mask Register (I<sub>2</sub>CSIMR)**.
- As an interrupt occurred, the software determines whether the module should write (transmit) or read (receive) data from the **I<sub>2</sub>C Slave Data Register (I<sub>2</sub>CSDR)**, by checking the **RREQ** and **TREQ** bits of the **I<sub>2</sub>C Slave Control/Status Register (I<sub>2</sub>CSCSR)**. If the slave module is in receive mode and the first byte of a transfer is received, the **FBR** bit is set along with the **RREQ** bit. The interrupt is cleared by setting the **DATAIC** bit in the **I<sub>2</sub>C Slave Interrupt Clear Register (I<sub>2</sub>CSICR)**.
- The slave module can generate an interrupt when a **START** and a **STOP** state is detected. These interrupts are enabled by setting the **STARTIM** and **STOPIM** bits in the **I<sub>2</sub>C Slave Interrupt Mask Register (I<sub>2</sub>CSIMR)** and cleared by writing a **1** to the **STOPIC** and **STARTIC** bits of the **I<sub>2</sub>C Slave Interrupt Clear Register (I<sub>2</sub>CSICR)**.
- If the application does not need to use any interrupt, the raw interrupt status is always visible via the **I<sub>2</sub>C Slave Raw Interrupt Status (I<sub>2</sub>CSRIS)** register.

## 8.4.7 I2C INTERFACE CONTROL SIGNALS AND GPIO I2C CONTROL REGISTERS

- All peripheral devices used in the TM4C123GH6PM MCU system are interfaced to MCU via related GPIO Ports and pins, and this is also true to **I2C** modules.
- Table 8.23 lists the external signals of the I2C interface and related function.

Table 8.23 I2C control signals and GPIO pins distributions

I2C Pin	GPIO Pin	Pin Type	Buffer Type	Pin Function
I2C0SCL	PB2 (3)	I/O	NON-OD	I2C Module 0 Clock.
I2C0SDA	PB3 (3)	I/O	OD	I2C Module 0 Data.
I2C1SCL	PA6 (3)	I/O	NON-OD	I2C Module 1 Clock.
I2C1SDA	PA7 (3)	I/O	OD	I2C Module 1 Data.
I2C2SCL	PE4 (3)	I/O	NON-OD	I2C Module 2 Clock.
I2C2SDA	PE5 (3)	I/O	OD	I2C Module 2 Data.
I2C3SCL	PD0 (3)	I/O	NON-OD	I2C Module 3 Clock.
I2C3SDA	PD1 (3)	I/O	OD	I2C Module 3 Data.

## 8.4.7 I2C INTERFACE CONTROL SIGNALS AND GPIO I2C CONTROL REGISTERS

- The I<sub>2</sub>C interface signals are **alternate functions** for some GPIO pins and default to be GPIO signals at reset, with the exception of the **I2CoSCL** and **I2CSDA** pins which default to the I<sub>2</sub>C function.
- The **AFSEL** bit in the **GPIOAFSEL** register should be set to choose the I<sub>2</sub>C function. The number **(3)** in the parentheses is the encoding that must be programmed into the **PMCx** field in the **GPIOPCTL** register to assign the I<sub>2</sub>C signal to the specified GPIO port pins.
- Note that the **I2CSDA** pin should be set to open drain using the **GPIO Open Drain Select (GPIOODR)** register, but the **I2CSCL** cannot do that setting.

## 8.4.8 I2C MODULE CONTROL REGISTERS AND THEIR FUNCTIONS

- As shown in Figure 8.44, all I<sub>2</sub>C module components and their control functions are globally illustrated by this block diagram. However, in order to get more details about these components and control signals, we need to discuss them one by one with more details based on each group of registers and their functions.

## 8.4.8.1 I2C MODULE MASTER CONTROL REGISTERS

- As we discussed in section 8.4.2, there are **11** popular control registers in the master control register group and we will discuss them one by one in the following sections.

### 8.4.8.1.1 I2C MASTER SLAVE ADDRESS REGISTER (I2CMSA)

- This is a 32-bit register and only **lower 8 bits** are used to contain seven bits **address (A6-A0)** for the selected slave device and a **Receive/Send (R/S)** bit, which determines whether the next operation is a **Receive (1)**, or a **Transmit (0)** (Figure 8.49).

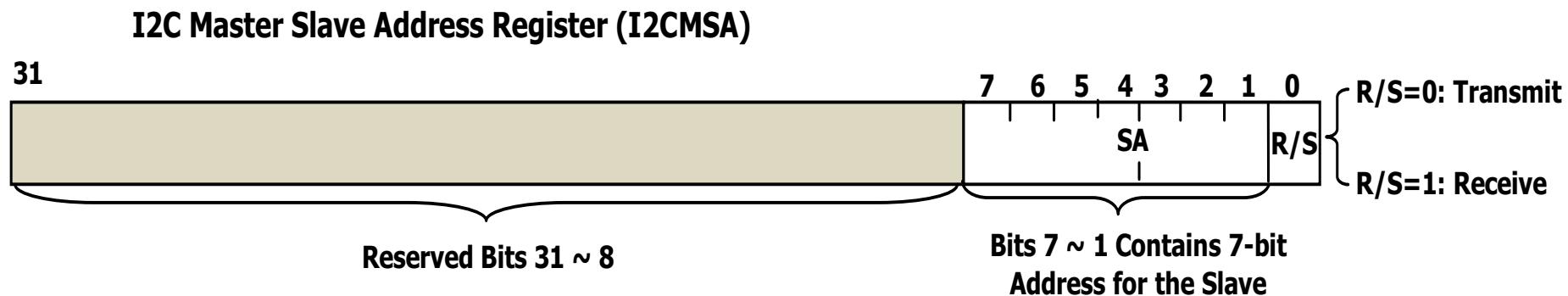


Figure 8.49 The bit field and function for the I2CMSA register.

### 8.4.8.1.2 I2C MASTER CONTROL/STATUS REGISTER (I2CMCS)

- This is a 32-bit register and provides **dual functions** for the I2C module to perform either a **reading** or a **writing** operation.
- When **reading**, this register uses the **lower 8-bit** to indicate the current **running status** of the I2C module. When **writing**, this register uses the **lower 5 bits** to setup the operational parameters for the next data operation.
- Figure 8.50 shows the bit field configurations in the reading status, and Table 8.24 (next slide) shows the bit field values and functions in the **reading** status.

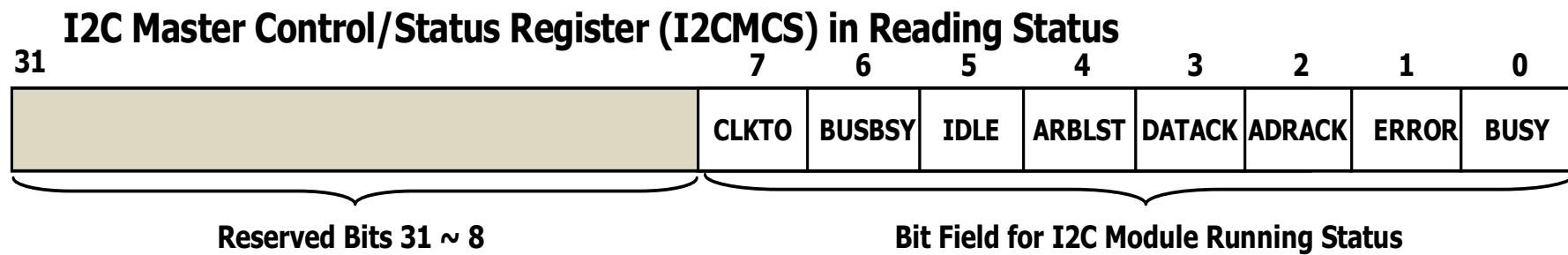


Figure 8.50 The bit configurations in the I2CMCS register (Reading).

## 8.4.8.1.2 I2C MASTER CONTROL/STATUS REGISTER (I2CMCS)

Table 8.24 Bit field value and its function for I2CMCS register (Reading Status).

Bit	Name	Reset	Function
<b>31:8</b>	<b>Reserved</b>	0x0	Reserved
<b>7</b>	<b>CLKTO</b>	0x0	Clock Timeout Error. <b>0:</b> No clock timeout error occurred. <b>1:</b> A clock timeout error has occurred.
<b>6</b>	<b>BUSBSY</b>	0x0	Bus Busy. <b>0:</b> The I2C bus is idle. <b>1:</b> The I2C bus is busy. The bit changes based on the <b>START</b> and <b>STOP</b> conditions.
<b>5</b>	<b>IDLE</b>	0x0	<b>0:</b> The I2C controller is NOT idle. <b>1:</b> The I2C controller is idle
<b>4</b>	<b>ARBLST</b>	0x0	Arbitration Lost. <b>0:</b> The I2C controller won arbitration. <b>1:</b> The I2C controller lost arbitration
<b>3</b>	<b>DATAACK</b>	0x0	Acknowledge Data. <b>0:</b> The transmitted data was acknowledged. <b>1:</b> The transmitted data was NOT acknowledged.
<b>2</b>	<b>ADRACK</b>	0x0	Acknowledge Address. <b>0:</b> The transmitted address was acknowledged. <b>1:</b> The transmitted address was NOT acknowledged.
<b>1</b>	<b>ERROR</b>	0x0	<b>0:</b> No error was detected on the last operation. <b>1:</b> An error occurred on the last operation.
<b>0</b>	<b>BUSY</b>	0x0	<b>0:</b> The I2C controller is idle. <b>1:</b> The I2C controller is busy.



## 8.4.8.1.2 I2C MASTER CONTROL/STATUS REGISTER (I2CMCS)

- Figure 8.51 shows the bit field configurations in the **writing** status, and Table 8.25 shows the bit field values and functions in the **writing** status.

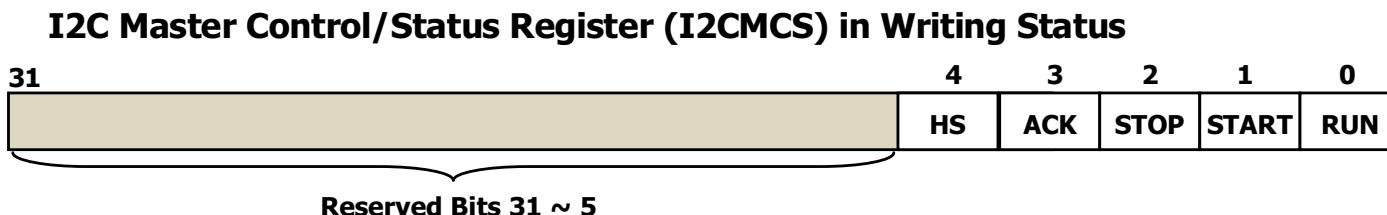


Figure 8.51 The bit configurations in the I2CMCS register (Writing).

Table 8.25 Bit field value and its function for I2CMCS register (Writing Status).

Bit	Name	Reset	Function
31:5	Reserved	0x0	Reserved
4	HS	0x0	High-Speed Enable. <b>0:</b> The master operates in Standard, Fast mode, or Fast mode plus as selected by using a value in the <b>I2CMTPR</b> register that results in an SCL frequency of 100 kbps for Standard mode, 400 kbps for Fast mode, or 1 Mbps for Fast mode plus. <b>1:</b> The master operates in High-Speed mode with transmission speeds up to 3.33 Mbps.
3	ACK	0x0	Data Acknowledge Enable. <b>0:</b> The received data byte is not acknowledged automatically by the master. <b>1:</b> The received data byte is acknowledged automatically by the master.
2	STOP	0x0	<b>0:</b> The controller does not generate the STOP condition. <b>1:</b> The controller generates the STOP condition.
1	START	0x0	<b>0:</b> The controller does not generate the STOP condition. <b>1:</b> The controller generates the STOP condition.
0	RUN	0x0	I2C Master Enable. <b>0:</b> The master is unable to transmit or receive data. <b>1:</b> The master is able to transmit or receive data.

### 8.4.8.1.3 I2C MASTER DATA REGISTER (I2CMDR)

- This is a **32-bit** register with **lowest 8-bit** (byte) containing the data to be transmitted when in the **Master Transmit** state and the data received when in the **Master Receive** state.

### 8.4.8.1.4 I2C MASTER TIMER PERIOD REGISTER (I2CMTPR)

- This register is programmed to set the timer period for the **SCL** clock and assign the **SCL** clock to either **standard** or **high-speed** mode (Figure 8.52).
- The bits **6:0** store a **TPR** value (**TIMER\_PRD**). if **HS = 0**, the calculated **SCL** rate is used for **Standard**, **Fast** or **Fast Plus** mode data operations. If **HS = 1**, the calculated **SCL** rate is used for **High-Speed** data operations. The **TPR** is the **Timer Period** register value with a range of **1** to **127**. It is exactly equal to **TIMER\_PRD** value.

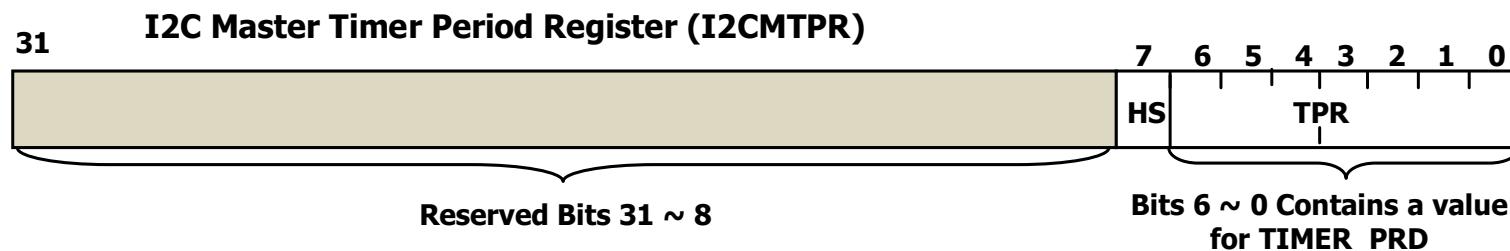


Figure 8.52 The bit configurations in the I2CMTPR register.

### 8.4.8.1.5 I2C MASTER CONFIGURATION REGISTER (I2CMCR)

- This **32-bit** register is used to configure the operational mode (**Master** or **Slave**) for the I2C modules, enable the glitch filter, and set the interface for test mode loopback. Figure 8.53 shows the bit configurations for this register.

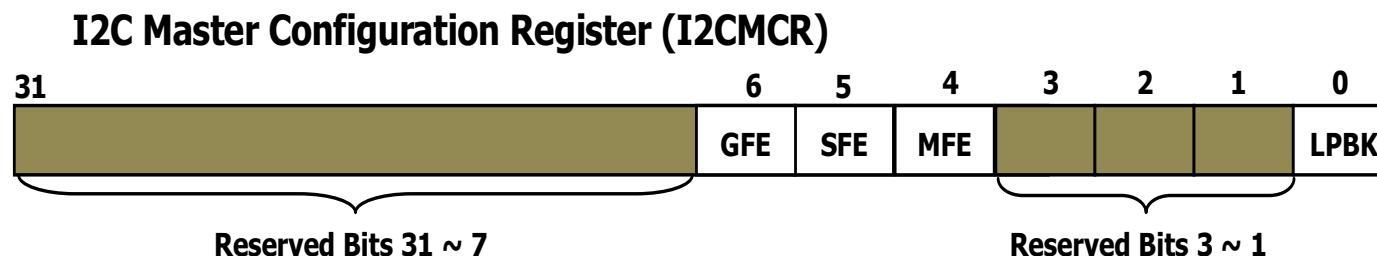


Figure 8.53 The bit configurations for the I2CMCR register.

- If **GFE = 0**, the **glitch filter** is not applied. Otherwise if **GFE = 1**, the glitch filter is enabled. The pulse width of this glitch filter can be configured by programming the **GFPW** bits (**bits 6:4**) in the **I2C Master Configuration 2 Register (I2CMCR2)**.
- If **SFE** and **MFE** is **0**, both slave mode and the master mode is disabled. Otherwise if **SFE = 1** or **MFE = 1**, the slave mode or master mode is enabled and the I2C module works in either mode.
- The bit **0** (**LPBK** bit) is used to set the I2C module in a **loopback mode** to test its function when this bit is set to **1**.

#### 8.4.8.1.6 I2C MASTER CLOCK LOW TIMEOUT COUNT REGISTER (I2CMCLKOCNT)

- This **32-bit** register only uses the **lowest 8 bits** to contain the **upper 8 bits** of a 12-bit counter that can be used to keep the timeout limit for clock stretching by a remote slave. The **lower four bits** of the counter are not user visible and are always **0x0**.
- The **Master Clock Low Timeout counter** counts for the entire time when **SCL** is held **Low** continuously. If **SCL** is returned to **High** at any point, the **Master Clock Low Timeout Counter** is reloaded with the value in this **I2CMCLKOCNT** register and begins counting down from this value.

#### 8.4.8.1.7 I2C MASTER BUS MONITOR REGISTER (I2CMBMON)

- This is a 32-bit register but only the **lowest 2 bits** are used to determine the **SCL** and **SDA** signal status.
- If bit **0 (SCL) = 1**, the **I2CSCL** signal is **High**, otherwise if this bit is **0**, the **I2CSCL** signal is **Low**. Similar situation is true to bit **1 (SDA)** signal.



### 8.4.8.1.8 I2C MASTER INTERRUPT MASK REGISTER (I2CMIMR)

- This is a 32-bit register but only the **lowest 2 bits** are used to control whether a raw interrupt is promoted to a controller interrupt.
- The bit **1 (CLKIM)** is used to control whether a clock timeout interrupt that has been set by the **CLKRIS** bit in the **I2CMRIS** register can be sent to the NVIC and bit **0 (IM)** is used to control whether a master interrupt that has been set by the **RIS** bit in the **I2CMRIS** register can be sent to the interrupt controller.
- If any of these bits is set to **1**, the related interrupt can be sent to the NVIC. Otherwise if any of these bits is **0**, the related interrupt cannot be sent to the NVIC.



### 8.4.8.1.9 I2C MASTER RAW INTERRUPT STATUS REGISTER (I2CMRIS)

- This is a 32-bit register but only the **lowest 2 bits** are used to specify whether a raw interrupt is pending to be processed.
- The bit **1 (CLKRIS)** is used to indicate whether a clock timeout raw interrupt has been generated and pended in the **I2CMRIS** register:
  - If this bit is set to **1**, a timeout raw interrupt is pending.
  - Otherwise there is no timeout interrupt if this bit is **0**.
- Similarly, the bit **0 (RIS)** is to monitor whether a master raw interrupt is pending. If this bit is **1**, a master raw interrupt is pending. Otherwise there is no master interrupt.
- This register specifies whether an interrupt is pending or not.



### 8.4.8.1.10 I2C MASTER MASKED INTERRUPT STATUS REGISTER (I2CMMIS)

- This is a 32-bit register but only the **lowest 2 bits** are used to specify whether an interrupt has been sent to the NVIC to be processed.
- The bit **1 (CLKMIS)** is used to indicate whether a **clock timeout interrupt** has been sent to the interrupt controller to be processed:
  - If this bit is set to **1**, a timeout interrupt is signaled and sent to the NVIC.
  - Otherwise no timeout interrupt has been occurred if this bit is **0**.
- Similarly, the bit **0 (MIS)** is to monitor whether an **I2C interrupt** has been sent to the interrupt controller to be processed:
  - If this bit is **1**, an I2C interrupt is signaled and sent to the NVIC.
  - Otherwise no I2C interrupt has been occurred.
- This register specifies whether an interrupt is signaled and sent to the interrupt controller.



### 8.4.8.1.11 I2C MASTER INTERRUPT CLEAR REGISTER (I2CMICR)

- This is a 32-bit register but only the **lowest 2 bits**, bit **1** (**CLKIC**) and bit **0** (**IC**), are used to clear the **raw** and **masked interrupts**.
- If a **1** is written to the bit **CLKIC**, both the **CLKRIS** bit in the **I2CMRIS** register and the **CLKMIS** bit in the **I2CMMIS** register are cleared to **0**.
- Similarly, if a **1** is written to the bit **IC**, both the **RIS** bit in the **I2CMRIS** register and the **MIS** bit in the **I2CMMIS** register are cleared to **0**.
- Figure 8.54 (next slide) shows these four I2C master interrupt registers' bit functions.



### 8.4.8.1.11 I2C MASTER INTERRUPT CLEAR REGISTER (I2CMICR)

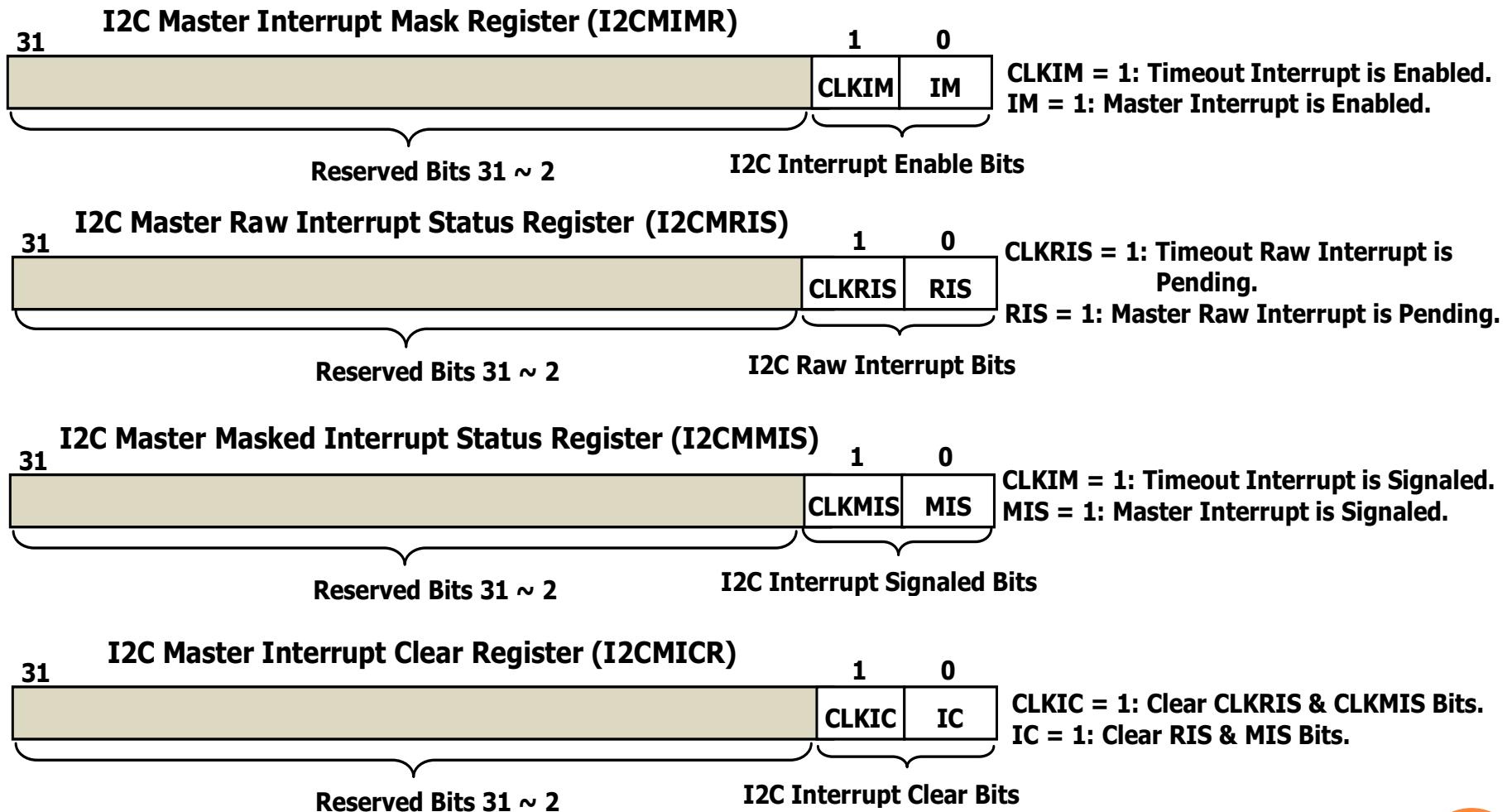


Figure 8.54 Bit functions of four I2C master interrupt registers.

