PSoC 4200M Specs
I2C
3.5.2 Serial communication blocks (SCB)
The PSoC™ 4200M has four SCBs, which can each implement an I2C, UART, or SPI interface.
I 2C Mode: The hardware I2C block implements a full multi-master and slave interface (it is capable of multimaster arbitration). This block is capable of operating at speeds of up to 1 Mbps (Fast Mode Plus) and has flexible buffering options to reduce interrupt overhead and latency for the CPU. It also supports EzI2C that creates a mailbox address range in the memory of the PSoC™ 4200M and effectively reduces I2C communication to reading from and writing to an array in memory. In addition, the block supports an 8-deep FIFO for receive and transmit which, by increasing the time given for the CPU to read data, greatly reduces the need for clock stretching caused by the CPU not having read data on time. The FIFO mode is available in all channels and is very useful in the absence of DMA.
The I2C peripheral is compatible with the I2C Standard-mode, Fast-mode, and Fast-mode Plus devices as defined in the NXP I2C-bus specification and user manual (UM10204). The I2C bus I/O is implemented with GPIO in open-drain modes.
The Pins of Port 6 (up to 6 depending on the package) are overvoltage tolerant (VIN can exceed VDD). The overvoltage cells will not sink more than 10 µA when their inputs exceed VDDIO in compliance with I2C specifications.

Port/ Pin	Analog	Alt. Function 1	Alt. Function 2	Alt. Function 3	Alt. Function 4
P1.0	ctb0.oa0.inp	tcpwm.line[2]	scb[0].uart_rx:1	–	scb[0].i2c_scl:0
P1.1	ctb0.oa0.inm	tcpwm.line_compl[2]	scb[0].uart_tx:1	–	scb[0].i2c_sda:0
P1.2	ctb0.oa0.out	tcpwm.line[3]	scb[0].uart_cts:1	–	–
P1.3	ctb0.oa1.out	tcpwm.line_compl[3]	scb[0].uart_rts:1	–	–
P2.0	sarmux.0	tcpwm.line[4]	–	–	scb[1].i2c_scl:1	scb[1].spi_mosi:2
P2.1	sarmux.1	tcpwm.line_compl[4]	–	–	scb[1].i2c_sda:1	scb[1].spi_miso:2
P2.2	sarmux.2	tcpwm.line[5]	–	–	–	scb[1].spi_clk:2
P2.3	sarmux.3	tcpwm.line_compl[5]	–	–	–	scb[1].spi_select0:2
P6.0	–	tcpwm.line[4]	scb[3].uart_rx:0	can[0].can_tx_enb_n:0	scb[3].i2c_scl:0	scb[3].spi_mosi:0
P6.1	–	tcpwm.line_compl[4]	scb[3].uart_tx:0	can[0].can_rx:0	scb[3].i2c_sda:0	scb[3].spi_miso:0
P6.2	–	tcpwm.line[5]	scb[3].uart_cts:0	can[0].can_tx:0	–	scb[3].spi_clk:0
P6.3	–	tcpwm.line_compl[5]	scb[3].uart_rts:0	–	–	scb[3].spi_select0:0

P3.0	–	tcpwm.line[0]	scb[1].uart_rx:1	–	scb[1].i2c_scl:2	scb[1].spi_mosi:0
P3.1	–	tcpwm.line_compl[0]	scb[1].uart_tx:1	–	scb[1].i2c_sda:2	scb[1].spi_miso:0
P3.2	–	tcpwm.line[1]	scb[1].uart_cts:1	–	swd_data	scb[1].spi_clk:0
P3.3	–	tcpwm.line_compl[1]	scb[1].uart_rts:1	–	swd_clk	scb[1].spi_select0:0
P4.0	–	–	scb[0].uart_rx:0	can[0].can_rx:1	scb[0].i2c_scl:1	scb[0].spi_mosi:0
P4.1	–	–	scb[0].uart_tx:0	can[0].can_tx:1	scb[0].i2c_sda:1	scb[0].spi_miso:0
P4.2	csd[0].c_mod	–	scb[0].uart_cts:0	can[0].can_tx_enb_n:1	lpcomp.comp[0]:0	scb[0].spi_clk:0
P4.3	csd[0].c_sh_tank	–	scb[0].uart_rts:0	–	lpcomp.comp[1]:0	scb[0].spi_select0:0