## 8.4.8.2 I2C MODULE SLAVE CONTROL REGISTERS

- As we discussed in section 8.4.2, there are **9** popular control registers in the slave control register group and we will discuss them one by one in the following sections.

### 8.4.8.2.1 I2C SLAVE OWN ADDRESS REGISTERS (I2CSOAR)

- This is a 32-bit register but only the **lowest 7 bits** (bits **6:0**) are used to contain **seven address bits** that identify the TM4C123GH6PM I2C device on the I2C bus.
- Exactly these **7 bits** contain an **I2C Slave Own Address (OAR)** field that specifies bits **A6** through **A0** of the slave address.



### 8.4.8.2.2 I2C SLAVE CONTROL STATUS REGISTER (I2CSCSR)

- Similar to the **I2C Master Control Status Register (I2CMCSR)**, this register also provides **dual** functions for the I2C slave device.
- When **reading**, the **lowest 4 bits** on this register indicate the current **running status** of the slave device. When **writing**, the value of the **LSB** on this register can be used to configure whether the I2C slave operation is enabled or disabled.
- Figures 8.55 shows the bit field configurations of this register in the **reading** and **writing** status. Table 8.26 (next slide) shows the bit field values and functions of this register in the reading status.

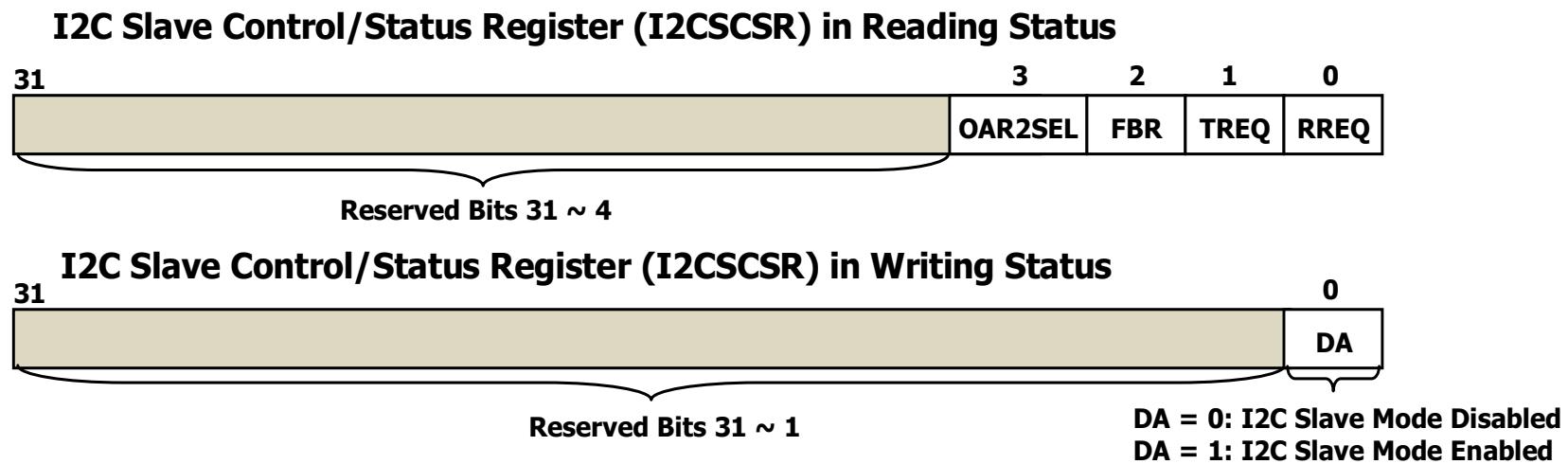


Figure 8.55 Bit field configurations of the I2CSCS register.

## 8.4.8.2.2 I2C SLAVE CONTROL STATUS REGISTER (I2CSCSR)

- When working in the writing status, writing **1** to the **DA** bit in this register is to **enable** the I<sub>2</sub>C slave operation, and writing **0** to this bit is to **disable** the I<sub>2</sub>C slave operation.

Table 8.26 Bit field value and its function for I2CSCSR register (Reading Status).

Bit	Name	Reset	Function
31:4	Reserved	0x0	Reserved
3	OAR2SEL	0x0	OAR2 Address Matched. <b>0:</b> Either the address is not matched or the match is in legacy mode. <b>1:</b> OAR2 address matched and <b>ACKed</b> by the slave.
2	FBR	0x0	First Byte Received. <b>0:</b> The first byte has not been received. <b>1:</b> The first byte following the slave's own address has been received.
1	TREQ	0x0	Transmit Request. <b>0:</b> No transmit request. <b>1:</b> The I <sub>2</sub> C controller has been addressed as a slave transmitter and is using clock stretching to delay the master until data has been written to the <b>I2CSDR</b> register.
0	RREQ	0x0	Receive Request. <b>0:</b> No receive request. <b>1:</b> The I <sub>2</sub> C controller has outstanding receive data from the I <sub>2</sub> C master and is using clock stretching to delay the master until the data has been read from the <b>I2CSDR</b> register.



### 8.4.8.2.3 I2C SLAVE DATA REGISTER (I2CSDR)

- This is a **32-bit** register but only the **lowest 8 bits** are used to contain the data to be transmitted when working in the **Slave Transmit** state, and the data received when working in the **Slave Receive** state.

### 8.4.8.2.4 I2C SLAVE OWN ADDRESS 2 REGISTERS (I2CSOAR2)

- This is a 32-bit register but only the **lowest 8 bits** are used to enable and provide an **alternate address** for the I<sub>2</sub>C data operations.
- The bit **7 (OAR2EN)** is used to enable (**OAR2EN = 1**) or disable (**OAR2EN = 0**) the alternate address.
- The bits **6:0** are used to provide a valid **7-bit alternate address**.
- By using this register, an **alternate address** can be used for the I<sub>2</sub>C data operation if the primary address is not matched or not desired.



### 8.4.8.2.5 I2C SLAVE ACK CONTROL REGISTER (I2CSACKCTL)

- This is a 32-bit register but only the **lowest 2 bits** are used to enable the I<sub>2</sub>C slave to **NACK** for invalid data or command or **ACK** for valid data or command.
- The I<sub>2</sub>C clock is pulled **low** after the last data bit until this register is written. Table 8.27 shows the bit field values and functions of this register.

Table 8.27 Bit field value and its function for I2CSACKCTL register.

Bit	Name	Reset	Function
31:2	Reserved	0x0	Reserved
1	<b>ACKOVAL</b>	0x0	I <sub>2</sub> C Slave ACK Override Value. <b>0:</b> An <b>ACK</b> is sent indicating <b>valid</b> data or command. <b>1:</b> A <b>NACK</b> is sent indicating <b>invalid</b> data or command.
0	<b>ACKOEN</b>	0x0	I <sub>2</sub> C Slave ACK Override Enable. <b>0:</b> A response is not provided. <b>1:</b> An <b>ACK</b> or <b>NACK</b> is sent according to the value written to the <b>ACKOVAL</b> bit.

### 8.4.8.2.6 I2C SLAVE INTERRUPT MASK REGISTER (I2CSIMR)

- This is a 32-bit register but only the **lowest 3 bits** are used to **enable** or **disable** three kinds of slave-related raw interrupts to be sent to the interrupt controller or not. These three bits are:
  - **Bit-2 (STOPIM)**: Set this bit to **1** to enable the **STOP** interrupt, **0** to disable it.
  - **Bit-1 (STARTIM)**: Set this bit to **1** to enable the **START** interrupt, **0** to disable it.
  - **Bit-0 (DATAIM)**: Set this bit to **1** to enable the **DATA** interrupt, and **0** to disable it.
- The precondition of using this register is that the related **STOPRIS**, **STARTRIS** and **DATARIS** bits in the **I2CSRIS** register have been set to indicate that the related raw interrupt is pending.



## 8.4.8.2.7 I2C SLAVE RAW INTERRUPT STATUS REGISTER (I2CSRIS)

- Similar to **I2CSIMR**, only the **lowest 3 bits** are used for this 32-bit register, and they are:
  - **Bit-2 (STOPRIS)**: If a **STOP** condition interrupt occurred, this bit is set to **1**.
  - **Bit-1 (STARTRIS)**: If a **START** raw interrupt occurred, this bit is set to **1**.
  - **Bit-0 (DATARIS)**: If a **DATA** received or a **DATA** requested raw interrupt occurred, this bit is set to **1** to indicate this.

## 8.4.8.2.8 I2C SLAVE MASKED INTERRUPT STATUS REGISTER (I2CSMIS)

- Similar to **I2CSIMR**, only the **lowest 3 bits** are used for this 32-bit register, and they are:
  - **Bit-2 (STOPMIS)**: If a **STOP** condition interrupt is signaled, this bit is set to **1**.
  - **Bit-1 (STARTMIS)**: If a **START** raw interrupt is signaled, this bit is set to **1**.
  - **Bit-0 (DATAMIS)**: If a **DATA** received or a **DATA** requested raw interrupt is signaled, this bit is set to **1** to indicate this.



### 8.4.8.2.9 I<sub>2</sub>C SLAVE INTERRUPT CLEAR REGISTER (I<sub>2</sub>CSICR)

- This register is used to clear any signaled and responded interrupt.
- Similar to **I<sub>2</sub>CSIMR**, only the **lowest 3 bits** are used for this 32-bit register, and they are:
  - **Bit-2 (STOPIC)**: Writing a **1** to this bit clears the **STOPRIS** bit in the **I<sub>2</sub>CSRIS** register and the **STOPMIS** bit in the **I<sub>2</sub>CSMIS** register.
  - **Bit-1 (STARTIC)**: Writing a **1** to this bit clears the **STARTRIS** bit in the **I<sub>2</sub>CSRIS** register and the **STARTMIS** bit in the **I<sub>2</sub>CSMIS** register.
  - **Bit-0 (DATAIC)**: Writing a **1** to this bit clears the **DATARIS** bit in the **I<sub>2</sub>CSRIS** register and the **DATAMIS** bit in the **I<sub>2</sub>CSMIS** register.
- At this point, we complete our discussions about control and status registers used in the I<sub>2</sub>C master and slave operations.
- Next let's take a look at the initialization and configuration of these registers to make them ready to perform desired I<sub>2</sub>C data operations.



## 8.4.9 I2C MODULE INITIALIZATIONS AND CONFIGURATIONS

- Before using the **I2C module** to perform related data operations, all I<sub>2</sub>C modules must be properly initialized and configured.
- Depends on the different operational modes, the initialization and configuration procedures may be different.
- In this section, we use an example to illustrate how to initialize and configure one of the most popular operation modes, **I2C module** to transmit multiple bytes by a master. The system clock is **16 MHz**.
- This configuration process can be divided into two major parts:
  - The configurations to the **I2C-related GPIO Ports and pins**.
  - The configurations to the **I2C module**.



#### 8.4.9.1 INITIALIZATIONS AND CONFIGURATIONS FOR I2C RELATED GPIO PINS

- These initializations and configurations include the following operations:
  1. Enable the clock to the appropriate GPIO module via the **RCGCGPIO** register.
  2. In the GPIO module, enable the appropriate pins for their **alternate function** using the **GPIOAFSEL** register.
  3. Configure the **PMCx** fields in the **GPIOPCTL** register to assign the I2C signals to the appropriate pins.
  4. Enable the **I2CSDA** pin for open-drain operation using the **GPIOODR** register.



## 8.4.9.2 INITIALIZATIONS AND CONFIGURATIONS FOR THE I<sub>2</sub>C MODULE

- These initializations and configurations include the following operations:
    1. Initialize the **I<sub>2</sub>C Master** by writing the **I<sub>2</sub>CMCR** register with a value of **ox0000.0010** to set **MFE** bit (bit-4) to **1** to enable the master function mode.
    2. Set the desired **SCL** clock speed as **100 Kbps** by writing the **I<sub>2</sub>CMTPR** register with the correct value. This value represents the number of system clock periods in one **SCL** clock period. The **TPR** or **TIMER\_PRD** value is determined by the following equation (see Table 8.21 and section 8.4.6):
      - **TPR = (System Clock/(2 × (SCL\_LP + SCL\_HP) × SCL\_CLK)) – 1**, which is
      - **TPR = (16MHz/(2 × (6 + 4) × 100000)) – 1 = 7**
- Write **ox0000.0007** into the **I<sub>2</sub>CMTPR** register.
3. Specify the slave address for the master and followed by a Transmit by writing the **I<sub>2</sub>CMSA** register with **ox0000.0076**. This sets the slave address to **ox3B**.
  4. Place a desired data byte to be transmitted into the **I<sub>2</sub>CMDR** register.
  5. Initiate the first byte transmit of the data from Master to Slave by writing the **I<sub>2</sub>CMCS** register with **ox0000.0003** (**HS:ACK:STOP:START:RUN = 00011**).
  6. Wait until the transmission completes by polling the **I<sub>2</sub>CMCS** register's **BUSBSY** bit until it has been cleared.

## 8.4.9.2 INITIALIZATIONS AND CONFIGURATIONS FOR THE I2C MODULE

7. Check the **ERROR** bit in the **I2CMCS** register to confirm the transmit was acknowledged.
  8. Write the next data to be transmitted into the **I2CMDR** register.
  9. Check whether all data have been transmitted.
  10. If no, write **0x0000.0001** (**HS:ACK:STOP:START:RUN = 00001**) to the **I2CMCS** register to continue transmitting the next data.
  11. If yes, write **0x0000.0101** (**HS:ACK:STOP:START:RUN = 00101**) to the **I2CMCS** register to send **STOP** condition to the slave to inform the slave that the data transmission is done.
  12. Wait until the transmission completes by polling the **I2CMCS** register's **BUSBSY** bit until it has been cleared.
  13. Check the **ERROR** bit in the **I2CMCS** register to confirm that the transmit is acknowledged.
- Now let's build an example project to illustrate how to use the I2C module to perform data transmission job between a master and a slave.

## 8.4.10 BUILD AN EXAMPLE I<sub>2</sub>C MODULE PROJECT

- In the EduBASE ARM® Trainer, a **Real Time Clock (RTC)** module **BQ32000** is installed to work as an I<sub>2</sub>C device.
- In this project, we use this device as an **I<sub>2</sub>C slave** device to ask it to transmit some real time data to the TM4C123GH6PM MCU who works as an **I<sub>2</sub>C master** device, and display these received data on four LEDs **PB3 ~ PBo** via GPIO Port B on the Trainer.
- First let's have a closer look at the structures and functions of the **RTC BQ32000**.



## 8.4.10.1 THE BQ32000 REAL TIME CLOCK (RTC)

- The **BQ32000** is an **8-bit** real time clock device that provides accurate real time for most industrial applications. It contains a 32 KHz oscillator input and two-wire I<sub>2</sub>C interface with IRQ function to allow this RTC to interface to most popular I<sub>2</sub>C devices via I<sub>2</sub>C bus, SDA and SCL.
- The BQ32000 also provides an automatic **backup supply** with integrated trickle charger. The backup supply can be implemented using a capacitor or non-rechargeable battery to greatly save the system power.
- The specific features provided by the RTC include:
  - I<sub>2</sub>C interface supports serial clock up to **400 kHz**.
  - I<sub>2</sub>C slave address **11010000B** for **writing commands** and slave address **11010001B** for **reading commands**.
  - Ten (**10**) **8-bit** normal registers stored the current time, including the seconds, minutes, hours, date and years. Some other items are calibration and configuration parameters. These registers are addressed from **0x00 ~ 0x09**. The **0x00** is for **SECONDS**, **0x01** is for **MINUTES**, and **0x02** is for **CENT\_HOURS** registers.
  - Three (**3**) special function registers stored special function keys are addressed from **0x20 ~ 0x22**.

### 8.4.10.1 THE BQ32000 REAL TIME CLOCK (RTC)

- A functional block diagram of the **RTC BQ32000** is shown in Figure 8.56.

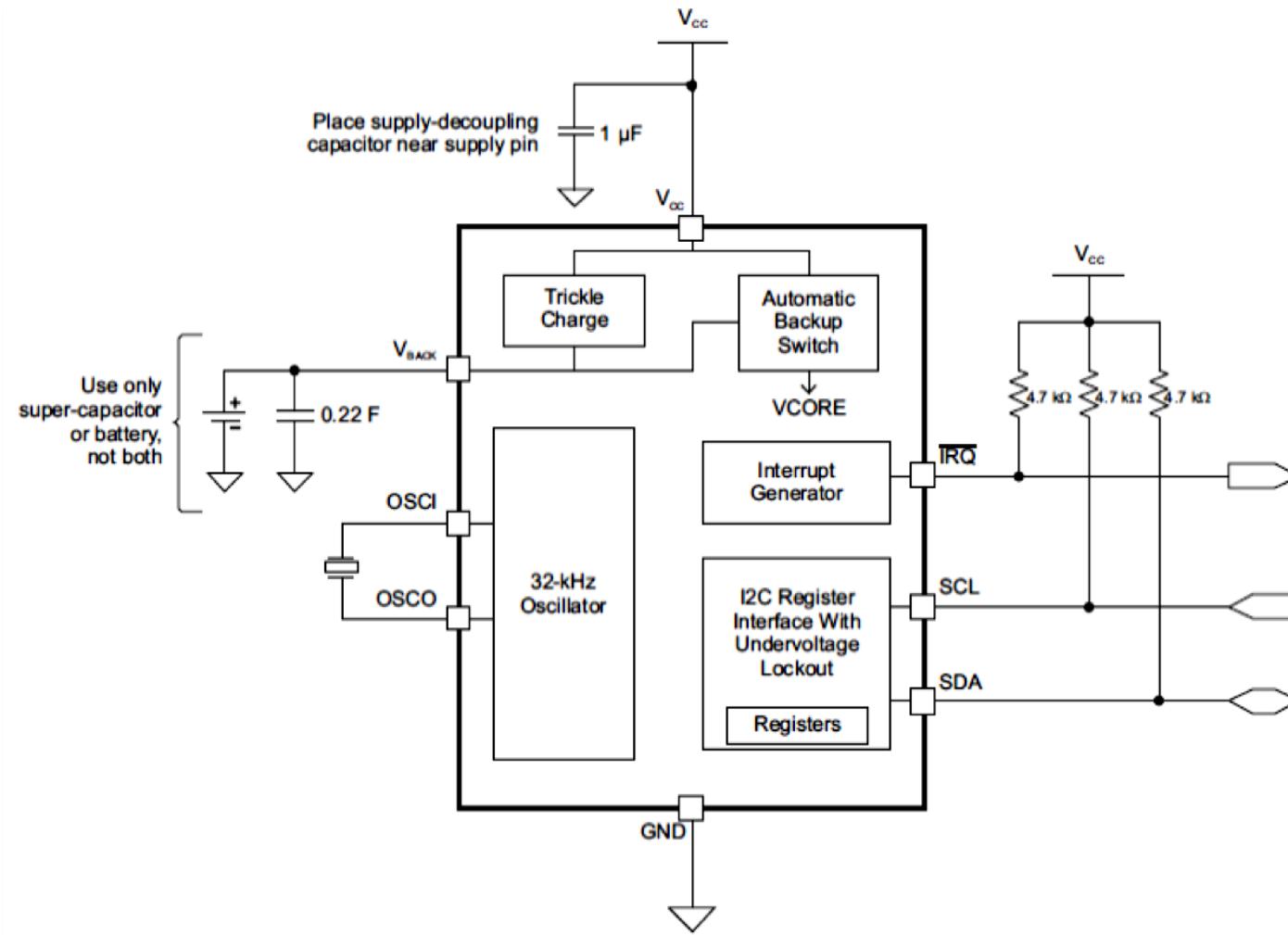


Figure 8.56 The functional block diagram of BQ32000 RTC.

### 8.4.10.1 THE BQ32000 REAL TIME CLOCK (RTC)

- All of these registers can be accessed in terms of their unique addresses.
- Each register, either normal or special, is an 8-bit register. Specially for the normal registers containing the time information, such as seconds, minutes and hours, each of them has seven (7) bits (**D6 ~ D0**) used to store the valid time information, but the **MSB (D7)** is used to store a **function bit** that can be used to perform some function to the RTC.
- For 3 time information registers, **SECONDS**, **MUNITES** and **CENT\_HOURS**, the current time information is formatted in the **BCD (0 ~ 9)** coding. An example structure of this kind of register, **SECONDS**, is shown in Figure 8.57.

D7	D6	D5	D4	D3	D2	D1	D0	BIT(S)
STOP	10_SECOND			1_SECOND				
r/w	r/w			r/w				
0	X	X	X	X	X	X	X	Initial
UC	UC	UC	UC	UC	UC	UC	UC	Cycle

Figure 8.57 The bit field and function of the SECONDS register.

## 8.4.10.1 THE BQ32000 REAL TIME CLOCK (RTC)

- The **STOP** bit **D7** is used to force the oscillator to stop oscillating.
- **STOP** is reset to **0** (normal) on initial application of power, on all subsequent power cycles **STOP** remains unchanged.
- On initial power application **STOP** can be written to **1** (stop) and then written to **0** to force start the oscillator again.
- The **10\_SECOND** bits (**D6 ~ D4**) are the **BCD** representation of the number of tens of seconds on the clock and the valid values are **0** to **5**.
- The **1\_SECOND** bits (**D3 ~ D0**) are also the **BCD** representation of the number of seconds on the clock with the valid values of **0** to **9**.



## 8.4.10.2 THE INTERFACE BETWEEN BQ32000 & EDUBASE ARM® TRAINER

- A functional block diagram of the interface between the EduBASE Trainer and BQ32000 RTC is shown in Figure 8.58.
- The **I2C Module 1** is used to interface to the **BQ32000 RTC** with **PA7** as **SDA1** and **PA6** as **SCL1** wires. Both wires are pulled-up by two resistors in the **RN3** resistor group.

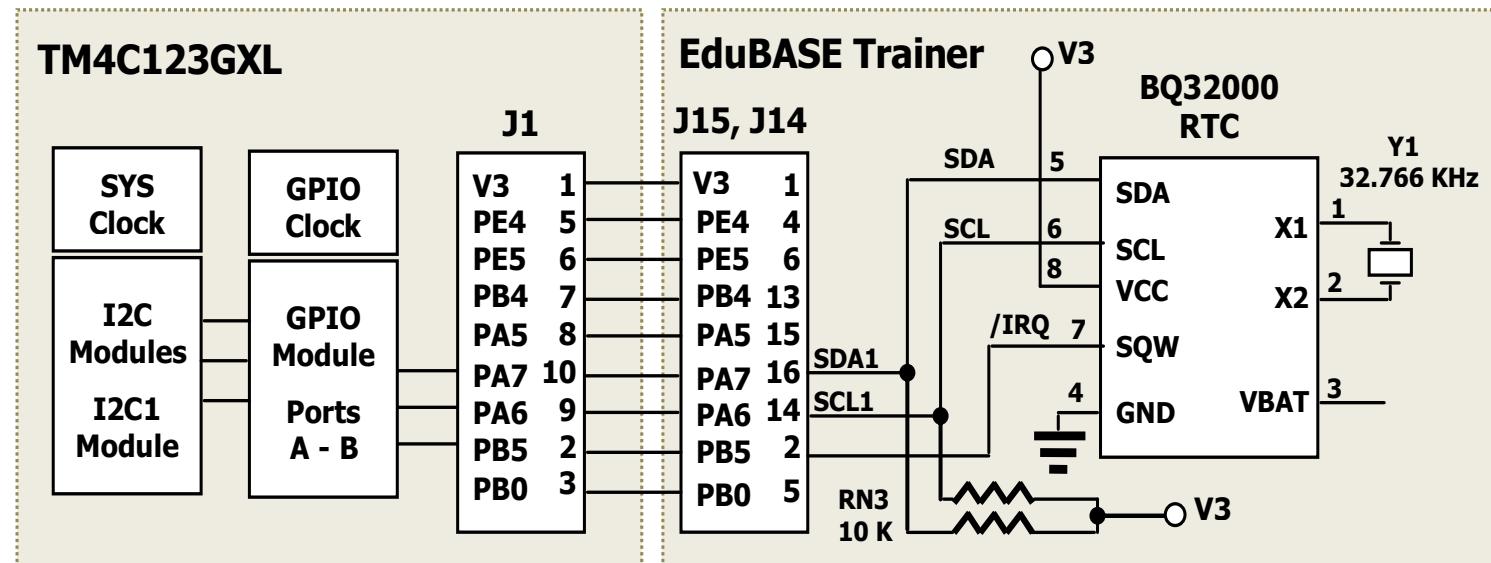


Figure 8.58 The interface between EduBASE Trainer and BQ32000 RTC.

## 8.4.10.2 THE INTERFACE BETWEEN BQ32000 & EDUBASE ARM® TRAINER

- In this example project, we use the TM4C123GH6PM MCU as a master to request data from the slave device **BQ32000**, exactly request the time information from the **SECONDS** register in the BQ32000 RTC via **I<sub>2</sub>C1** module with **SDA1** and **SCL1** wires.
- The acquired time information or seconds are in the **BCD** format and displayed in 4 LEDs, **PB3 ~ PBo** in the EduBASE Trainer via **PB3 ~ PBo** pins.
- To request the data from the slave RTC via **I<sub>2</sub>C1** module, the following operational sequence should be performed:
  1. The I<sub>2</sub>C related GPIO Ports are enabled and related pins should be first initialized and configured with the **GPIOAFSEL**, **GPIOPCTL** and **GPIOODR** registers.
  2. The **I<sub>2</sub>C module 1** related registers should be initialized and configured, these include the **I<sub>2</sub>CMCR** and **I<sub>2</sub>CMTPR** registers.
  3. Write the first nine bits commands, including **7 bits** slave address (**1101000 = 0x68**), **R/S** bit (**W = 0**) and an **ACK** bit to the **I<sub>2</sub>C1\_MSA** register.
  4. Then write the address of the **SECONDS** register (**0x00**) in the **BQ32000** to the **I<sub>2</sub>C1\_MDR** register as a data item to be transmitted.

## 8.4.10.2 THE INTERFACE BETWEEN BQ32000 & EDUBASE ARM® TRAINER

5. Write **0x3 (HS:ACK:STOP:START:RUN = 00011)** to the **I2C1\_MCS** register to start and send this commands to the slave device **BQ32000** via the **SDA** line. The complete command is a combination of steps 3 ~ 5, which is:  
**START:Slave\_Address + Write + ACK :SECONDS\_Register\_Address + ACK.**
6. Check the **I2C1\_MCS** register to make sure that the **ACK** is received with no error and the bus is not busy.
7. Then send a **0** as the first data to the slave by placing it into the **I2C1\_MDR** register. Since we do not want to transmit any data to the slave, instead we want to require data (*seconds* in time) from the **SECONDS** register in the slave RTC, therefore we do not need to send any real data to it.
8. Write **0x5 (HS:ACK:STOP:START:RUN = 00101)** to the **I2C1\_MCS** register to send this data item following with an **ACK** to the slave device **BQ32000** via the **SDA** line. This command indicates that the data transfer is done (**STOP = 1**) after this item is transmitted.
9. Check the **I2C1\_MCS** register to make sure that the **ACK** is received with no error and the bus is not busy.
10. Next we can use an infinitive loop to repeatedly require or read data (*seconds* in time) from the **SECONDS** register in the slave RTC. In order to this, we need to repeat the jobs we did in steps 3 ~ 5 to identify and inform the slave RTC that we want to access the **SECONDS** register to pick up the current seconds in time from this salve device.

## 8.4.10.2 THE INTERFACE BETWEEN BQ32000 & EDUBASE ARM® TRAINER

11. Then write a nine-bit command, including **7 bits** slave address **0x68**, **R/S bit = 1 (Read)** and an **ACK** bit to the **I2C1\_MSA** register to inform the slave RTC that the master needs to require data from the slave. The complete command is:  
**START:Slave\_Address + Read + ACK.**
  12. Write **0x7 (HS:ACK:STOP:START:RUN = 00111)** to the **I2C1\_MCS** register to inform the slave that the master needs to get the current seconds from the slave and stop after this data query operation. This means that we only read a single data item from the slave.
  13. Check the **I2C1\_MCS** register to make sure that the **ACK** is received with no error and the bus is not busy.
  14. Read the received data from the **I2C1\_MDR** register and store it to the desired location.
  15. Check the **I2C1\_MCS** register to make sure that the **ACK** is received with no error and the bus is not busy.
  16. Return to step **10** to read and pick up the next data from the slave RTC.
- Ok, now let's create our example project **DRAI2C**.



### 8.4.10.3 CREATE A DRA MODEL I2C PROJECT DRAI2C

- Perform the following operations to create a new project **DRAI2C**:
  1. Open the Windows Explorer to create a new folder named **DRAI2C** under the **C:\ARM Class Projects\Chapter 8** folder.
  2. Open the Keil® ARM-MDK µVersion5 and go to **Project|New µVersion Project** menu item to create a new µVersion Project. On the opened wizard, browse to our new folder **DRAI2C** that is created in step 1 above. Enter **DRAI2C** into the **File name** box and click on the **Save** button to create this project.
  3. On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series** and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
  4. Next the Software Components wizard is opened, and you need to setup the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, and click on the **OK** button since we need these two components to build our project.
- Since this project is relative simple, therefore only a C source file is good enough.

#### 8.4.10.4 CREATE THE SOURCE FILE DRAI2C

- Create a new C source file **DRAI2C.c** and enter the first part codes shown in Figure 8.59 into this file.

```
1 //*****  
2 // DRAI2C.c - Main Application File for I2C Module 1 to BQ32000 (First Part Codes)  
3 //*****  
4 #include <stdint.h>  
5 #include <stdbool.h>  
6 #include "tm4c123gh6pm.h"  
7 void delay_ms(int n);  
8 void I2C1_Init(void);  
9 uint8_t I2C1_Write(int sAddr, uint8_t rAddr, uint8_t data);  
10 void I2C1_Read(int sAddr, uint8_t rAddr);  
11 #define SLAVE_ADDR 0x68           // 1101 000.  
12 int main(void)  
13 {  
14     uint8_t error;  
15     SYSCTL_RCGCGPIO_R |= 0x2|0x20;    // enable clock to GPIOB & GPIOF  
16     error = SYSCTL_RCGCGPIO_R;  
17     GPIO_PORTB_DIR_R |= 0x0F;          // set PORTB 3-0 as output pins  
18     GPIO_PORTB_DEN_R |= 0x0F;          // set PORTB 3-0 as digital pins  
19     GPIO_PORTF_DIR_R |= 0x0F;          // set PORTF 3-0 as output pins  
20     GPIO_PORTF_DEN_R |= 0x0F;          // set PORTF 3-0 as digital pins  
21     I2C1_Init();  
22     error = I2C1_Write(SLAVE_ADDR, 0, 0);  
23     if (error) {GPIO_PORTF_DATA_R = 0x0F;}  
24     I2C1_Read(SLAVE_ADDR, 0);  
25 }
```

Figure 8.59 The first part codes for the C source file DRAI2C.c.

## 8.4.10.4 CREATE THE SOURCE FILE DRAI2C

```
1 //*****
2 // DRAI2C.c - Main Application File for I2C Module 1 to BQ32000 (Second Part Codes)
3 //*****
4
5 void I2C1_Init(void)
6 {
7     SYSCTL_RCGCI2C_R |= 0x02;           // enable clock to I2C1
8     SYSCTL_RCGCGPIO_R |= 0x1;          // enable clock to GPIOA
9     GPIO_PORTA_AFSEL_R |= 0xC0;        // PA7 for SDA1, PA6 for SCL1
10    GPIO_PORTA_PCTL_R |= 0x33000000;   // PA7 & PA6 as digital pins
11    GPIO_PORTA_DEN_R |= 0xC0;          // PA7 as open drain
12    GPIO_PORTA_ODR_R |= 0x80;          // I2C1 works as master mode
13    I2C1_MCR_R = 0x10;                // SCL1 = 100 kHz at system clock = 16 MHz
14    I2C1_MTPR_R = 0x7;
15 }
16
17 static int I2C1_Wait_Done(void)
18 {
19     while(I2C1_MCS_R & 1);           // wait until I2C1 master is not busy
20     return I2C1_MCS_R & 0xE;         // return I2C1 error code
21 }
22
23 uint8_t I2C1_Write(int sAddr, uint8_t rAddr, uint8_t data) // Write one byte
24 {
25     uint8_t error;
26
27     I2C1_MSA_R = sAddr << 1;        // write: S:(sAddr+W)+ACK+rAddr+ACK+data+ACK+P
28     I2C1_MDR_R = rAddr;              // S:(sAddr+W)+ACK+rAddr+ACK
29     I2C1_MCS_R = 0x3;                // +data+ACK+P
30
31     error = I2C1_Wait_Done();        // wait until write is complete
32     if (error) { return error; }
33     I2C1_MDR_R = data;
34     I2C1_MCS_R = 0x5;                // +data+ACK+P
35     error = I2C1_Wait_Done();        // wait until write is complete
36     while(I2C1_MCS_R & 0x40);       // wait until bus is not busy
37     error = I2C1_MCS_R & 0xE;         // check if any error occurred
38
39     return error;
40 }
```

Figure 8.59 The second part codes for the C source file DRAI2C.c.

#### 8.4.10.4 CREATE THE SOURCE FILE DRAI2C

```
1 //*****  
2 // DRAI2C.c - Main Application File for I2C Module 1 to BQ32000 (Third Part Codes)  
3 //*****  
4 void I2C1_Read(int sAddr, uint8_t rAddr) // Read multiple bytes  
5 {  
6     uint8_t error, data;  
7     while (1)  
8     {  
9         I2C1_MSA_R = sAddr << 1;      // read: S:(sAddr+W)+ACK+rAddr+ACK+R+(sAddr+R)+ACK+data+NACK+P  
10        I2C1_MDR_R = rAddr;  
11        I2C1_MCS_R = 0x3;            // S:(sAddr+W)+ACK+rAddr+ACK  
12        error = I2C1_Wait_Done();  
13        if (error) { GPIO_PORTF_DATA_R |= 0x0F; }  
14        I2C1_MSA_R = (sAddr << 1) + 1;          // restart: +R+(sAddr+R)+ACK  
15        I2C1_MCS_R = 0x7;                  // +S+(sAddr+R)+NACK+P  
16        error = I2C1_Wait_Done();  
17        if (error) { GPIO_PORTF_DATA_R |= 0x0F; }  
18        data = I2C1_MDR_R;  
19        while(I2C1_MCS_R & 0x40);           // wait until bus is not busy  
20        error = I2C1_MCS_R & 0xE;  
21        if (error) { GPIO_PORTF_DATA_R |= 0x0F; }  
22        GPIO_PORTB_DATA_R = data;          // display received data on LEDs  
23        delay_ms(500);  
24    }  
25 }  
26 void delay_ms(int time)                // delay n milliseconds (16 MHz CPU clock)  
27 {  
28     int i, j;  
29     for(i = 0 ; i < time; i++)  
30         for(j = 0; j < 3180; j++){}  
}
```

Figure 8.60 The third part codes for the C source file DRAI2C.c.

#### 8.4.10.4 CREATE THE SOURCE FILE DRAI2C

- Some key points for this file are:
  - In line 34, the I<sub>2</sub>C module 1 is configured to work as a master device by setting bit **4 (MFE)** in the **I2CMCR** register (refer to Figure 8.53).
  - The **TPR** value in the **I2CMTPR** is set to **0x7** to generate a **100 Kbps** bit rate for the SCL signal (refer to section 8.4.6 and Table 8.21) when system clock is **16 MHz** (**SCL\_PERD** =  $2 \times (1 + TPR) \times (6 + 4) \times CLK\_PERD = 2 \times 8 \times 10 \times (1/16 \times 10^6) = 10^{-5}$ , **SCL\_FREQ** =  $10^5$  Hz = 100 Kbps).
  - In line 45, the slave address (**0x68**) is shifted left by **1** bit to make it as a slave writing command address **11010000B** (refer to section 8.4.10.1) and assigned to the **I2C1\_MSA** register.
  - Then the address of the **SECONDS** register in the BQ32000, which is **rAddr** with a value of **0x00**, is assigned to the **I2C1\_MDR** register as the first data to be sent to the slave.
  - In line 47, the **0x3 (HS:ACK:STOP:START:RUN) = 00011** is written into the **I2C1\_MCS** register and sent to the slave to ask the latter to respond with ACK signal.
  - Then the first data to be transmitted, which is **0**, is written into the **I2C1\_MDR** register in line 50, and a command **0x5 (HS:ACK:STOP:START:RUN) = 00101** is written to the **I2C1\_MCS** register in line 51 to ask the slave to stop any operation after this operation.

#### 8.4.10.4 CREATE THE SOURCE FILE DRAI2C

- Some key points for this file are:

- In the **I2C1\_Read()** function, an infinitive **while()** loop is used to repeatedly get the data from the slave RTC in line 7.
- As we did in the **I2C1\_Write()** subroutine, the codes in lines 9 ~ 13 are used to send commands to the slave to start a data operation. These commands include the slave address, the SECONDS register's address and a R/S bit (= 0).
- In line 14, the slave address is shifted left by one **1** bit and a **1** is added into the **LSB** of this address to get a **slave reading command address 11010001B** (refer to section 8.4.10.1) and assigned to the **I2C1\_MSA** register.
- The parameter **0x7 (HS:ACK:STOP:START:RUN) = 00111** is written to the **I2C1\_MCS** register in line 15 to ask the slave to read the data from the **SECONDS** register and send it back the master. This command is executed by one time (start, run and stop this operation).
- The codes in lines 16 and 17 are used to wait for this transmission to be done and return any error if any of them is occurred.
- The requested data obtained from the slave is stored to the local variable **data** in line 18.

## 8.4.10.5 SETUP ENVIRONMENT TO BUILD AND RUN THE PROJECT

- Before you can build and run the project, make sure that the following two issues have been setup correctly:
  - The path of the system header file **tm4c123gh6pm.h** has been included into the project **Include Paths**, which is in the **C/C++** tab under the menu item **Project|Options for Target ‘Target 1’**. The path is:  
**C:\ti\TivaWare\_C\_Series-2.0.1.11577\inc.**
  - The debug adapter used is **Stellaris ICDI**. This adapter can be configured in the **Debug** tab under the menu item **Project|Options for Target ‘Target 1’**.
- Now go to the **Project|Build target** menu item to build the project. Then go to the menu item **Flash|Download** to download the project image file into the flash memory. Run the project by going to **Debug|Start/Stop Debug Session** menu item.
- As the project runs, you can find that the 4-bit LEDs installed on the EduBASE ARM® Trainer are periodically displaying BCD codes staring from **0** and ending at **9**, second by second.



## 8.4.11 I2C API FUNCTIONS PROVIDED BY TIVAWARE PERIPHERAL DRIVER LIBRARY

- In this section, we will introduce another way to access and interface the I<sub>2</sub>C modules to perform desired I<sub>2</sub>C serial communication tasks via I<sub>2</sub>C modules.
- The TivaWare™ Peripheral Driver Library provides more than **50** API functions to interface to I<sub>2</sub>C modules. However, in this section, we concentrate on some most important and popular master functions to illustrate how to use them to build sophisticated project to efficiently access and interface to I<sub>2</sub>C modules to fulfill desired serial data communication tasks.
- These I<sub>2</sub>C API functions can be categorized into the following groups:
  - **Status and initialization** API functions for the I<sub>2</sub>C modules are:  
**I2CMasterInitExpClk()**, **I2CMasterEnable()**, **I2CMasterDisable()**,  
**I2CMasterBusBusy()**, **I2CMasterBusy()**, **I2CMasterErr()**.
  - **Sending and receiving data** from the I<sub>2</sub>C modules are handled by:  
**I2CMasterSlaveAddrSet()**, **I2CMasterControl()**, **I2CMasterDataGet()**,  
**I2CMasterDataPut()**.
- We only concentrate on the master device operations in this part.



## 8.4.11.1 MASTER OPERATIONS

- When using these APIs to drive the I<sub>2</sub>C master module, the user must perform the I<sub>2</sub>C module configurations and data operations in the following sequence:
  1. First initialize the I<sub>2</sub>C master module with a call to **I2CMasterInitExpClk()**. That function sets the bus speed and enables the master module.
  2. After that, the user may transmit or receive data via I<sub>2</sub>C master module. Data is transferred by first setting the slave address using **I2CMasterSlaveAddrSet()**. This function is also used to define whether the transfer is a send (a write to the slave from the master) or a receive (a read from the slave by the master).
  3. Then, if connected to an I<sub>2</sub>C bus that has multiple masters, the I<sub>2</sub>C master must first call **I2CMasterBusBusy()** before attempting to initiate the desired transaction.
  4. After determining that the bus is not busy, if trying to send data, the user must call the **I2CMasterDataPut()** function.
  5. The transaction can then be initiated on the bus by calling the **I2CMasterControl()** function with any of the following commands:
    - **I2C\_MASTER\_CMD\_SINGLE\_SEND**
    - **I2C\_MASTER\_CMD\_SINGLE\_RECEIVE**
    - **I2C\_MASTER\_CMD\_BURST\_SEND\_START**
    - **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_START**

### 8.4.11.1 MASTER OPERATIONS

6. For the single send and receive, the polling method involves looping on the return from **I2CMasterBusy()**. Once that function indicates that the I2C master is not busy, the bus transaction has been completed and can be checked for errors using **I2CMasterErr()**.
  7. If there are no errors, then the data has been sent or is ready to be read using **I2CMasterDataGet()**.
  8. For the burst send and receive cases, the polling method also involves calling the **I2CMasterControl()** function for each byte transmitted or received (using **I2C\_MASTER\_CMD\_BURST\_SEND\_CONT** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_CONT** commands), and for the last byte sent or received (using **I2C\_MASTER\_CMD\_BURST\_SEND\_FINISH** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_FINISH** commands).
  9. If any error is detected during the burst transfer, the **I2CMasterControl()** function should be called using the appropriate stop command (**I2C\_MASTER\_CMD\_BURST\_SEND\_ERROR\_STOP** or **I2C\_MASTER\_CMD\_BURST\_RECEIVE\_ERROR\_STOP**).
  10. For the interrupt-driven transaction, the user must register an interrupt handler for the I2C devices and enable the I2C master interrupt; the interrupt occurs when the master is no longer busy.
- Now let's have a closer look at these API functions.

## 8.4.11.2 I2C MODULE STATUS AND INITIALIZATION API FUNCTIONS

- Table 8.28 (next slide) shows some popular and important I2C module status and configuration API functions.
- The API function **I2CMasterInitExpClk()** is a very important and useful function used to setup and configure the entire I2C module.
- Only after the I2C module is correctly configured by using this function, it can work properly to provide the normal functions.
- Before performing any data operations, the I2C master and master bus must be checked to confirm that both are in the **idle** status by using API functions **I2CMasterBusBusy()** and **I2CMasterBusy()**.
- The **I2CMasterEnable()** function must be called before the master can perform any data operation.
- The error status can be checked using the **I2CMasterErr()** function.



## 8.4.11.2 I2C MODULE STATUS AND INITIALIZATION API FUNCTIONS

Table 8.28 The I2C module status and configuration functions.

API Function	Parameter	Description
<pre>void <b>I2CMasterInitExpClk(</b>     uint32_t ui32Base,     uint32_t ui32I2CClk,     bool bFast) <b>)</b></pre>	<p><b>ui32Base</b> is the base address of the I2C Master module.  <b>ui32I2CClk</b> is the rate of the clock for the I2C module.  <b>bFast</b> set up for fast data transfers.</p>	<p>Initialize operation of the I2C Master block by configuring the bus speed for the master and enabling the I2C Master block. If the parameter <b>bFast</b> is <b>true</b>, then the master block is set up to transfer data at 400 Kbps; otherwise, it is set up to transfer data at 100 Kbps. If Fast Mode Plus (1 Mbps) is desired, software should manually write the I2CMTPR after calling this function. For High Speed (3.4 Mbps) mode, a specific command is used to switch to the faster clocks after the initial communication with the slave is done at either 100 Kbps or 400 Kbps.</p>
<pre>void <b>I2CMasterEnable(</b>     uint32_t ui32Base) <b>)</b></pre>	<p><b>ui32Base</b> is the base address of the I2C Master module.</p>	<p>This function enables operation of the I2C Master block.</p>
<pre>bool <b>I2CMasterBusBusy(</b>     uint32_t ui32Base) <b>)</b></pre>	<p><b>ui32Base</b> is the base address of the I2C Master module.</p>	<p>Indicate whether or not the I2C bus is busy.  Returns <b>true</b> if the I2C bus is busy; otherwise, returns <b>false</b></p>
<pre>bool <b>I2CMasterBusy(</b>     uint32_t ui32Base) <b>)</b></pre>	<p><b>ui32Base</b> is the base address of the I2C Master module.</p>	<p>Indicate whether or not the I2C Master is busy.  Returns <b>true</b> if the I2C Master is busy; otherwise, returns <b>false</b></p>
<pre>uint32_t <b>I2CMasterErr(</b>     uint32_t ui32Base) <b>)</b></pre>	<p><b>ui32Base</b> is the base address of the I2C Master module.</p>	<p>Get the error status of the I2C Master module.  Returns the error status, as one of <b>I2C_MASTER_ERR_NONE</b>, <b>I2C_MASTER_ERR_ADDR_ACK</b>, <b>I2C_MASTER_ERR_DATA_ACK</b>, or <b>I2C_MASTER_ERR_ARB_LOST</b>.</p>

### 8.4.11.3 I2C MODULE SENDING AND RECEIVING DATA API FUNCTIONS

- Table 8.29 (next slide) shows some popular and important I2C module sending and receiving data API functions.
- The **I2CMasterSlaveAddSet()** is an important function used to initiate a data operation, either transmit or receive between the master and the slave.
- The second argument is a 7-bit slave address **ui8SlaveAddr**.
- The third argument **bReceive** is a Boolean value used to indicate the address type for the slave device:
  - A **true** means that the slave address **ui8SlaveAddr** is a reading address and the master initiated a reading operation from the slave.
  - A **false** means that the master is issuing a writing or transmitting operation to the slave device with the slave writing address.



### 8.4.11.3 I2C MODULE SENDING AND RECEIVING DATA API FUNCTIONS

Table 8.29 The I2C module sending and receiving data functions.

API Function	Parameter	Description
void <b>I2CMasterSlaveAddrSet(</b> uint32_t ui32Base, uint8_t ui8SlaveAddr, bool bReceive) <b>)</b>	<b>ui32Base</b> is the base address of the I2C Master module. <b>ui8SlaveAddr</b> 7-bit slave address. <b>bReceive</b> flag for the type of communication with the slave.	Configure the address that the I2C Master places on the bus when initiating a transaction. When the <b>bReceive</b> parameter is set to <b>true</b> , the address indicates that the I2C Master is initiating a read from the slave; otherwise the address indicates that the I2C Master is initiating a write to the slave.
void <b>I2CMasterControl(</b> uint32_t ui32Base, uint32_t ui32Cmd) <b>)</b>	<b>ui32Base</b> is the base address of the I2C Master module. <b>ui32Cmd</b> is the command to be issued to the I2C Master module.	Control the state of the Master module send and receive data. The ui8Cmd parameter can be one of the following values: <b>I2C_MASTER_CMD_SINGLE_SEND</b> <b>I2C_MASTER_CMD_SINGLE_RECEIVE</b> <b>I2C_MASTER_CMD_BURST_SEND_START</b> <b>I2C_MASTER_CMD_BURST_SEND_CONT</b> <b>I2C_MASTER_CMD_BURST_SEND_FINISH</b> <b>I2C_MASTER_CMD_BURST_SEND_ERROR_STOP</b> <b>I2C_MASTER_CMD_BURST_RECEIVE_START</b> <b>I2C_MASTER_CMD_BURST_RECEIVE_CONT</b> <b>I2C_MASTER_CMD_BURST_RECEIVE_FINISH</b> <b>I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP</b> <b>I2C_MASTER_CMD_QUICK_COMMAND</b> <b>I2C_MASTER_CMD_HS_MASTER_CODE_SEND</b>
void <b>I2CMasterDataPut(</b> uint32_t ui32Base, uint8_t ui8Data) <b>)</b>	<b>ui32Base</b> is the base address of the I2C Master module. <b>ui8Data</b> is the data to be transmitted from the I2C Master	Place the supplied data into I2C Master Data Register.
uint32_t <b>I2CMasterDataGet(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the I2C Master module.	Read a byte of data from the I2C Master Data Register. Returns the byte received from by the I2C Master, cast as an uint32_t.

### 8.4.11.3 I2C MODULE SENDING AND RECEIVING DATA API FUNCTIONS

- The **I2CMasterControl()** is a multi-purpose API function with more powerful control abilities for the master data operations.
- The second argument of this function **ui32Cmd** is a command parameter used to indicate what kind of control function should be applied to the master device.
- This command can be any one from a collection of commands, exactly from a collection of command macros listed in Table 8.29.
- For a single data operation, it should be prefixed by **I2C\_MASTER\_CMD\_SINGLE**, and for burst operations, it can be **I2C\_MASTER\_CMD\_BURST**.
- An I2C project built with these I2C Module API functions can be found from the Lab project **Lab8\_5**, which is a loopback I2C master and slave testing project modified based on an example project provided by TI. This lab belongs to a part of the home works for the readers.



## 8.5 UNIVERSAL ASYNCHRONOUS RECEIVERS/TRANSMITTERS (UARTs)

- The **Universal Asynchronous Receivers and Transmitters (UART)** provide asynchronous serial data communications for MCU and UART compatible devices.
- The UART is very similar to **Synchronous Serial Interface (SSI)** in working principle and structure, but the only difference is that the former belongs to the **asynchronous** communications and the latter is the **synchronous** serial communication interface.
- The UART performs the **parallel-to-serial** functions in the **transmitting** mode and **serial-to-parallel** conversions in the **receiving** mode.
- It is very similar in functionality to a popular UART, but is not register-compatible with any UART.



## 8.5 UNIVERSAL ASYNCHRONOUS RECEIVERS/TRANSMITTERS (UARTs)

- Some of important features of the Tiva™ UART include:
  - A **16x12** bit receive **FIFO** ( $16 \times 8$  bit data with 4 control-bit) and a **16x8** bit transmit FIFO.
  - Programmable baud rate generator.
  - Automatic generation and stripping of **Start**, **Stop**, and **Parity** bits.
  - Line break generation and detection.
  - Programmable serial interface with
    - **5, 6, 7, or 8** data bits.
    - **Even, Odd, Stick**, or no **parity** bit generation and detection.
    - **1 or 2** stop bits generation.
    - Baud rate generation, from **0** to processor **clock/16**.
  - Modem control/flow control.
  - **IrDA serial-IR (SIR)** encoder/decoder.
  - DMA interface and 9-bit operation.
- Let's take a look at the asynchronous serial data communication protocols.

## 8.5.1 ASYNCHRONOUS SERIAL COMMUNICATION PROTOCOLS AND DATA FRAMING

- We have provided detailed discussions about the asynchronous serial data communication protocols and data framing in section 8.3.1.
- In asynchronous serial data communications, the transmitter and receiver use its own clock as the timing base, and therefore the pre-defined data communication protocols and data framing are necessary to make this kind of data transmitting and receiving successful.
- As shown in Figure 8.3, each piece of serial data should be configured as:
  - A logic **High** is on the communication line when the transmission line is **idle**.
  - A **High-to-Low** transaction starts an asynchronously data communication.
  - The data can be 5, 6, 7 or 8 bits attached with **0** or **1** parity bit and **1** or **2** stop bits.
- Figure 8.61 shows an example of using **7** data bits with **1** parity and **1** stop bit protocol.

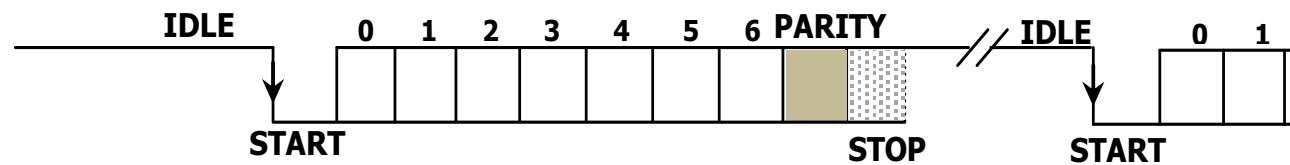


Figure 8.61 7 Bits + Parity + 1 Stop Bit data framing.

## 8.5.1 ASYNCHRONOUS SERIAL COMMUNICATION PROTOCOLS AND DATA FRAMING

- The TM4C123GH6PM MCU system provides eight (**8**) UART modules (**UART0 ~ UART7**), and each module can work independently to perform asynchronous serial data communications.
- Each module can be programmed to transmit (**TX**) and receive (**RX**) data by using the different baud rates and each module can be driven by using the **System Clock** or the internal **Precision Oscillator PIOSC**.
- The data to be transmitted is first converted from the parallel format to the serial format, and placed into the **transmit FIFO** via **UARTDR**. Then the data can be serially transmitted to the communication line via the transmit logic.
- When receiving data, the data stream is first converted from the serial to the parallel format, and placed into the **receive FIFO**. Then the data can be picked up from the receive FIFO and loaded into the MCU via **UARTDR**.

## 8.5.1 ASYNCHRONOUS SERIAL COMMUNICATION PROTOCOLS AND DATA FRAMING

- The **UART** can be configured for transmit or receive via the **TXE** and **RXE** bits in the **UART Control (UARTCTL)** register.
- Transmit and receive are both enabled after a system reset.
- Before any control registers can be programmed, the **UART** must be **disabled** by clearing the **UARTEN** bit in **UARTCTL** register. If the **UART** is disabled during a **TX** or **RX** operation, the current transaction is completed prior to the **UART** stopping.
- The **UART** module also includes a serial IR (**SIR**) encoder/decoder block that can be connected to an infrared transceiver to implement an IrDA SIR physical layer.
- The **SIR** function is programmed using the **UARTCTL** register.



## 8.5.2 ASYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

- The functional block diagram for one UART module is shown in Figure 8.62 (next slide).
- Each UART module can be configured and controlled by **four** groups of registers: the **Clock Control and Baud Rate Generation registers**, the **Control/Status registers** (including FIFOs), the **Interrupt Control and DMA Control registers**, and **Identification registers**.
- The functions and properties of these registers can be described in terms of each group.

### **1. Clock Control and Baud Rate Generation Register Group:**

- UART Clock Configuration (**UARTCC**) Register: Configure UART clock source.
- UART Control (**UARTCTL**) Register: Control the operations of UART.
- UART Integer Baud Rate Divisor (**UARTIBRD**) Register: Provide integer divider.
- UART Fractional Baud Rate Divisor (**UARTFBRD**) Register: Provide fraction divider.



## 8.5.2 ASYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

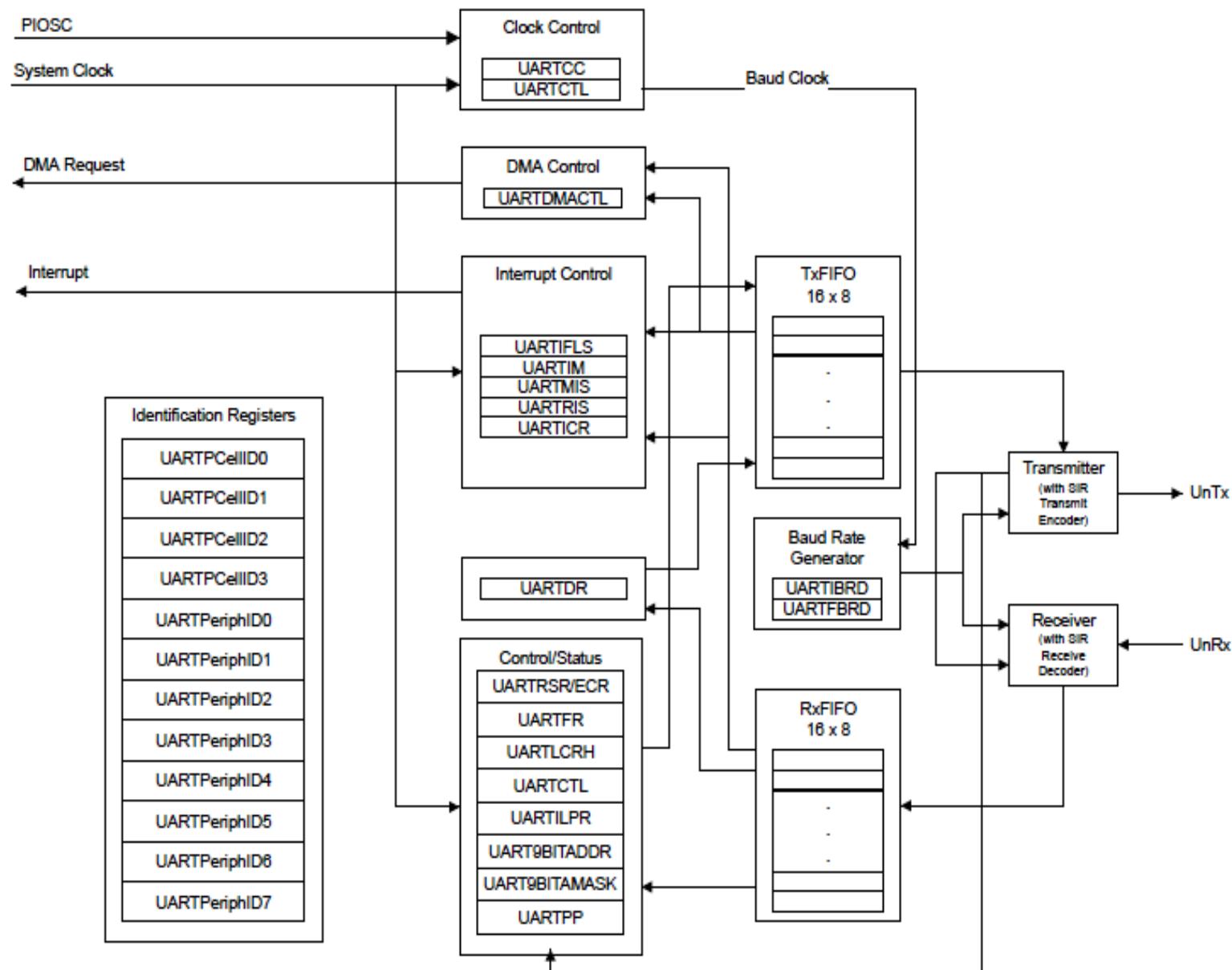


Figure 8.62 The functional block diagram for one UART module.

## 8.5.2 ASYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

### 2. Control/Status Register Group:

- UART Control (**UARTCTL**) Register: Control the operations of UART.
- UART Line Control Register (**UARTLCHR**): Setup data framing parameters.
- UART Data Register (**UARTDR**): Provide a temporary data storage for FIFO.
- UART Receive Status/Error Clear Register (**UARTRSR/UARTECR**): Provide data receive status and clear data framing, break and overrun errors.
- UART Flag Register (**UARTFR**): Provide UART working status.
- UART IrDA Low-Power Register (**UARTILPR**): Store the 8-bit low-power counter divisor value.
- UART 9-Bit Self Address Register (**UART9BITADDR**): This register is used in conjunction with **UART9BITAMASK** to form a match for address-byte received.
- UART 9-Bit Self Address Mask (**UART9BITAMASK**): Enable the address mask for 9-bit mode.
- UART Peripheral Properties (**UARTPP**) Register: Provide information to indicate whether to support 9-bit operation or smart card operation.
- Transmit FIFO (**T<sub>x</sub>FIFO**): Store  $16 \times 8$ -bit data to be transmitted.
- Receive FIFO (**R<sub>x</sub>FIFO**): Store  $16 \times 12$ -bit data to be received.



## 8.5.2 ASYNCHRONOUS SERIAL INTERFACE ARCHITECTURE & FUNCTIONAL BLOCK DIAGRAM

### 3. Interrupt Control and DMA Control Register Group:

- UART Interrupt FIFO Level Select (**UARTIFLS**) Register: Select the interrupt triggering level based on the FIFO data storage level.
- UART Raw Interrupt Status (**UARTRIS**) Register: Provide raw interrupt status.
- UART Interrupt Mask (**UARTIM**) Register: Control whether the UARTRIS can be sent to the interrupt controller or not.
- UART Masked Interrupt Status (**UARTMIS**) Register: Provide the current masked interrupt status.
- UART Interrupt Clear Register (**UARTICR**): Clear any interrupt.
- UART DMA Control (**UARTDMACTL**) Register: Control the DMA operations.

### 4. UART Identification Register Group:

- 12 registers are included in this group to provide software identification functions for each UART module.
- Next let's discuss these registers based on their group one by one.



### 8.5.3.1 TRANSMIT/RECEIVE LOGIC AND DATA TRANSMISSION AND RECEIVING

- The transmit logic performs parallel-to-serial conversion on the data read from the transmit FIFO.
- The control logic outputs the serial bit stream beginning with a **Start** bit (High-to-Low edge) and followed by the data bits with **LSB** first, parity bit, and the stop bits.
- The receive logic performs serial-to-parallel conversion on the received data bit stream after a valid **Start** pulse (Low-to-High) has been detected.
- The Overrun, parity, frame error checking, and line-break detection are also performed, and their status (**4** bits) accompanying the data (**8** bits) together are written to the receive FIFO ( $16 \times 12$ ).
- Data received or transmitted are stored in two 16-byte FIFOs, however the receiving FIFO has an extra 4 bits per 8-bit data item for status information.



### 8.5.3.1 TRANSMIT/RECEIVE LOGIC AND DATA TRANSMISSION AND RECEIVING

- For transmission, data is written into the **transmit FIFO**.
- If the UART is enabled, it causes a data frame to start transmitting with the parameters indicated in the **UARTLCRH** register. Data continues to be transmitted until there is no data left in the transmit FIFO.
- The **BUSY** bit in the **UART Flag (UARTFR)** register is asserted as soon as data is written to the transmit to make the FIFO non-empty and remains asserted as long as data is being transmitted. The **BUSY** bit is cleared only when the transmit FIFO is empty, and the last character has been transmitted from the shift register, including the **Stop** bits.
- The UART can indicate that it is busy even though the UART may no longer be enabled.
- When the receiver is idle (the **UnRx** signal is continuously **1**), and the data input detects a High-to-Low transition (a **Start** bit has been received), the receive counter begins running and data is sampled on the **8<sup>th</sup>** cycle of **Baud16** or **4<sup>th</sup>** cycle of **Baud8** depending on the setting of the **HSE** bit (bit 5) in **UARTCTL** register.

### 8.5.3.1 TRANSMIT/RECEIVE LOGIC AND DATA TRANSMISSION AND RECEIVING

- The **Start** bit is valid and recognized if the **UnRx** signal is still low on the **8<sup>th</sup>** cycle of **Baud16** (**HSE = 0**) or the **4<sup>th</sup>** cycle of **Baud 8** (**HSE = 1**), otherwise it is ignored.
- After a valid **Start** bit is detected, successive data bits are sampled on every **16<sup>th</sup>** cycle of **Baud16** or **8<sup>th</sup>** cycle of **Baud8** (that is, one bit period later) according to the programmed length of the data and value of the **HSE** bit in **UARTCTL** register.
- The **parity** bit is then checked if parity mode is enabled. Data length and parity are defined in the **UARTLCRH** register.
- Lastly, a valid **Stop** bit is confirmed if the **UnRx** signal is **High**, otherwise a framing error has occurred.
- When a full word is received, the data is stored into the **receive FIFO** along with any error bits associated with that word.



### 8.5.3.2 UART MODEM HANDSHAKE SUPPORT

- A UART can work as a **modem** to provide the interface between a computer and a serial asynchronous device.
- A computer can also be considered as UART device when it is connected as **Data Terminal Equipment (DTE)** and a model can be considered a **Data Communication Equipment (DCE)**.
- In general, a modem is a **DCE** and a computing device that connects to a modem is a **DTE**. Figure 8.63 shows an illustrating block diagram for these equipments.

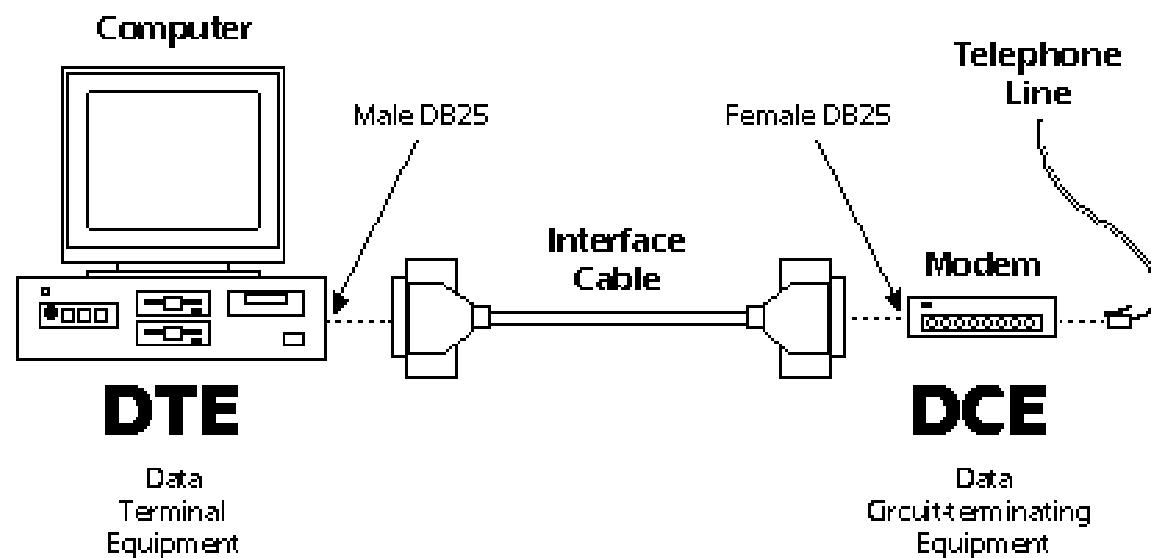


Figure 8.63 The DTE and DCE connections.

### 8.5.3.2 UART MODEM HANDSHAKE SUPPORT

- In all eight UART modules, only **UART1** can be used to work as a **modem** to provide flow control ability for asynchronous data communications.
- When using UART to perform asynchronous data operations, the data communication can be divided into the following **4 modes** based on its functionality: **simplex**, **half-duplex**, **full-duplex** and **multiplex**.
  - **Simplex**: Serial communication is only taking place in one direction, either from DTE to DCE or visa over.
  - **Half-duplex**: Serial communication can take place in both directions, but the communication can only take place in one direction at a moment. This means that either sender or receiver can send or receive the information in different time, but they cannot send and receive the data simultaneously.
  - **Full-duplex**: Allows serial communications to take place in both directions at the same time, which means that both sender and receiver can handle and exchange the data information simultaneously.
  - **Multiplex**: Allows multiple serial communications channels to occur over the same serial communication line. Multiplex operations are performed by either allocating separate frequencies or time slice to the individual serial communication channels.

### 8.5.3.2 UART MODEM HANDSHAKE SUPPORT

- A full group of modem flow control signals used for **DTE** and **DCE** are shown in Figure 8.64.

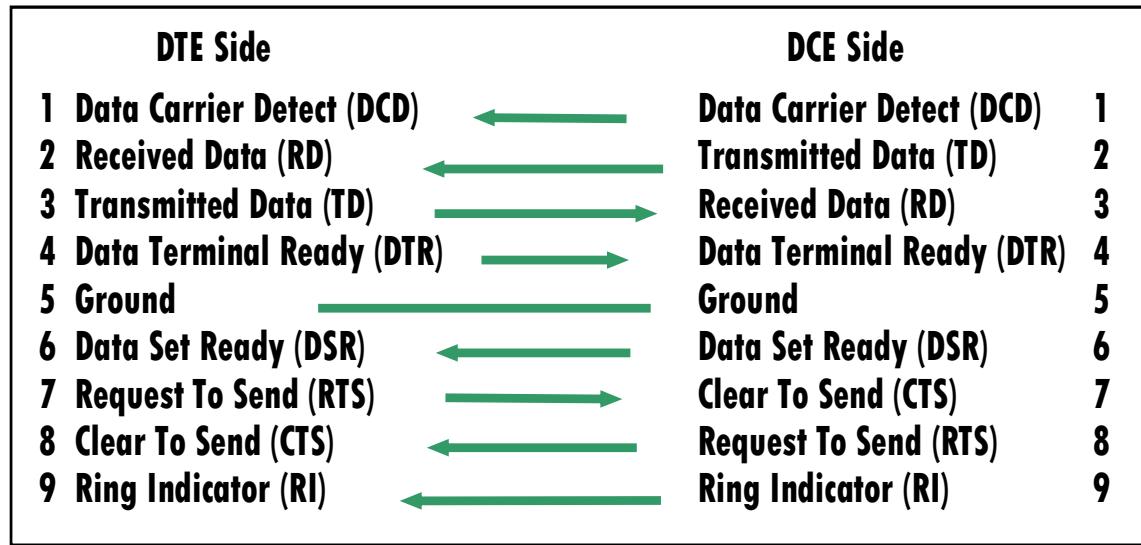


Figure 8.64 The modem flow control signals used in UART communications.

- In UART **module 1**, only two signals, **U1CTS** and **U1RTS**, are used and they are defined differently based the roles of these signals. When used as a **DTE**, the modem flow control signals are defined as (see Figure 8.64):
  - **/U1CTS** is Clear To Send (**CTS**).
  - **/U1RTS** is Request To Send (**RTS**).

### 8.5.3.2 UART MODEM HANDSHAKE SUPPORT

- When used as a **DCE**, the modem flow control signals are defined as:
  - **/U1CTS** is Request To Send (**RTS**).
  - **/U1RTS** is Clear To Send (**CTS**).
- The flow control can be designed and implemented by either hardware or software. The so-called hardware flow control is to use **RTS** and **CTS** lines to get the UART status. The software flow control is to use UART interrupts to get the UART status.
- Hardware flow control between two devices is accomplished by connecting the **/U1RTS** output to the **/CTS** input on the receiving device, and connecting the **/RTS** output on the receiving device to the **/U1CTS** input.
- The **/U1CTS** input controls the transmitter. The transmitter may only transmit data when the **/U1CTS** input is asserted. The **/U1RTS** output signal indicates the state of the receive FIFO. **/U1CTS** remains asserted until the preprogrammed watermark level is reached, indicating that the **Receive FIFO** has no space to store additional characters.

### 8.5.3.2 UART MODEM HANDSHAKE SUPPORT

- The **UARTCTL** register bits **15 (CTSEN)** and **14 (RTSEN)** specify the flow control mode as shown in Table 8.30.

Table 8.30 The modem flow control signals used in UART1.

<b>CTSEN</b>	<b>RTSEN</b>	<b>Description</b>
<b>1</b>	<b>1</b>	Both RTS and CTS flow control enabled
<b>1</b>	<b>0</b>	Only CTS flow control enabled
<b>0</b>	<b>1</b>	Only RTS flow control enabled
<b>0</b>	<b>0</b>	Both RTS and CTS flow control disabled

- Software flow control between two devices is implemented by using interrupts to indicate the status of the UART.
- Interrupts may be generated for the **/U1CTS** signal using bit **1** of the **UARTIM** register. The raw and masked interrupt status may be checked using the **UARTRIS** and **UARTMIS** register. These interrupts may be cleared using the **UARTICR** register.



### 8.5.3.3 UART FIFO OPERATIONS

- The UART has **two** FIFOs:
  - One for **transmit FIFO** ( $16 \times 8$ ).
  - One for **receive FIFO** ( $16 \times 12$ ).
- Both FIFOs are accessed via two **UARTDR** registers.
- **Read** operations of the **UARTDR** register return a **12-bit** value from the **receive FIFO**, which is consisting of **8** data bits and **4** error flags.
- **Write** operations place **8-bit** data into the **transmit FIFO**.
- Two **different** **UARTDR** registers, one is for **transmit data** and the other one is for **receive data**, are used in each UART module although they have the **same address**.
- After a system reset, both FIFOs are **disabled** and act as **1-byte-deep** holding registers. Both FIFOs are **enabled** by setting the **FEN** bit in **UARTLCRH** register.



### 8.5.3.3 UART FIFO OPERATIONS

- FIFO **status** can be monitored via the **UART Flag Register (UARTFR)** and the **UART Receive Status Register (UARTRSR)**.
- **Three** FIFO statuses, **empty**, **full** and **overrun**, are monitored.
- The **UARTFR** register contains **empty** and **full flags** in TXFE, TXFF, RXFE, and RXFF bits, the **UARTRSR** register shows **overrun** status via the **OE** bit.
- If both FIFOs are **disabled**, the empty and full flags are set based on the status of the **1-byte-deep** holding registers.
- The trigger points at which the FIFOs generate interrupts is controlled via the **UART Interrupt FIFO Level Select (UARTIFLS)** register.
- Both FIFOs can be individually configured to trigger interrupts at different levels, such as **1/8**, **1/4**, **1/2**, **3/4**, and **7/8**. For example, if the **1/4** option is selected for the receive FIFO, the UART generates a receive interrupt after 4 data bytes are received.
- After a system reset, both FIFOs are set to trigger an interrupt at the **1/2** level.

#### 8.5.3.4 UART INTERRUPTS AND DMA CONTROL

- An UART interrupt can be generated for many possible reasons and interrupt sources, which include 1) any data operation error such as overrun, break, parity, framing and receive timeout, 2) either data transmit or data receive complete.
- All of these can be considered as interrupt sources and can be combined or **ORed** together to generate a single interrupt request that can be handled and processed in a single interrupt service routine (ISR) or handler.
- The interrupt events that can trigger a controller-level interrupt are defined in the **UART Interrupt Mask (UARTIM)** register by setting the corresponding **IM** bits. If interrupts are not used, the raw interrupt status is visible via the **UART Raw Interrupt Status (UARTRIS)** register.
- Any UART interrupt can be cleared by writing **1** to the corresponding bit in the **UART Interrupt Clear Register (UARTICR)**.



#### 8.5.3.4 UART INTERRUPTS AND DMA CONTROL

- A **receive timeout interrupt** is generated when the **receiving FIFO** is not empty and no further data is received over a 32-bit period when the **HSE** bit is clear or over a 64-bit period when the **HSE** bit is set.
- This receive timeout interrupt can be cleared either when the FIFO becomes empty through reading all the data or by reading the holding register, or when a **1** is written to the corresponding bit in the **UARTICR** register.
- The receive interrupt changes state when one of the following events occurs:
  - If both FIFOs are **enabled** and the receive FIFO reaches the programmed trigger level, the **RXRIS** bit is set. The receive interrupt is cleared by reading data from the receive FIFO until it becomes less than the trigger level, or by clearing the interrupt by writing a **1** to the **RXIC** bit.
  - If both FIFOs are **disabled** and data is received thereby filling the location, the **RXRIS** bit is set. The receive interrupt is cleared by performing a single read of the receive FIFO, or by clearing the interrupt by writing a **1** to the **RXIC** bit.



#### 8.5.3.4 UART INTERRUPTS AND DMA CONTROL

- The transmit interrupt changes state when one of the following events occurs:
  - If both FIFOs are **enabled** and the transmit FIFO progresses over the programmed trigger level, the **TXRIS** bit is set.
  - Then the transmit interrupt is generated based on a transition through level and therefore the FIFO must be written past the programmed trigger level otherwise no further transmit interrupts will be generated.
  - The transmit interrupt is cleared by writing data to the transmit FIFO until it becomes greater than the trigger level, or by clearing the interrupt by writing a **1** to the **TXIC** bit.
  - If both FIFOs are **disabled** and there is no data present in the transmitters single location, the **TXRIS** bit is set.
  - It is cleared by performing a single write to the transmit FIFO, or by clearing the interrupt by writing a **1** to the **TXIC** bit.
- The UART also provides an interface to the μDMA controller with separate channels for data transmit and receive.
- The DMA operation is enabled via the **UART DMA Control (UARTDMACTL)** register.

#### 8.5.3.4 UART INTERRUPTS AND DMA CONTROL

- When DMA operation is **enabled**, the UART asserts a DMA request on the receive or transmit channel when the associated FIFO can transfer data.
  - For the receive channel, a single transfer request is generated whenever any data is in the receive FIFO. A burst transfer request is asserted when the amount of data in the receive FIFO is at or above the FIFO trigger level set in the **UARTIFLS** register.
  - For the transmit channel, a single transfer request is generated whenever there is at least one empty location in the transmit FIFO. The burst request is generated whenever the transmit FIFO contains fewer characters than the FIFO trigger level. The single and burst DMA transfer requests are handled automatically by the μDMA controller depending on how the DMA channel is configured.
- To enable DMA operation for the receive channel, set the **RXDMAE** bit of the **UARTDMACTL** register.
- To enable DMA operation for the transmit channel, set the **TXDMAE** bit of the **UARTDMACTL** register. The UART can also be configured to stop using DMA for the receive channel if a receive error occurs. If the **DMAERR** bit of the **UARTDMACR** register is set and a receive error occurs, the DMA receive requests are automatically disabled. This error condition can be cleared by clearing the appropriate UART error interrupt.

### 8.5.3.5 UART SERIAL IR (SIR) SUPPORT

- The UART peripheral includes an **IrDA serial-IR (SIR)** encoder/decoder block. The IrDA SIR block provides functionality that converts between an asynchronous UART data stream and a half-duplex serial SIR interface.
- No analog processing is performed on-chip. The role of the **SIR** block is to provide a digital encoded output and decoded input to the UART.
- When enabled, the SIR block uses the **UnTx** and **UnRx** pins for the SIR protocol.
- These signals should be connected to an infrared transceiver to implement an IrDA SIR physical layer link. The SIR block can receive and transmit, but it is only half-duplex so it cannot do both at the same time.
- Transmission must be stopped before data can be received. The IrDA SIR physical layer specifies a minimum **10** ms delay between transmission and reception.



### 8.5.3.6 9-BIT UART MODE

- The UART provides a **9-bit** mode that is enabled with the **9BITEN** bit in the **UART9BITADDR** register.
- This feature is useful in a multi-drop configuration of the UART where a single master connected to multiple slaves can communicate with a particular slave through its address or set of addresses along with a qualifier for an address byte.
- When working in this mode, all the slaves check for the address qualifier in the place of the parity bit and, if set, then compare the byte received with the preprogrammed address. If the address matches, then it receives or sends further data. If the address does not match, it drops the address byte and any subsequent data bytes.
- If the UART is in 9-bit mode, then the receiver operates with no parity mode. The address can be predefined to match with the received byte and it can be configured with the **UART9BITADDR** register. The matching can be extended to a set of addresses using the address mask in the **UART9BITAMASK** register. By default, the **UART9BITAMASK** is **0xFF**, meaning that only the specified address is matched.

### 8.5.3.6 9-BIT UART MODE

- When no match can be found, the rest of the data bytes with the **9th** bit cleared are dropped. If a match is found, then an interrupt is generated to the NVIC for further action.
- The subsequent data bytes with the cleared 9th bit are stored in the FIFO. Software can mask this interrupt in case µDMA and/or FIFO operations are enabled for this instance and processor intervention is not required.
- All the sending transactions with 9-bit mode are data bytes and the 9th bit is cleared. Software can override the 9th bit to be set by overriding the parity settings to sticky parity with odd parity enabled for a particular byte.
- To match the transmission time with correct parity settings, the address byte can be transmitted as a single then a burst transfer. The **Transmit FIFO** does not hold the address/data bit, hence software should take care of enabling the address bit appropriately.
- Next let's concentrate on these control registers used for these operations.



### 8.5.3.7 UART MODULE CLOCK CONTROL & BAUD RATE GENERATION REGISTERS

- The registers in this group are used to select the **clock source** for the UART module and define the **baud rate** for the data operations. There are **four** registers in this group and they are:
  - UART Clock Configuration (**UARTCC**) Register.
  - UART Control (**UARTCTL**) Register.
  - UART Integer Baud Rate Divisor (**UARTIBRD**) Register.
  - UART Fractional Baud Rate Divisor (**UARTFBRD**) Register.
- The **UARTCC** is a 32-bit register but only the **4** lowest bits, bits **3 ~ 0** or **CS** bit-field, are used to determine the clock source for the UART module. When **CS = 0x0**, the **System Clock** is selected. When **CS = 0x5**, the **PIOSC** is selected.
- Only bit **5 (HSE)** in the **UARTCTL** register is used to enable the UART module to use different clock rate by dividing the System Clock with different factors. If the **HSE = 0**, the System Clock is divided by **16**, otherwise if **HSE = 1**, the System Clock is divided by **8**.



### 8.5.3.7 UART MODULE CLOCK CONTROL & BAUD RATE GENERATION REGISTERS

- Both **UARTIBDR** and **UARTFBDR** registers are 32-bit registers and used to store integer and fractional dividing factors used to calculate the baud rate.
- The lower 16-bit on the **UARTIBDR** register is used to reserve an integer divisor (**BRDI**), and the lower 6-bit in the **UARTFBDR** register is to reserve a fractional divisor (**BRDF**). The relationship between these divisors and the Baud Rate Divisor (**BRD**) is:
  - **BRD = BRDI + BRDF = UARTSysClk / (ClkDiv × Baud Rate)**
- where **UARTSysClk** is the System Clock frequency. The **ClkDiv = 16** if **HSE = 0** and **ClkDiv = 8** if **HSE = 1** in the **UARTCTL** register. Therefore the Baud Rate for the UART is:
  - **Baud Rate = UARTSysClk / (BRDI + BRDF) × ClkDiv**
- The 6-bit fractional number that is in the **DIVFRAC** bit field in the **UARTFBRD** register can be calculated by taking the fractional part of the baud-rate divisor, multiplying it by **64**, and adding **0.5** to account for rounding errors:
  - **UARTFBRD[DIVFRAC] = Integer (BRDF \* 64 + 0.5)**

### 8.5.3.7 UART MODULE CLOCK CONTROL & BAUD RATE GENERATION REGISTERS

- The reason for using the **ClkDiv** parameter is because the UART generates an *internal baud-rate reference clock* at **8x** or **16x** the baud-rate referred to as **Baud8** and **Baud16**, depending on the setting of the **HSE** bit (**bit 5**) in **UARTCTL**.
- This reference clock must be divided by **8** or **16** to get the transmit clock. This reference clock is used for error detection during receive operations. Note that the state of the **HSE** bit has no effect on clock generation in **ISO 7816** smart card mode.
- The **UART Line Control Register (UARTLCRH)** is combined with the **UARTIBRD** and **UARTFBRD** registers together to form an internal **30-bit** register (**UARTLCRH:Lower 8-bit**, **UARTIBRD: Lower 16-bit** and **UARTFBRD: Lower 6-bit**).
- Only when a writing operation is performed to the **UARTLCRH** register, this international 30-bit register should be updated. This means that any change to the baud rate divisor (including changes to either **UARTIBRD** or **UARTFBRD** register) must be followed by a write operation to the **UARTLCRH** register for the change to take effect.

### 8.5.3.8 UART MODULE CONTROL/STATUS AND FIFO CONTROL REGISTERS

- The registers in this group includes:
  - UART Control (**UARTCTL**) Register.
  - UART Line Control Register (**UARTLCHR**).
  - UART Data Register (**UARTDR**).
  - UART Receive Status/Error Clear Register (**UARTRSR/UARTECR**).
  - UART Flag Register (**UARTFR**).
  - UART IrDA Low-Power Register (**UARTILPR**).
  - UART 9-Bit Self Address Register (**UART9BITADDR**).
  - UART 9-Bit Self Address Mask (**UART9BITAMASK**).
  - UART Peripheral Properties (**UARTPP**) Register.
  - Transmit FIFO (**T<sub>x</sub>FIFO**) and Receive FIFO (**R<sub>x</sub>FIFO**).
- In this section, we only introduce and discuss some important and popular registers. For **UART9BITADDR**, **UART9BITAMASK** and **UARTPP** registers, refer to related documents to get details.

### 8.5.3.8.1 UART CONTROL REGISTER (UARTCTL)

- The bit configuration for the **UARTCTL** register is shown in Figure 8.65. The bit field and bit function for this register is shown in Table 8.31 (next slide).

UARTCTL Register Bit Configuration																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
reserved																
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	CTSEN	RTSEN	reserved		RTS	reserved	RXE	TXE	LBE	reserved	HSE	EOT	SMART	SIRLP	SIREN	UARTEN
Type	RW	RW	RO	RO	RW	RO	RW	RW	RW	RO	RW	RW	RW	RW	RW	
Reset	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	

Figure 8.65 Bit configurations of the UARTCTL register.

- The **UARTCTL** register is the control register. All bits are cleared on a system reset except for the Transmit Enable (TXE) and Receive Enable (RXE) bits, which are set.
- To enable the UART module, the **UARTEN** bit must be set. If software requires a configuration change in the module, the **UARTEN** bit must be cleared to disable the UART before the configuration changes can be written.

## 8.5.3.8.1 UART CONTROL REGISTER (UARTCTL)

Table 8.31 Bit field value and its function for UARTCTL register.

Bit	Name	Reset	Function
<b>31:16</b>		0x000	
<b>13:12</b>	<b>Reserved</b>	0x0 0x0	Reserved
<b>10, 6</b>		0x0	
<b>15</b>	<b>CTSEN</b>	0x0	Clear To Send Enable. <b>0:</b> CTS is disabled. <b>1:</b> CTS is enabled. Data is only transmitted when the U1CTS signal is asserted.
<b>14</b>	<b>RTSEN</b>	0x0	Request To Send Enable. <b>0:</b> RTS is disabled. <b>1:</b> RTS is enabled. Data is only requested when receive FIFO has available entries..
<b>11</b>	<b>RTS</b>	0x0	Request To Send. When <b>RTSEN</b> is clear, the status of this bit is reflected on the <b>U1RTS</b> signal. If <b>RTSEN</b> is set, this bit is ignored on a write and should be ignored on read.
<b>9</b>	<b>RXE</b>	0x0	UART Receive Enable. <b>0:</b> Receiver is disabled. <b>1:</b> Receiver is enabled.
<b>8</b>	<b>TXE</b>	0x0	UART Transmit Enable. <b>0:</b> Transmitter is disabled. <b>1:</b> Transmitter is enabled.
<b>7</b>	<b>LBE</b>	0x0	UART Loop Back Enable. <b>0:</b> Normal operation. <b>1:</b> <b>UnTx</b> is connected to the <b>UnRx</b> to form a loop.
<b>5</b>	<b>HSE</b>	0x0	High-Speed Operation. <b>0:</b> UART clock = System Clock / 16; <b>1:</b> UART clock = System Clock / 8.
<b>4</b>	<b>EOT</b>	0x0	End Of Transmission. <b>0:</b> The <b>TXRIS</b> bit is set when the transmit FIFO condition specified in <b>UARTIFLS</b> is met. <b>1:</b> The TXRIS bit is set only after all transmitted data have cleared the serialized.
<b>3</b>	<b>SMART</b>	0x0	Smart Card Support. <b>0:</b> Normal operation. <b>1:</b> Smart card mode.
<b>2</b>	<b>SIRLP</b>	0x0	UART SIR Low-Power Mode. <b>0:</b> Low-level bits are transmitted as an active High pulse with a width of 3/16th of the bit period; <b>1:</b> SIR low-power mode.
<b>1</b>	<b>SIREN</b>	0x0	UART SIR Enable. <b>0:</b> Normal operation; <b>1:</b> The SIR mode.
<b>0</b>	<b>UARTEN</b>	0x0	UART Enable. <b>0:</b> UART is disabled; <b>1:</b> UART is enabled.

### 8.5.3.8.1 UART CONTROL REGISTER (UARTCTL)

- A point to be noted is that the **UARTCTL** register should not be modified while the UART is enabled or else the results are unpredictable.
- The following sequence is recommended for making changes to the **UARTCTL** register.
  - Disable the UART.
  - Wait for the end of transmission or reception of the current character.
  - Flush the transmit FIFO by clearing bit **4 (FEN)** in the line control register **UARTLCRH**.
  - Reprogram to modify the control register.
  - Enable the UART.



### 8.5.3.8.2 UART LINE CONTROL REGISTER (UARTLCRH)

- The **UARTLCRH** register is the line control register and it is used to configure serial data communication parameters such as data length, parity, and stop bit.
- When updating the baud-rate divisor (**UARTIBRD** and/or **UARTIFRD**), the **UARTLCRH** register must also be written.
- The bit configuration for the **UARTLCRH** register is shown in Figure 8.66. The bit field and bit function for this register is shown in Table 8.32 (next slide).

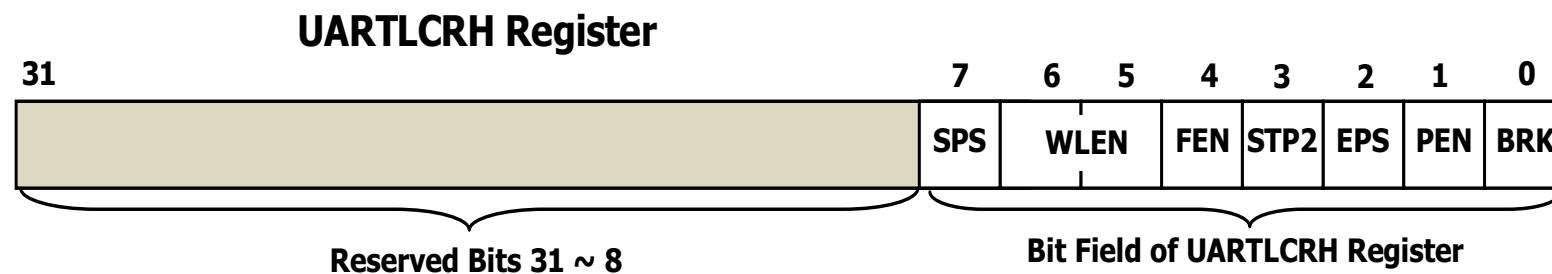


Figure 8.66 Bits configurations for UARTLCRH register.

## 8.5.3.8.2 UART LINE CONTROL REGISTER (UARTLCRH)

Table 8.32 Bit field value and its function for UARTLCRH register.

Bit	Name	Reset	Function
<b>31:8</b>	<b>Reserved</b>	0x000	Reserved
<b>7</b>	<b>SPS</b>	0x0	UART Stick Parity Select When bits 1, 2, and 7 of <b>UARTLCRH</b> are set, the parity bit is transmitted and checked as a 0. When bits 1 and 7 are set and 2 is cleared, the parity bit is transmitted and checked as a 1. When this bit is cleared, stick parity is disabled.
<b>6:5</b>	<b>WLEN</b>	0x0	UART Word Length The bits indicate the number of data bits transmitted or received in a frame as follows: <b>0x0</b> : 5 bits (default); <b>0x1</b> : 6 bits; <b>0x2</b> : 7 bits; <b>0x3</b> : 8 bits.
<b>4</b>	<b>FEN</b>	0x0	UART FIFOs Enable. <b>0</b> : The FIFOs are disabled (Character mode). The FIFOs become 1-byte-deep holding registers. <b>1</b> : The transmit and receive FIFO buffers are enabled (FIFO mode).
<b>3</b>	<b>STP2</b>	0x0	UART 2 Stop Bits Select. <b>0</b> : 1 stop bit is used. <b>1</b> : 2 stop bits are used.
<b>2</b>	<b>EPS</b>	0x0	UART Even Parity Select. <b>0</b> : Odd parity is selected. <b>1</b> : Even parity is selected.
<b>1</b>	<b>PEN</b>	0x0	UART Parity Enable. <b>0</b> : No parity is used. <b>1</b> : Parity is enabled.
<b>0</b>	<b>BRK</b>	0x0	UART Send Break. <b>0</b> : Normal operation. <b>1</b> : A Low level is continually output on the UnTx signal, after completing transmission of the current character. For the proper execution of the break command, software must set this bit for at least two frames (character periods).

### 8.5.3.8.3 UART RECEIVE STATUS/ERROR CLEAR REGISTER (UARTRSR/UARTECR)

- The **UARTRSR/UARTECR** is a 32-bit register but only the lowest **4/8** bits are used to indicate the data operation status and all of these status bits can be cleared by writing any value to this register (Figure 8.67).
  - Therefore this register has two functions; when **reading**, the data operational status and error status are displayed, when **writing**, all error bits are cleared.

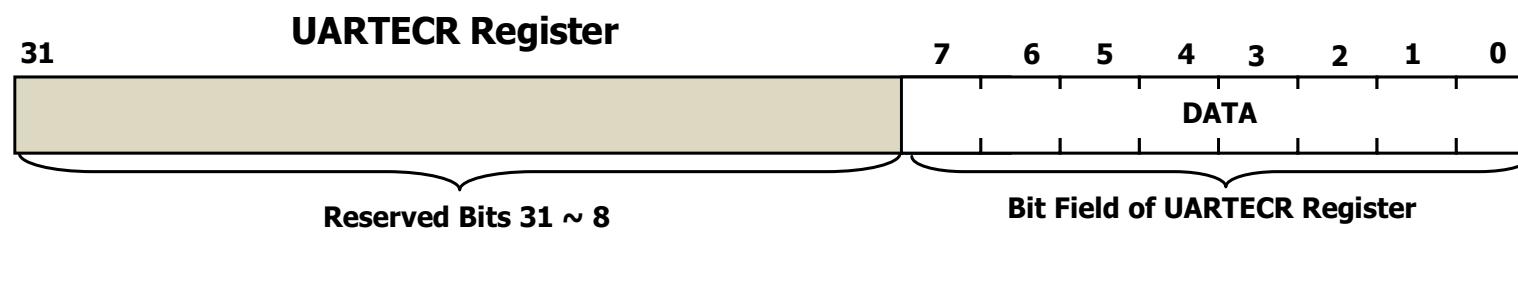
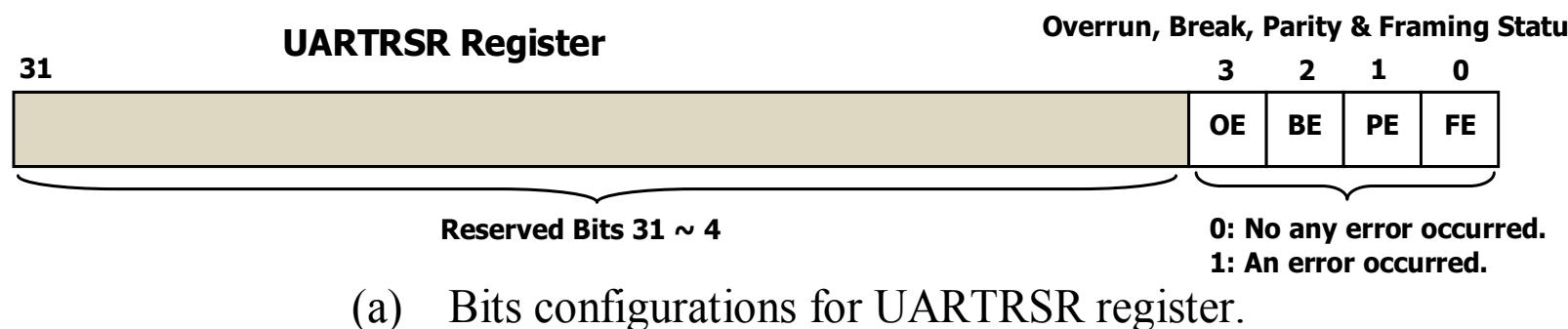


Figure 8.67 Bits configurations for UARTRSR/ECR register.

### 8.5.3.8.3 UART RECEIVE STATUS/ERROR CLEAR REGISTER (UARTRSR/UARTECR)

- In addition to the **UARTDR** registers, receive status can also be read from this **UARTRSR** register. If the status is read from this register, then the status information corresponds to the entry read from **UARTDR** prior to reading **UARTRSR**. The status information for overrun is set immediately when an overrun condition occurs.
- The **UARTRSR** register is a read-only register and it cannot be written. The **UARTECR** is a write-only register and it cannot be read.
- The **lower 4 bits** on the **UARTRSR** register are used to monitor 4 data operational statuses, which include: **Overrun**, **Break**, **Parity** and **Framing** statuses. If any kind of error is occurred, the corresponding bit is set to **1**.
- The **lower 8 bits** in the **UARTECR** register allows users to write any data into these fields to clear all errors displayed in the **UARTRSR** register if any of them occurred. The point to be noted is that the **UARTRSR** is a read-only register, but the **UARTECR** is a write-only register.



#### 8.5.3.8.4 UART DATA REGISTER (UARTDR)

- There are **two** 32-bit **UARTDR** registers with two different functions for data transmit and receive.
- In fact, these registers work as a bridge buffer between the MCU and FIFOs to temporarily store data to be transmitted into the transmit FIFO, and store data receiving from the receive FIFO. Physically these two **UARTDR** registers have no connection between them.
- Figure 8.68 shows the bits configurations for the receive **UARTDR** register.

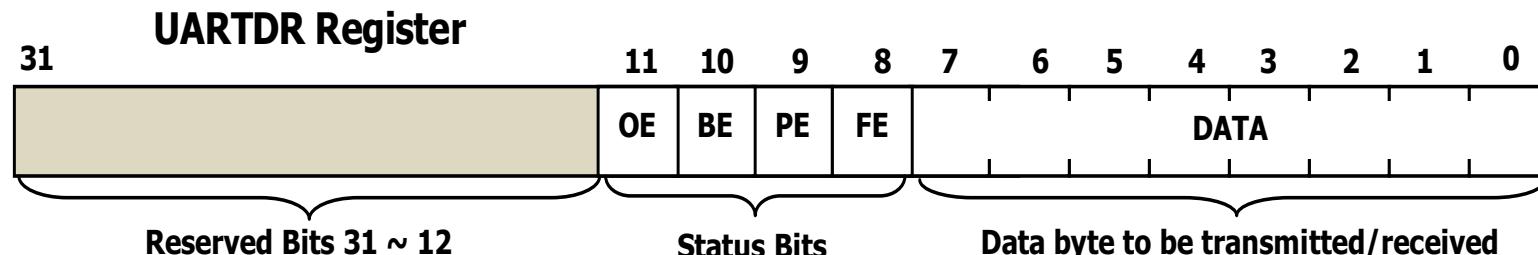


Figure 8.68 Bits configurations for UARTDR register.

#### 8.5.3.8.4 UART DATA REGISTER (UARTDR)

- For **transmitted data**, if the FIFO is enabled, an **8-bit** or a byte data written to the **transmit UARTDR** register is pushed onto the **transmit FIFO**.
- If the FIFO is disabled, data is stored in the transmitter holding register (the bottom word of the transmit FIFO). A write to this register initiates a transmission from the UART.
- For **received data**, if the FIFO is enabled, the received data byte and the **4-bit** status, **break**, **frame**, **parity**, and **overrun**, are pushed onto the **12-bit** wide **receiving FIFO**.
- If the FIFO is disabled, the data byte and status are stored in the receiving holding register (the bottom word of the receive FIFO). The received data can be retrieved by reading this **receive UARTDR** register.

### 8.5.3.8.5    UART FLAG REGISTER (UARTFR)

- The **UARTFR** is a 32-bit flag register used to monitor and display the additional UART running status.
  - After reset, the **TXFF**, **RXFF**, and **BUSY** bits are **0**, and **TXFE** and **RXFE** bits are **1**. The **CTS** bit indicates the modem flow control. The modem bits are only implemented on **UART1** and are reserved on **UART0** and **UART2**.
  - The bit configuration for the **UARTFR** register is shown in Figure 8.69. The bit field and bit function for this register is shown in Table 8.33 (next slide).
  - It can be found that this register is an addition to the **UARTRSR** register with more running statuses provided. Most of these statuses are depended on the FIFOs Enable bit **FEN** in the **UARTLCRH** register. The bit 3 (**BUSY**) is an important status used to indicate whether the UART is currently busy in transmitting data or not.

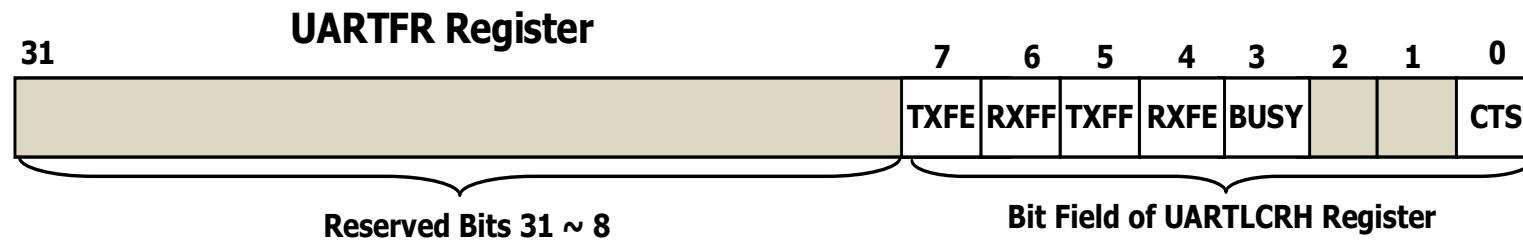


Figure 8.69 Bits configurations for UARTFR register.

## 8.5.3.8.5 UART FLAG REGISTER (UARTFR)

Table 8.33 Bit field value and its function for UARTFR register.

Bit	Name	Reset	Function
31:8	Reserved	0x000	Reserved
7	<b>TXFE</b>	0x0	<p>UART Transmit FIFO Empty. The meaning of this bit depends on the state of the <b>FEN</b> bit in the <b>UARTLCRH</b> register.</p> <p><b>0:</b> Transmitter has data to transmit. <b>1:</b> If the FIFO is disabled (<b>FEN</b> = 0), the transmit holding register is empty. If the FIFO is enabled (<b>FEN</b> = 1), the transmit FIFO is empty.</p>
6	<b>RXFF</b>	0x0	<p>UART Receive FIFO Full. The meaning of this bit depends on the state of the <b>FEN</b> bit in the <b>UARTLCRH</b> register.</p> <p><b>0:</b> The receiver can receive data. <b>1:</b> If the FIFO is disabled (<b>FEN</b> = 0), the receive holding register is full. If the FIFO is enabled (<b>FEN</b> = 1), the receive FIFO is full.</p>
5	<b>TXFF</b>	0x0	<p>UART Transmit FIFO Full. The meaning of this bit depends on the state of the <b>FEN</b> bit in the <b>UARTLCRH</b> register.</p> <p><b>0:</b> The transmitter is not full. <b>1:</b> If the FIFO is disabled (<b>FEN</b> = 0), the transmit holding register is full. If the FIFO is enabled (<b>FEN</b> = 1), the transmit FIFO is full.</p>
4	<b>RXFE</b>	0x0	<p>UART Receive FIFO Empty. The meaning of this bit depends on the state of the <b>FEN</b> bit in the <b>UARTLCRH</b> register.</p> <p><b>0:</b> Receiver is not empty. <b>1:</b> If the FIFO is disabled (<b>FEN</b> = 0), the receive holding register is empty. If the FIFO is enabled (<b>FEN</b> = 1), the receive FIFO is empty.</p>
3	<b>BUSY</b>	0x0	UART Busy. <b>0:</b> UART is not busy. <b>1:</b> UART is busy transmitting data.
2:1	Reserved	0x0	Reserved
0	<b>CTS</b>	0x0	Clear To Send. <b>0:</b> The <b>U1CTS</b> signal is not asserted. <b>1:</b> The <b>U1CTS</b> signal is asserted.

### 8.5.3.9 UART MODULE INTERRUPT AND DMA CONTROL REGISTERS

- The registers in this group include:
  - UART Interrupt FIFO Level Select (**UARTIFLS**) Register.
  - UART Raw Interrupt Status (**UARTRIS**) Register.
  - UART Interrupt Mask (**UARTIM**) Register.
  - UART Masked Interrupt Status (**UARTMIS**) Register.
  - UART Interrupt Clear Register (**UARTICR**).
  - UART DMA Control (**UARTDMACTL**) Register.
- Now let's have a closer look at these registers one by one.



### 8.5.3.9.1 UART INTERRUPT FIFO LEVEL SELECT (UARTIFLS) REGISTER

- The **UARTIFLS** register is the interrupt FIFO level select register. This register can be used to define the FIFO level at which the **TXRIS** and **RXRIS** bits in the **UARTRIS** register are triggered.
- The interrupts are generated based on a transition through a level rather than being based on the level. This means that the interrupts are generated when the fill level progresses through the trigger level. For example, if the receive trigger level is set to the half-way mark, the interrupt is triggered as the module is receiving the **9th** character (for 16 characters in FIFOs).
- Figure 8.70 shows the bit configurations of the **UARTIFLS** register.

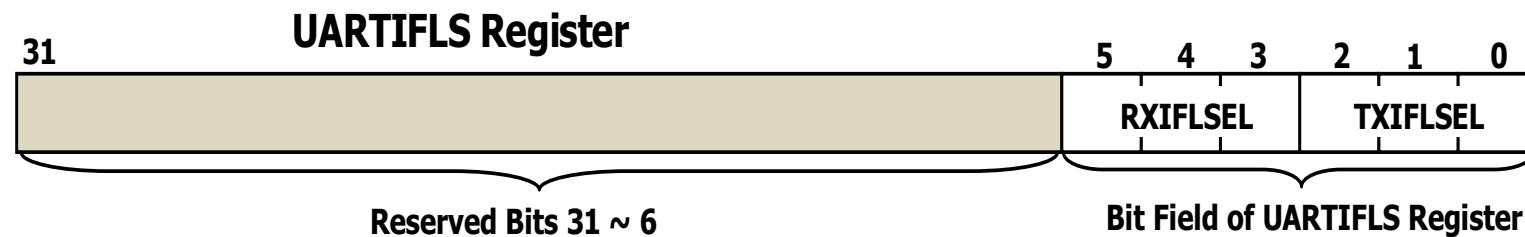


Figure 8.70 Bits configurations for UARTIFLS register.

### 8.5.3.9.1 UART INTERRUPT FIFO LEVEL SELECT (UARTIFLS) REGISTER

- After a system reset, the **TXIFLSEL** and **RXIFLSEL** bits are configured so that the FIFOs trigger an interrupt at the **half-way** mark.
- Table 8.34 shows the bit field and functions for this register.

Table 8.34 Bit field value and its function for UARTIFLS register.

Bit	Name	Reset	Function
31:6	<b>Reserved</b>	0x000	Reserved
5:3	<b>RXIFLSEL</b>	0x0	UART Receive Interrupt FIFO Level Select. <b>0x0:</b> RX FIFO $\geq \frac{1}{8}$ full; <b>0x1:</b> RX FIFO $\geq \frac{1}{4}$ full; <b>0x2:</b> RX FIFO $\geq \frac{1}{2}$ full (default) <b>0x3:</b> RX FIFO $\geq \frac{3}{4}$ full; <b>0x4:</b> RX FIFO $\geq \frac{7}{8}$ full; <b>0x5-0x7</b> Reserved.
2:0	<b>TXIFLSEL</b>	0x0	UART Transmit Interrupt FIFO Level Select. <b>0x0:</b> TX FIFO $\leq \frac{7}{8}$ empty; <b>0x1:</b> TX FIFO $\leq \frac{3}{4}$ empty; <b>0x2:</b> TX FIFO $\leq \frac{1}{2}$ empty (default) <b>0x3:</b> TX FIFO $\leq \frac{1}{4}$ empty; <b>0x4:</b> TX FIFO $\leq \frac{1}{8}$ empty; <b>0x5-0x7</b> Reserved.

## 8.5.3.9.2 UART RAW INTERRUPT STATUS (UARTRIS) REGISTER

- This 32-bit register uses the **lower 9 bits** to indicate the raw interrupt status for related UART events or errors. If a bit is **1**, a corresponding raw interrupt has been occurred. Otherwise if that bit is **0**, no related interrupt occurred.
- Figure 8.71 (top one) shows the bit configurations for this register.

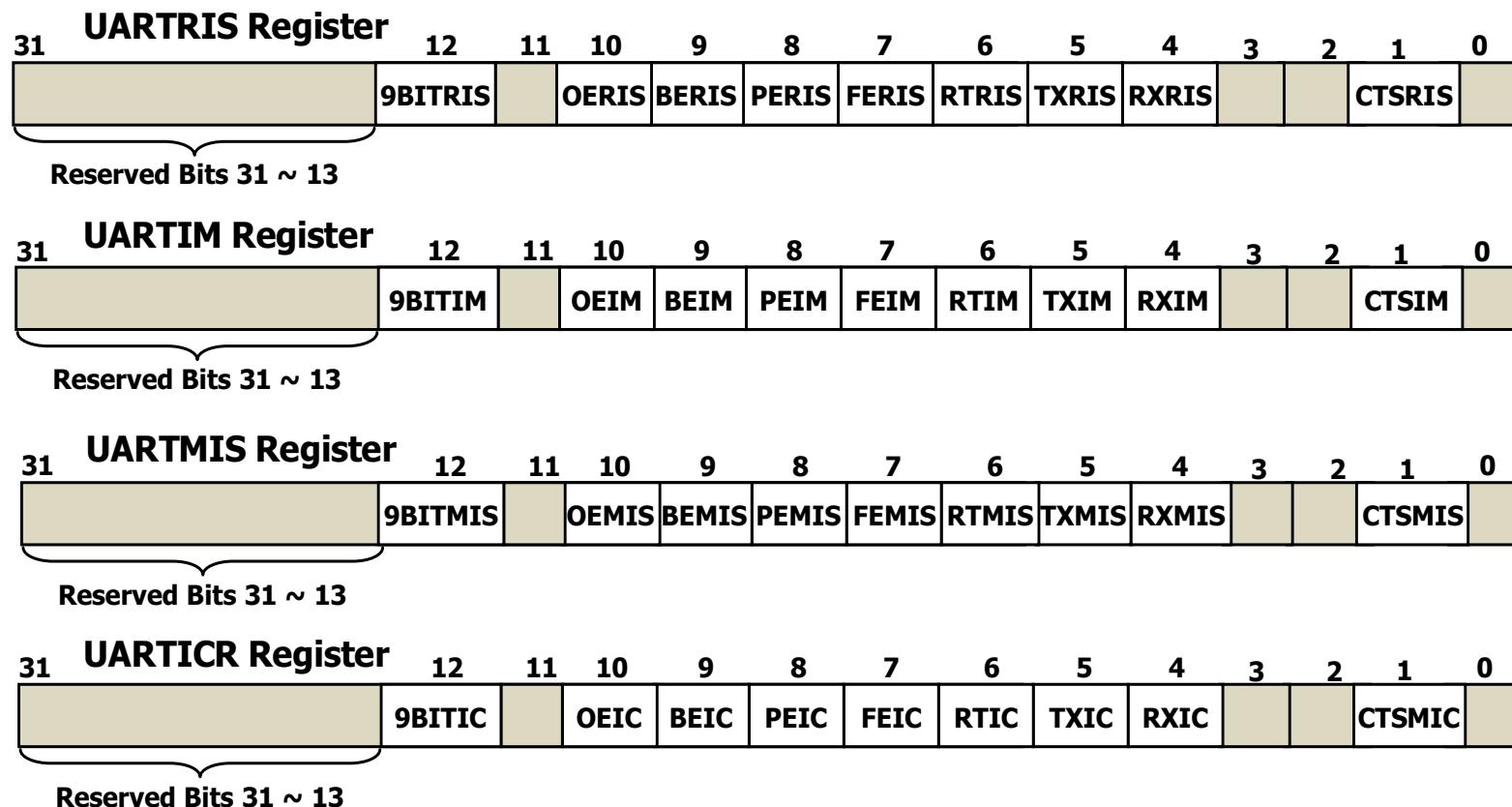


Figure 8.71 Bit configurations for UARTRIS, UARTIM, UARMIS and UARTICR.

### 8.5.3.9.3 UART INTERRUPT MASK (UARTIM) REGISTER

- Similar to **UARTRIS** register, this register is used to enable or disable the related raw interrupt to be sent to the NVIC (Figure 8.71). If a bit is **1**, the corresponding raw interrupt set in the **UARTRIS** register is unmasked and can be sent to the NVIC. If a bit is **0**, that raw interrupt cannot be sent to the NVIC.

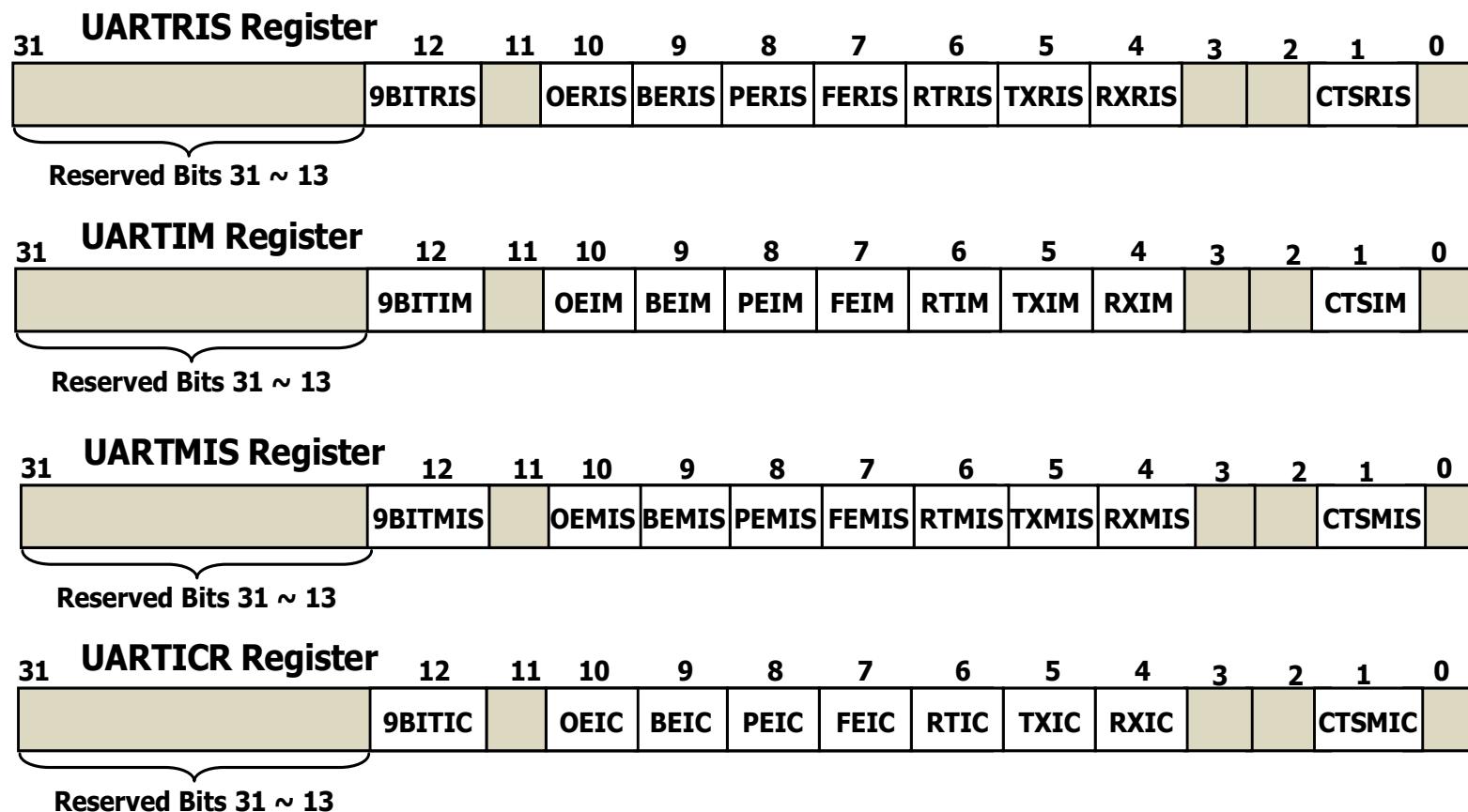


Figure 8.71 Bit configurations for UARTRIS, UARTIM, UARMIS and UARTICR.

## 8.5.3.9.4 UART MASKED INTERRUPT STATUS (UARTMIS) REGISTER

- The **UARTMIS** register is the masked interrupt status register. On a read, this register gives the current masked status of the corresponding interrupt. A write has no effect. The bit configuration is shown in Figure 8.71. If a bit is **1**, the corresponding raw interrupt set in the **UARTRIS** register has been sent to the NVIC. If a bit is **0**, the corresponding raw interrupt was not sent to the NVIC.

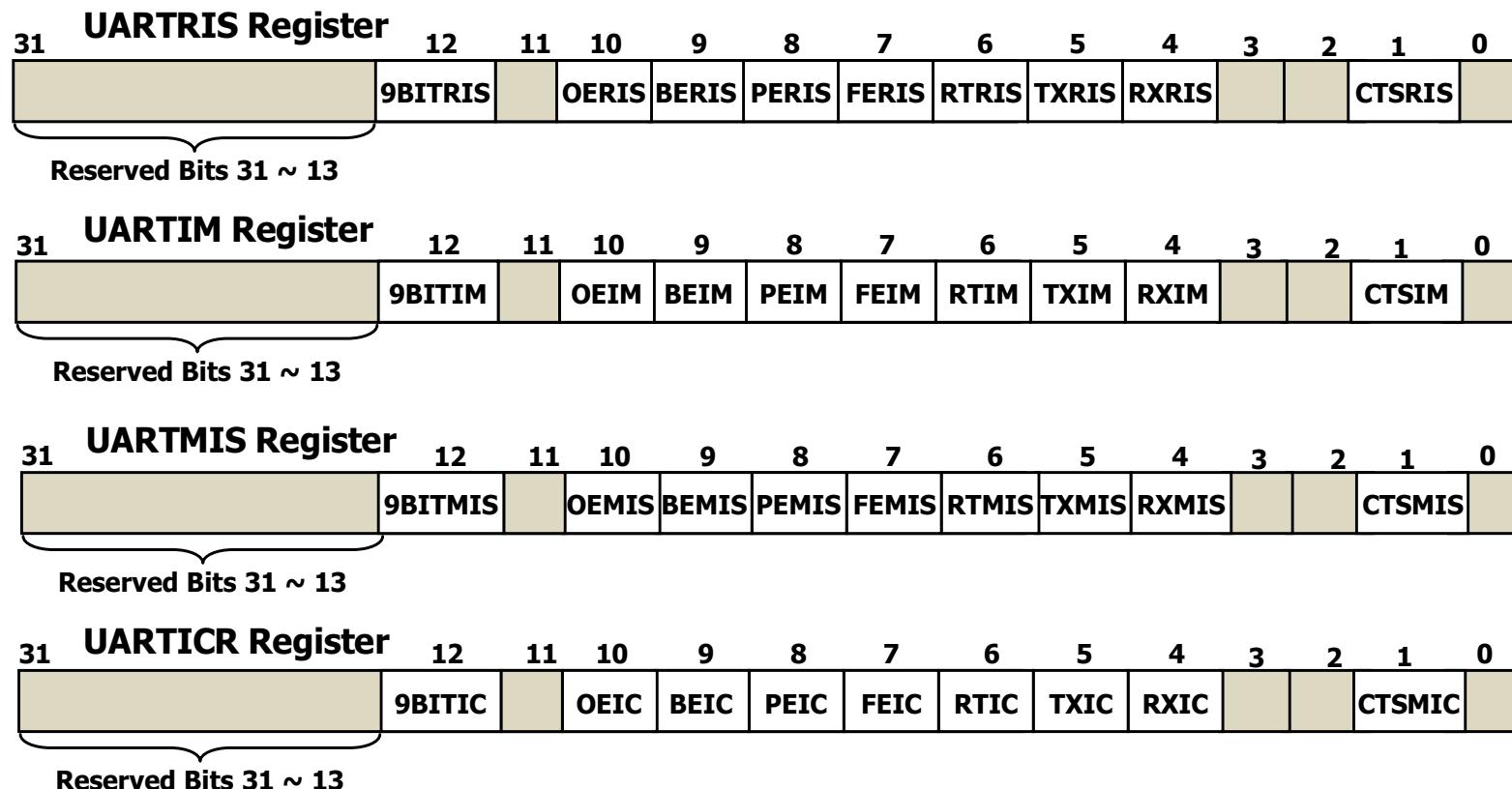


Figure 8.71 Bit configurations for UARTRIS, UARTIM, UARMIS and UARTICR.

## 8.5.3.9.5 UART INTERRUPT CLEAR REGISTER (UARTICR)

- The **UARTICR** register is the interrupt clear register. When writing a **1**, the corresponding interrupt, both raw interrupt and masked interrupt, is cleared. A write of **0** has no effect. The bit configurations are shown in Figure 8.71.

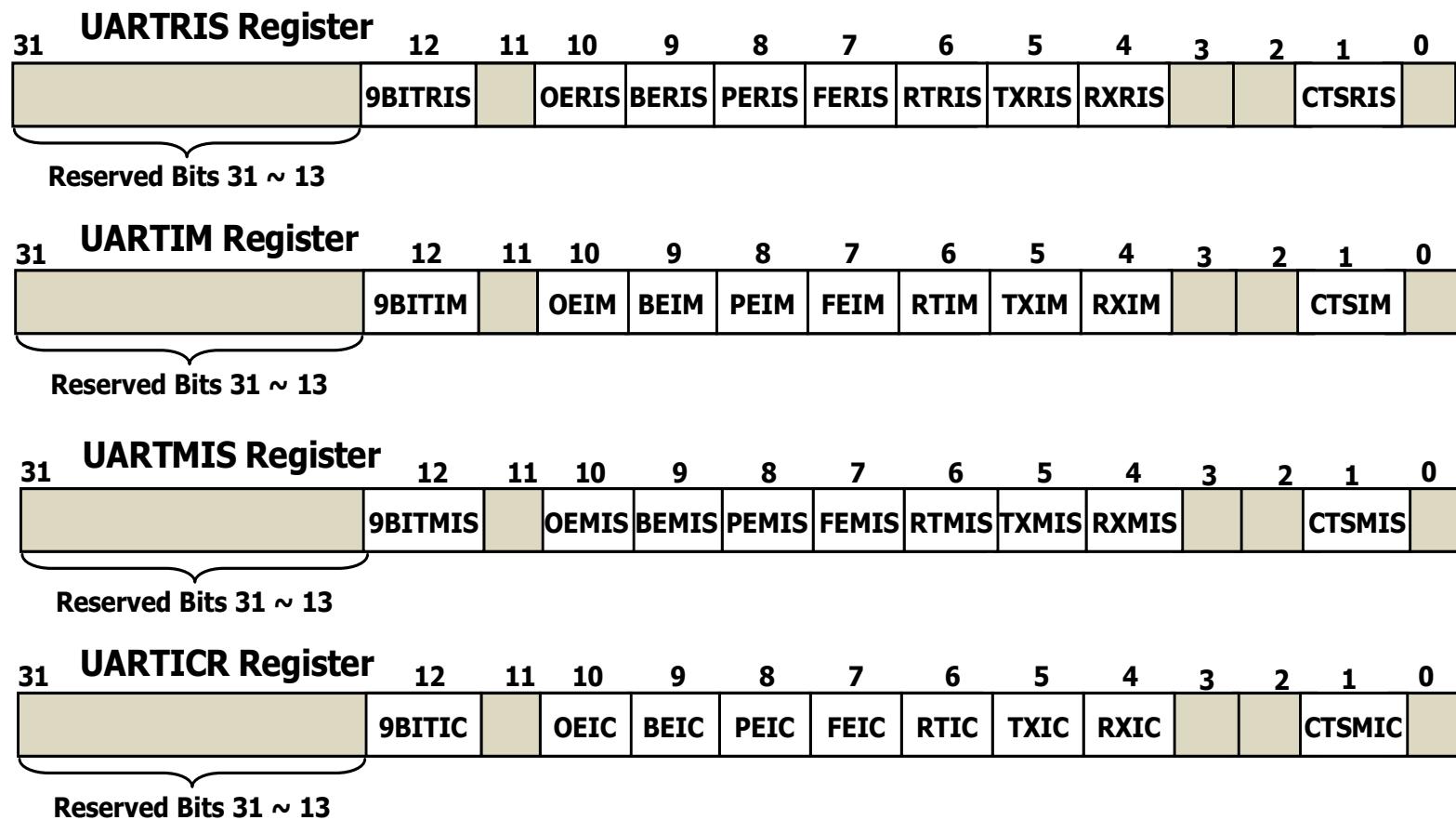


Figure 8.71 Bit configurations for UARTRIS, UARTIM, UARTMIS and UARTICR.

### 8.5.3.9.6 UART DMA CONTROL (UARTDMACTL) REGISTER

- This is a 32-bit register but only the **lowest 3 bits** are used to control the DMA operations on the UART. The bit configurations of this register are shown in Figure 8.72. The bit field and function are shown in Table 8.35.

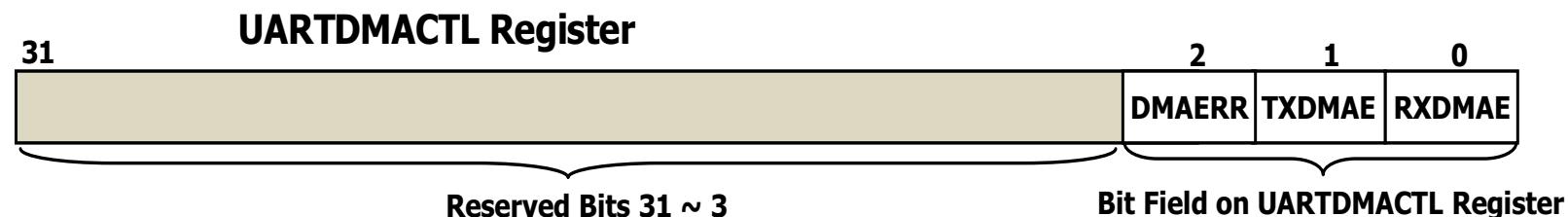


Figure 8.72 Bit configuration on UARTDAMCTL register.

Table 8.35 Bit field value and its function for the UARTDMACTL register.

Bit	Name	Reset	Function
31:3	Reserved	0x000	Reserved
2	DMAERR	0x0	DMA Error. <b>0:</b> μDMA receiving requests are unaffected when a receive error occurs. <b>1:</b> μDMA receiving requests are automatically disabled when a receive error occurs.
1	TXDMAE	0x0	Transmit DMA Enable. <b>0:</b> μDMA for the transmit FIFO is disabled. <b>1:</b> μDMA for the transmit FIFO is enabled.
0	RXDMAE	0x0	Receive DMA Enable. <b>0:</b> μDMA for the receive FIFO is disabled. <b>1:</b> μDMA for the receive FIFO is enabled.

## 8.5.4 UART MODULE CONTROL SIGNALS AND RELATED GPIO PINS

- All eight (**8**) UART modules in the TM4C123GH6PM MCU system use the related GPIO pins to perform UART functions, such as transmit and receiving operations.
- The UART signals are **alternate functions** for related GPIO signals and default to be GPIO signals at reset, with the exception of the **UoRx** and **UoTx** pins which default to the UART function.
- Table 8.36 (next slide) lists these GPIO pins when they work as alternate functions.
- The column titled **GPIO Pin** lists the possible GPIO pin placements for these UART signals.
- The **AFSEL** bit field in the **GPIOAFSEL** register should be set to **1** to choose the UART function. The number in parentheses is the encoding that must be programmed into the **PMCx** field in the **GPIOPCTL** register to assign the UART signal to the specified GPIO port pin.



## 8.5.4 UART MODULE CONTROL SIGNALS AND RELATED GPIO PINS

Table 8.36 UART control signals and the related GPIO pins distributions

<b>UART Pin</b>	<b>GPIO Pin</b>	<b>Pin Type</b>	<b>Buffer Type</b>	<b>Pin Function</b>
<b>U0RX</b>	<b>PA0 (1)</b>	I	TTL	UART Module 0 Receive.
<b>U0TX</b>	<b>PA1 (1)</b>	O	TTL	UART Module 0 Transmit.
<b>U1CTS</b>	<b>PC5 (8) PF1 (1)</b>	I	TTL	UART Module 1 Clear To Send.
<b>U1RTS</b>	<b>PC4 (8) PF0 (1)</b>	O	TTL	UART Module 1 Request To Send.
<b>U1RX</b>	<b>PC4 (2) PB0 (1)</b>	I	TTL	UART Module 1 Receive.
<b>U1TX</b>	<b>PC5 (2) PB1 (1)</b>	O	TTL	UART Module 1 Transmit.
<b>U2RX</b>	<b>PD6 (1)</b>	I	TTL	UART Module 2 Receive..
<b>U2TX</b>	<b>PD7 (1)</b>	O	TTL	UART Module 2 Transmit.
<b>U3RX</b>	<b>PC6 (1)</b>	I	TTL	UART Module 3 Receive..
<b>U3TX</b>	<b>PC7 (1)</b>	O	TTL	UART Module 3 Transmit.
<b>U4RX</b>	<b>PC4 (1)</b>	I	TTL	UART Module 4 Receive..
<b>U4TX</b>	<b>PC5 (1)</b>	O	TTL	UART Module 4 Transmit.
<b>U5RX</b>	<b>PE4 (1)</b>	I	TTL	UART Module 5 Receive..
<b>U5TX</b>	<b>PE5 (1)</b>	O	TTL	UART Module 5 Transmit.
<b>U6RX</b>	<b>PD4 (1)</b>	I	TTL	UART Module 6 Receive..
<b>U6TX</b>	<b>PD5 (1)</b>	O	TTL	UART Module 6 Transmit.
<b>U7RX</b>	<b>PE0 (1)</b>	I	TTL	UART Module 7 Receive..
<b>U7TX</b>	<b>PE1 (1)</b>	O	TTL	UART Module 7 Transmit.

## 8.5.5 UART MODULE INITIALIZATIONS AND CONFIGURATIONS

- Before any UART module can perform data operations, either transmit or receive, the selected UART module must be **initialized** and **configured** properly to enable the module to perform its normal data operations.
- This initialization and configuration job can be divided into the following **three** parts based on their functions on different peripherals:
  - Initialize and configure the **UART related GPIO ports** and **pins**.
  - Initialize and configure the **clock source** and **baud rate** for the UART module.
  - Initialize and configure **UART module**.
- Let's use an example to illustrate how to initialize and configure a UART module with an assumption of using a default **16 MHz** system clock as the clock source for the **UART module 1**.



### 8.5.5.1 INITIALIZE AND CONFIGURE THE UART RELATED GPIO PORTS AND PINS

- Perform the following operations to complete the configurations for the UART related GPIO Ports and GPIO pins:
  - **Enable** and **clock** the appropriate GPIO module via the **RCGCGPIO** register. Here since we use **UART module 1**, enable and clock the **GPIO Port B** (see Table 8.36).
  - Set the **AFSEL** bits on the **GPIOB\_AFSEL** register for the appropriate pins. Here since we use **PBo** and **PB1**, set **0x03** to this register to select and set **PBo** and **PB1**.
  - Configure the **PMCx** fields in the **GPIO\_PCTL** register to assign the UART signals to the appropriate pins. Here since **PB1** and **PBo** are used for **U1TX** and **U1RX** pins, a **0x00000011** should be set to the **GPIO\_PCTL** register (see number in the parentheses on Table 8.36).
  - Set GPIO Port B, exactly **PB1** and **PBo**, as digital function pins.



## 8.5.5.2 INITIALIZE & CONFIGURE CLOCK SOURCE & BAUD RATE FOR UART MODULE

- Perform the following operations to complete these configurations:
  - Enable the clock for the UART module using the **RCGCUART** register.
  - The following asynchronous communication parameters are selected and used for this UART module 1 operations:
    - **115200 baud rate.**                   **8-bit data length.**
    - **One Stop bit.**                          **No parity.**
    - **FIFOs disabled.**                        **No interrupts.**
  - Use the following equation (see 8.5.3.7) to calculate the Baud Rate Divisor (**BRD**):
    - **BRD = 16,000,000 / (16 × 115,200) = 8.6806**
  - This means that the **DIVINT** field on the **UARTIBRD** register should be set to **8** decimal or **0x8**. The value to be loaded into the **UARTFBRD** register is calculated (see 8.5.3.7) by the equation:
    - **UARTFBRD[DIVFRAC] = Integer (0.6808 \* 64 + 0.5) = 44**
  - This decimal value **44** (**0x2C**) should be written to the **UARTFBDR** register as the fractional divisor.

### 8.5.5.3 INITIALIZE AND CONFIGURE THE UART MODULE

- Perform the following operations to complete this configuration job for this UART module:
  - **Disable** UART by clearing the **UARTEN** bit (bit 0) in the **UARTCTL** register.
  - Write the integer portion of the **BRD (0x8)** to the **UARTIBRD** register.
  - Write the fractional portion of the **BRD (0x2C)** to the **UARTFBRD** register.
  - Write the desired serial communication parameters to the **UARTLCRH** register. In this case, a value of **0x0000.0060** should be written into this register.
  - Setup **0x00** to the **UARTCC** register to select the **System Clock** as the clock source.
  - Optionally, configure the  $\mu$ DMA channel and enable the DMA options in the **UARTDMACTL** register if the DMA operation is used.
  - **Enable** UART module 1 by setting the **UARTEN** bit in the **UARTCTL** register.
- Let's build an example UART project to illustrate how to use this peripheral to communicate a PC to transmit and receive characters asynchronously.



## 8.5.6 BUILD AN EXAMPLE UART MODULE PROJECT

- In this section, we build an example UART project to **echo** each character sent via the UART transmitter. This project can be considered as a self-testing project to check the UART transmit and receive functions.
- In order to do this project, we need to perform a loop back testing for the UART **Module 1**. The hardware setups and software features for this project include the following points:
  1. In the EduBASE ARM® Trainer, GPIO **PB1** and **PBo** work as alternate function pins with **PB1** working as the **U1TX** pin and **PBo** as the **U1RX** pin (Table 8.36).
  2. To make the UART **Module 1** to work as a loop back testing mode, we need to connect **U1TX (PB1)** and **U1RX (PBo)** pins together with a wire.
  3. Make this connection by inserting a jumper wire in the **PB1** and **PBo** holes in connector **J10** in the EduBASE ARM® Trainer. After this connection, the **transmit UARTDR** is connected with the **receive UARTDR** and the transmitted data can be fed into the receive UARTDR.



## 8.5.6 BUILD AN EXAMPLE UART MODULE PROJECT

4. In our software code, the **FIFOs** are disabled and therefore only the single byte data can be sent to the **transmit UARTDR** and fed into the **receive UARTDR** because of the connection wire we made in step 2.
  5. In order to transmit a byte data, first we must confirm that the transmitter is **empty** and is ready to accept any data. This can be done by checking the **TXFF** bit (bit **5**) in the **UARTFR** register. A value of **1** on this bit indicates that the transmitter is full, and a **0** indicates that the transmitter is empty and it can accept new data item.
  6. A **while()** loop can be used to fulfill this checking job.
  7. In order to receive a valid data item, we also need to check whether the receiver is **empty** or not. This can be done by checking the **RXFE** bit (bit **4**) in the **UARTFR** register. A value of **0** on this bit means that the receiver is not empty and it contained a valid data item, but a **1** indicates that the receiver is empty with no any data item is available. A **while()** loop can also be used to fulfill this checking job.
- Keep all these points in mind, now let's create a new UART project named **DRAUART**.

## 8.5.6.1 CREATE A NEW UART MODULE PROJECT DRAUART

- Perform the following operations to create a new project **DRAUART**:
  1. Open the Windows Explorer to create a new folder named **DRAUART** under the **C:\ARM Class Projects\Chapter 8** folder.
  2. Open the Keil® MDK µVersion5 and go to **Project|New µVersion Project** menu item to create a new µVersion Project. On the opened wizard, browse to our new folder **DRAUART** that is created in step 1 above. Enter **DRAUART** into the **File name** box and click on the **Save** button to create this project.
  3. On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series** and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
  4. Next the Software Components wizard is opened, and you need to setup the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, and click on the **OK** button since we need these two components to build our project.
- Since this project is simple, therefore only a C source file is good enough.

## 8.5.6.2 CREATE A NEW C SOURCE FILE

- Create a new C Source File **DRAUART.c** in the new created project. Enter the codes shown in Figure 8.73 (next slide) into the source file.
- Some key points about this piece of codes are:
  - Three user defined subroutines, **UART\_Init()**, **UART\_Transmit()** and **UART\_Receive()**, are declared in lines 7 ~ 9. These are used to initialize, transmit and receive the UART data.
  - A **for()** loop is used in line 15 to continuously transmit **26** uppercase letters from **A** to **Z** via **UART\_Transmit()** subroutine and the **U1TX** pin.
  - In line 18, the subroutine **UART\_Receive()** is executed to receive the uppercase letter from the **receive** **UARTDR** register that is connected to the **transmit** **UARTDR** register. The received letter is sent to the array **chr[]** to be stored.
  - The index of the array **num** is updated by **1** in line 19.
  - In line 21, an infinitive **while()** loop is executed to keep the project running to enable us to check the received letters via **Call Stack + Locals** windows later.



## 8.5.6.2 CREATE A NEW C SOURCE FILE

```
1 //*****
2 // DRAUART.c - Main Application File for UART Module 1 Self-Loop Test (Part I)
3 //*****
4 #include <stdint.h>
5 #include <stdbool.h>
6 #include "TM4C123GH6PM.h"
7
8 void UART_Init(void);
9 void UART_Transmit(char sdata);
10 char UART_Receive(void);
11
12 int main(void)
13 {
14     uint8_t num = 0;
15     char s_data, chr[26];
16
17     UART_Init();
18     for(s_data = 'A'; s_data <= 'Z'; s_data = s_data + 1)
19     {
20         UART_Transmit(s_data);           // transmit a byte of data
21         chr[num] = UART_Receive();      // receive a byte of data
22         num++;
23     }
24     while(1);                         // keep program running...
25
26 void UART_Init(void)
27 {
28     SYSCTL->RCGCGPIO |= 0x02|0x20;    // enable clock to GPIOB & GPIOF
29     SYSCTL->RCGCUART = 0x02;          // enable clock to UART1
```

Figure 8.73 The codes for the source file DRAUART.c.

## 8.5.6.2 CREATE A NEW C SOURCE FILE

```
1 //*****  
2 // DRAUART.c - Main Application File for UART Module 1 Self-Loop Test (Part II)  
3 //*****  
  
27 GPIOB->AFSEL = 0x03; // PB1 & PB0 for U1TX & U1RX  
28 GPIOB->PCTL = 0x00000011;  
29 GPIOB->DEN = 0x03; // PB1 & PB0 as digital pins  
30 GPIOF->DEN = 0xF; // PF3 ~ PF0 as digital and output pins  
31 GPIOF->DIR = 0xF;  
  
32 UART1->CTL &= 0xFFFFFFF; // disable UART1  
33 UART1->IBRD = 0x8; // set the integer baud rate period (IBRD = 8)  
34 UART1->FBRD = 0x2C; // set fractional baud rate period (FBRD = 44)  
35 UART1->LCRH = 0x60; // set baud rate = 115200,8-bit data, 1 stop, no parity  
36 UART1->CC = 0x0; // set UARTCC as 00 to select System Clock as source  
37 UART1->CTL |= 0x1; // enable UART1  
38 }  
  
39 void UART_Transmit(char sdata)  
40 {  
41     while ((UART1->FR & 0x20) != 0);  
42     UART1->DR = sdata;  
43 }  
  
44 char UART_Receive(void)  
45 {  
46     uint32_t ret;  
47     char rdata;  
48     while((UART1->FR & 0x10) != 0);  
49     ret = UART1->DR;  
50     if (ret & 0xF00) { GPIOF->DATA = 0xF; } // error occurred...  
51     else { rdata = (char)(UART1->DR & 0xFF); }  
52     return rdata;  
}
```

Figure 8.73 The codes for the source file DRAUART.c.

### 8.5.6.3 SETUP THE ENVIRONMENT TO BUILD AND RUN THE PROJECT

- Before the project can be built, make sure that the following issue has been setup correctly:
  - The debug adapter used for this project is **Stellaris ICDI**. This adapter can be configured in the **Debug** tab under the menu item **Project|Options for Target ‘Target 1’**.
- Now go to the **Project|Build target** menu item to build the project. Then go to the menu item **Flash|Download** to download the project image file into the flash memory. Run the project by going to **Debug|Start/Stop Debug Session** and **Run** menu items.
- After the project runs, go to **Debug|Stop** menu item to stop the project.
- Then open the **Call Stack + Locals** windows and expand the **chr** variable.
- You can find that all 26 uppercase letters have been transmitted and received as shown in Figure 8.74 (next slide).



### 8.5.6.3 SETUP THE ENVIRONMENT TO BUILD AND RUN THE PROJECT

Name	Location/Value	Type
main	0x000003A8	int f()
num	<not in scope>	auto - unsigned char
s_data	<not in scope>	auto - char
chr	0x2000024C "ABCDEF..."	auto - char[26]
[0]	0x41 'A'	char
[1]	0x42 'B'	char
[2]	0x43 'C'	char
[3]	0x44 'D'	char
[4]	0x45 'E'	char
[5]	0x46 'F'	char
[6]	0x47 'G'	char
[7]	0x48 'H'	char
[8]	0x49 'I'	char
[9]	0x4A 'J'	char

Figure 8.74 Running result of the project DRAUART.

## 8.5.7 UART API FUNCTIONS PROVIDED BY TIVAWARE™ PERIPHERAL DRIVER LIBRARY

- There are more than **40** UART API functions provided by this library. In this section, we introduce and discuss some popular and important API functions.
- The UART API function set can be divided into **four** groups of functions: configure and control the UART modules, send and receive data, and deal with interrupt handling.
  1. The clock source for the baud rate generator is handled by:
    - **UARTClockSourceSet()**
    - **UARTClockSourceGet()**
  2. Configuring and control the UART can be handled by:
    - **UARTConfigSetExpClk()**
    - **UARTEnable()**
    - **UARTDisable()**
    - **UARTParityModeSet()**
    - **UARTDMAEnable()**
    - **UARTDMADisable()**



## 8.5.7 UART API FUNCTIONS PROVIDED BY TIVAWARE™ PERIPHERAL DRIVER LIBRARY

3. Sending and receiving data via the UART can be handled by:

- **UARTCharPut()** and **UARTCharGet()**
- **UARTCharPutNonBlocking()**
- **UARTCharGetNonBlocking()**
- **UARTCharsAvail()**
- **UARTSpaceAvail()**
- **UARTBreakCtl()**

4. Managing the UART interrupts can be handled by:

- **UARTIntRegister()**
- **UARTIntUnregister()**
- **UARTIntEnable()**
- **UARTIntDisable()**
- **UARTIntStatus()**
- **UARTIntClear()** and **UARTFIFOLevelSet()**



## 8.5.7.1 CLOCK SOURCE FOR THE BAUD RATE GENERATOR API FUNCTIONS

- Two API functions are included in this group and Table 8.37 shows the function body and arguments of these two functions.
- The function **UARTClockSourceSet()** selects the clock source for the selected UART module, either **UART\_CLOCK\_SYSTEM** or **UART\_CLOCK\_PIOSC**.
- The **UARTClockSourceGet()** is used to get back the current clock source for the selected UART module. If a default clock source is used, it is unnecessary to use these functions. You can use the **SysCtlClockGet()** function to get the clock.

Table 8.37 The clock source for the baud rate generator API functions.

API Function	Parameter	Description
void <b>UARTClockSourceSet(</b> uint32_t ui32Base, uint32_t ui32Source) <b>)</b>	<b>ui32Base</b> is the base address of the UART port.  <b>ui32Source</b> is the baud clock source for the UART.	Select the baud clock source for the specified UART.  Choose the baud clock source for the UART. The possible clock source are the system clock ( <b>UART_CLOCK_SYSTEM</b> ) or the precision internal oscillator ( <b>UART_CLOCK_PIOSC</b> ).
uint32_t <b>UARTClockSourceGet(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the UART port.	Get the baud clock source for the specified UART.  Return the baud clock source for the specified UART. The baud clock source are the system clock ( <b>UART_CLOCK_SYSTEM</b> ) or the precision internal oscillator ( <b>UART_CLOCK_PIOSC</b> ).

## 8.5.7.2 CONFIGURE AND CONTROL THE UART MODULES API FUNCTIONS

- Six popular API functions are involved in this group. Table 8.38 (next slides) shows these functions.
  - **UARTConfigSetExpClk()**
  - **UARTEnable()**
  - **UARTDisable()**
  - **UARTParityModeSet()**
  - **UARTDMAEnable()**
  - **UARTDMADisable()**
- Most popular and important API functions in this group are:
  - **UARTConfigSetExpClk()** and
  - **UARTEnable()**
- since they are directly used to configure and control the UART module.



## 8.5.7.2 CONFIGURE AND CONTROL THE UART MODULES API FUNCTIONS

Table 8.38 The configure and control the UART modules API functions (Part I).

API Function	Parameter	Description
<pre>void UARTConfigSetExpClk(     uint32_t ui32Base,     uint32_t ui32UARTClk,     uint32_t ui32Baud,     uint32_t ui32Config)</pre>	<p><b>ui32Base</b> is the base address of the UART port.</p> <p><b>ui32UARTClk</b> is the rate of the clock supplied to the UART.</p> <p><b>ui32Baud</b> is desired baud rate.</p> <p><b>ui32Config</b> is the data format for the port (number of data bits, number of stop bits, and parity).</p>	<p>Configure the UART for operation in the specified data format. The <b>ui32Config</b> parameter is the logical <b>OR</b> of three values: the number of data bits, the number of stop bits, and the parity. <b>UART_CONFIG_WLEN_8</b>, <b>UART_CONFIG_WLEN_7</b>, <b>UART_CONFIG_WLEN_6</b>, and <b>UART_CONFIG_WLEN_5</b> select from eight to five data bits per byte. <b>UART_CONFIG_STOP_ONE</b> and <b>UART_CONFIG_STOP_TWO</b> select one or two stop bits. <b>UART_CONFIG_PAR_NONE</b>, <b>UART_CONFIG_PAR_EVEN</b>, <b>UART_CONFIG_PAR_ODD</b>, <b>UART_CONFIG_PAR_ONE</b>, and <b>UART_CONFIG_PAR_ZERO</b> select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero).</p> <p>The peripheral clock is the processor clock. The frequency of the system clock is the value returned by <b>SysCtlClockGet()</b>, or it can be explicitly hard coded if it is constant.</p>
<pre>void UARTEnable(     uint32_t ui32Base)</pre>	<p><b>ui32Base</b> is the base address of the UART port.</p>	Enables the UART and its transmit and receive FIFOs.
<pre>void UARTDisable(     uint32_t ui32Base)</pre>	<p><b>ui32Base</b> is the base address of the UART port.</p>	Disable the UART, waits for the end of transmission of the current character, and flushes the transmit FIFO.

## 8.5.7.2 CONFIGURE AND CONTROL THE UART MODULES API FUNCTIONS

Table 8.38 The configure and control the UART modules API functions (Part II).

API Function	Parameter	Description
void <b>UARTParityModeSet(</b> uint32_t ui32Base, uint32_t ui32Parity) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32Parity</b> specifies the type of parity to use	Configure the type of parity to use for transmitting and expect when receiving. The <b>ui32Parity</b> parameter must be one of <b>UART_CONFIG_PAR_NONE</b> , <b>UART_CONFIG_PAR_EVEN</b> , <b>UART_CONFIG_PAR_ODD</b> , <b>UART_CONFIG_PAR_ONE</b> , or <b>UART_CONFIG_PAR_ZERO</b> . The last two parameters allow direct control of the parity bit; it is always either one or zero based on the mode.
void <b>UARTDMAEnable(</b> uint32_t ui32Base, uint32_t ui32DMAFlags) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32DMAFlags</b> is a bit mask of the DMA features to enable.	The UART can be configured to use DMA for transmit or receive and to disable receive if an error occurs. The <b>ui32DMAFlags</b> parameter is the logical <b>OR</b> of any of the following values: <b>UART_DMA_RX</b> - enable DMA for receive; <b>UART_DMA_TX</b> - enable DMA for transmit; <b>UART_DMA_ERR_RXSTOP</b> - disable DMA receive on UART error.
void <b>UARTDMADisable(</b> uint32_t ui32Base, uint32_t ui32DMAFlags) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32DMAFlags</b> is a bit mask of the DMA features to disable.	This function is used to disable UART DMA features that were enabled by <b>UARTDMAEnable()</b> . The specified UART DMA features are disabled. The <b>ui32DMAFlags</b> parameter is the logical <b>OR</b> of any of the following values: <b>UART_DMA_RX</b> - disable DMA for receive; <b>UART_DMA_TX</b> - disable DMA for transmit; <b>UART_DMA_ERR_RXSTOP</b> - do not disable DMA receive on UART error.

### 8.5.7.3 UART SEND AND RECEIVE DATA API FUNCTIONS

- **Seven** popular API functions are involved in this group. Table 8.39 (next slides) shows these functions.
- The difference between the **UARTCharPut()** & **UARTCharPutNonBlocking()** is:
  - The former will wait for a space on the transmit FIFO to be available and then put the character into the transmit FIFO,
  - The latter will not do this waiting and just put the character into the transmit FIFO if a space is available, and return to the main program without doing anything if no space is available.
- Similar case is true to the **UARTCharGet()** & **UARTCharGetNonBlocking()** functions.
- Regularly, the **UARTSpaceAvail()** and **UARTCharsAvail()** functions should be called before sending and receiving data for the selected UART modules.



### 8.5.7.3 UART SEND AND RECEIVE DATA API FUNCTIONS

Table 8.39 The UART send and receive data API functions (Part I).

API Function	Parameter	Description
void <b>UARTCharPut</b> ( uint32_t ui32Base, unsigned char ucData)	<b>ui32Base</b> is the base address of the UART port. <b>ucData</b> is the character to be transmitted.	Send the character <b>ucData</b> to the transmit FIFO for the specified port. If there is no space available in the transmit FIFO, this function waits until there is space available before returning.
int32_t <b>UARTCharGet</b> ( uint32_t ui32Base)	<b>ui32Base</b> is the base address of the UART port.	Get a character from the receive FIFO for the specified port. If there are no characters available, this function waits until a character is received before returning. Return the character read from the port, cast as a <b>int32_t</b> .
bool <b>UARTCharPutNonBlocking</b> ( uint32_t ui32Base, unsigned char ucData)	<b>ui32Base</b> is the base address of the UART port. <b>ucData</b> is the character to be transmitted.	Write the character <b>ucData</b> to the transmit FIFO for the specified port. This function does not block, so if there is no space available, then a <b>false</b> is returned and the application must retry the function later. Returns <b>true</b> if the character was successfully placed in the transmit FIFO or <b>false</b> if there was no space available in the transmit FIFO.

### 8.5.7.3 UART SEND AND RECEIVE DATA API FUNCTIONS

Table 8.39 The UART send and receive data API functions (Part II).

API Function	Parameter	Description
int32_t <b>UARTCharGetNonBlocking(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the UART port.	Receive a character from the specified port. Return the character read from the specified port, cast as a <b>int32_t</b> . This function does not block, so if there is no character present in the receive FIFO, then <b>-1</b> is returned. The function <b>UARTCharsAvail()</b> should be called before attempting to call this function to get data.
bool <b>UARTCharsAvail(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the UART port.	Return a flag indicating whether or not there is data available in the receive FIFO. Returns <b>true</b> if there is data in the receive FIFO or <b>false</b> if there is no data in the receive FIFO.
bool <b>UARTSpaceAvail(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the UART port.	Return a flag indicating whether or not there is space available in the transmit FIFO. Returns <b>true</b> if there is space available in the transmit FIFO or <b>false</b> if no space available in the transmit FIFO.
void <b>UARTBreakCtl(</b> uint32_t ui32Base, bool bBreakState) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>bBreakState</b> controls the output level.	Calling this function with <b>bBreakState</b> set to <b>true</b> asserts a break condition on the UART. Calling this function with <b>bBreakState</b> set to <b>false</b> removes the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

#### 8.5.7.4 UART INTERRUPT HANDLING API FUNCTIONS

- **Seven** popular API functions are involved in this group. Table 8.40 (next slides) shows these functions.
- The operational sequence of enable and response to an interrupt is:
  - Use the **UARTIntRegister()** function to first register the interrupt's handler.
  - Use the **UARTFIFOLevelSet()** function to setup the level at which the transmit and receive interrupts to be triggered. This function is important and must be correctly configured. To use single byte, either transmit or receive, to trigger an interrupt, the **UART\_FIFO\_TX1\_8** and **UART\_FIFO\_RX1\_8** should be used.
  - Use the **UARTIntEnable()** function to enable the desired UART interrupt sources.
  - When any interrupt occurred, use **UARTIntStatus()** function to check and identify the UART interrupt sources.
  - Use the **UARTIntClear()** function to clear the processed interrupt inside the interrupt handler.
- Because of the space limitation, we will discuss a UART loop back project with the receiving interrupt driven fashion in the project lab **Lab8\_6**, and we leave this as a part of home work.

## 8.5.7.4 UART INTERRUPT HANDLING API FUNCTIONS

Table 8.40 The UART interrupt handling API functions (Part I).

API Function	Parameter	Description
void <b>UARTIntRegister(</b> uint32_t ui32Base, void (*pfnHandler)(void) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>pfnHandler</b> is a pointer to the interrupt handler.	Register an interrupt handler for a UART interrupt. Enable the global interrupt in the interrupt controller; specific UART interrupts must be enabled via <b>UARTIntEnable()</b> . It is the interrupt handler's duty to clear the interrupt source.
void <b>UARTIntUnregister(</b> uint32_t ui32Base) <b>)</b>	<b>ui32Base</b> is the base address of the UART port.	Unregister an interrupt handler for a UART interrupt. Clear the handler to be called when a UART interrupt occurs. This function also masks off the interrupt in the interrupt controller so that the interrupt handler no longer is called.
void <b>UARTIntEnable(</b> uint32_t ui32Base, uint32_t ui32IntFlags) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32IntFlags</b> is the bit mask of the interrupt sources to be enabled.	Enables individual UART interrupt sources. The <b>ui32IntFlags</b> is the logical <b>OR</b> of any of the following: <b>UART_INT_9BIT</b> - 9-bit Address Match interrupt. <b>UART_INT_OE</b> - Overrun Error interrupt. <b>UART_INT_BE</b> - Break Error interrupt. <b>UART_INT_PE</b> - Parity Error interrupt. <b>UART_INT_FE</b> - Framing Error interrupt. <b>UART_INT_RT</b> - Receive Timeout interrupt. <b>UART_INT_TX</b> - Transmit interrupt. <b>UART_INT_RX</b> - Receive interrupt. <b>UART_INT_DSR</b> - DSR interrupt. <b>UART_INT_DCD</b> - DCD interrupt. <b>UART_INT_CTS</b> - CTS interrupt. <b>UART_INT_RI</b> - RI interrupt.

## 8.5.7.4 UART INTERRUPT HANDLING API FUNCTIONS

Table 8.40 The UART interrupt handling API functions (Part II).

API Function	Parameter	Description
void <b>UARTIntDisable(</b> uint32_t ui32Base, uint32_t ui32IntFlags) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32IntFlags</b> is the bit mask of the interrupt sources to be disabled.	Disable the indicated UART interrupt sources. Only the sources that are enabled can be reflected to the processor interrupts. The <b>ui32IntFlags</b> parameter has the same definition as the <b>ui32IntFlags</b> parameter to <b>UARTIntEnable()</b> .
uint32_t <b>UARTIntStatus(</b> uint32_t ui32Base, bool bMasked) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>bMasked</b> is <b>false</b> if the raw interrupt status is required and <b>true</b> if the masked interrupt status is required.	Return the interrupt status for the specified UART. Either the raw interrupt status or the status of interrupts that are allowed to reflect to the processor can be returned. Return the current interrupt status, enumerated as a bit field of values described in <b>UARTIntEnable()</b> .
void <b>UARTIntClear(</b> uint32_t ui32Base, uint32_t ui32IntFlags) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32IntFlags</b> is bit mask of interrupt to be cleared.	The specified UART interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit. The <b>ui32IntFlags</b> parameter has the same definition as the <b>ui32IntFlags</b> parameter to <b>UARTIntEnable()</b> .
void <b>UARTFIFOLevelSet(</b> uint32_t ui32Base, uint32_t ui32TxLevel, uint32_t ui32RxLevel) <b>)</b>	<b>ui32Base</b> is the base address of the UART port. <b>ui32TxLevel</b> is the transmit FIFO interrupt level, specified as one of <b>UART_FIFO_TX1_8</b> , <b>UART_FIFO_TX2_8</b> , <b>UART_FIFO_TX4_8</b> , <b>UART_FIFO_TX6_8</b> , or <b>UART_FIFO_TX7_8</b> . <b>ui32RxLevel</b> is the receive FIFO interrupt level, specified as one of <b>UART_FIFO_RX1_8</b> , <b>UART_FIFO_RX2_8</b> , <b>UART_FIFO_RX4_8</b> , <b>UART_FIFO_RX6_8</b> , or <b>UART_FIFO_RX7_8</b> .	Set the FIFO level at which interrupts are generated. <b>ui32TxLevel</b> is the transmit FIFO interrupt level, specified as one of <b>UART_FIFO_TX1_8</b> , <b>UART_FIFO_TX2_8</b> , <b>UART_FIFO_TX4_8</b> , <b>UART_FIFO_TX6_8</b> , or <b>UART_FIFO_TX7_8</b> . <b>ui32RxLevel</b> is the receive FIFO interrupt level, specified as one of <b>UART_FIFO_RX1_8</b> , <b>UART_FIFO_RX2_8</b> , <b>UART_FIFO_RX4_8</b> , <b>UART_FIFO_RX6_8</b> , or <b>UART_FIFO_RX7_8</b> .

## 8.6 CHAPTER SUMMARY

- The main topics in this chapter are about the ARM® Cortex®-M4 MCU serial port programming, which include the synchronous and asynchronous serial data communications and operations.
- Because the TM4C123GH6PM MCU contains quite a few peripherals related to serial interface and communications, we only introduced and discussed three major serial peripherals, **SSI**, **I2C** and **UART**. We leave some other serial peripherals to be introduced in the following chapters, such as **USB** and **CANs**.
- We started our discussion from the **Synchronous Serial Interface (SSI)** that is also called **Synchronous Peripheral Interface (SPI)** in this chapter. Three example projects related to SSI are introduced and discussed with line by line illustrations:
  - **On-Board LCD Interface Programming Project**
  - **On-Board 7 Segment LED Interface Programming Project**
  - **Digital to Analog Converter Programming Project**
- These projects are built by using the DRA model and method.



## 8.6 CHAPTER SUMMARY

- Some important and popular API functions provided by the TivaWare™ Peripheral Driver Library are discussed in section 8.3.8. A **SSI** related project using the API function, **SDLCD**, is discussed with details in that part.
- Following the SSI discussions, we began our discussion about the **I2C bus** with a real project to interface the **BQ32000** Real Time Clock via the I2C bus.
- The I2C API functions provided by TivaWare™ Peripheral Driver Library are discussed in section 8.4.11. A lab project **Lab8\_5** is assigned as a part of home work for the readers.
- The UART is discussed at section 8.5. This is an asynchronous serial data communication protocol and widely applied in most data communications. A real project that uses loop back test for the UART is built with the DRA model.
- The API functions provided by TivaWare™ Peripheral Driver Library are discussed in section 8.5.7. An example UART project developed by using the UART API functions is provided as a lab project **Lab8\_6**, which belongs to a part of home work.