Architecture TRM :-

15. Serial Communications Block (SCB)

The Serial Communications Block (SCB) of PSoC® 4 supports three serial interface protocols: SPI, UART, and I2C. Only one of the protocols is supported by an SCB at any given time. PSoC 4 devices have four SCBs. Additional instances of the serial peripheral interface (SPI) and UART protocols can be implemented using the universal digital blocks (UDBs) in PSoC 4200M.

15.1 Features

This block supports the following features:

■ Standard SPI master and slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols

■ Standard UART functionality with SmartCard reader, Local Interconnect Network (LIN), and IrDA protocols

■ Standard I2C master and slave functionality

■ Standard LIN slave functionality with LIN v1.3 and LIN v2.1/2.2 specification compliance

■ EZ mode for SPI and I2C, which allows for operation without CPU intervention

■ Low-power (Deep-Sleep) mode of operation for SPI and I2C protocols (using external clocking)

Each of the three protocols is explained in the following sections.

15. Inter Integrated Circuit (I2C)

This section explains the I2C implementation in PSoC 4. For more information on the I2C protocol specification, refer to the I 2C-bus specification available on the NXP website.

15.4.1 Features

This block supports the following features:

■ Master, slave, and master/slave mode

■ Slow-mode (50 kbps), standard-mode (100 kbps), fast-mode (400 kbps), and fast-mode plus (1000 kbps) data-rates

■ 7- or 10-bit slave addressing (10-bit addressing requires firmware support)

■ Clock stretching and collision detection

■ Programmable oversampling of I2C clock signal (SCL)

■ Error reduction using an digital median filter on the input path of the I2C data signal (SDA)

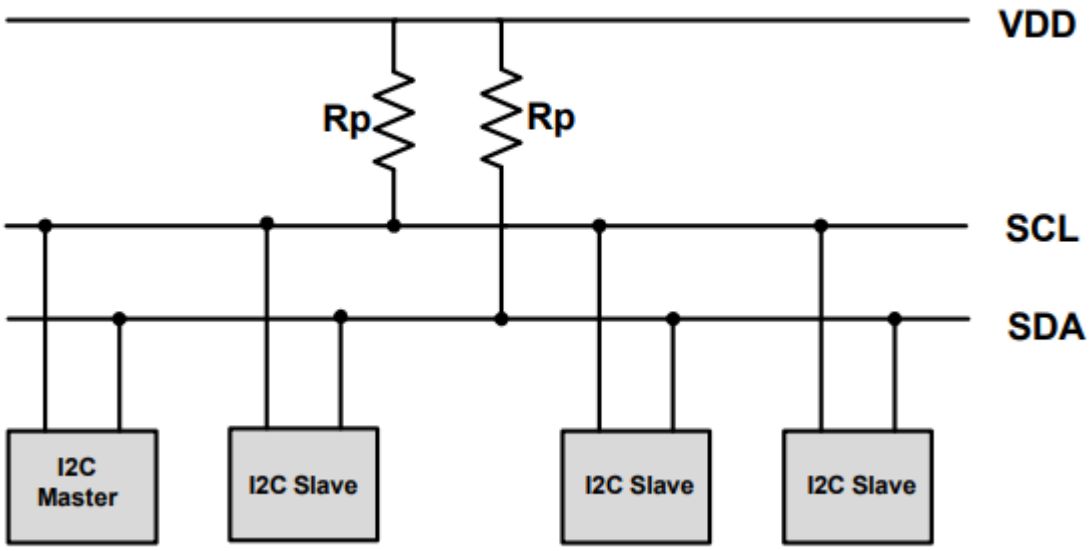
■ Glitch-free signal transmission with an analog glitch filter

■ Interrupt or polling CPU interface

15.4.2 General Description

Figure 15-22 illustrates an example of an I2C communication network

Figure 15-22. I²C Interface Block Diagram



The standard I2C bus is a two wire interface with the following lines:

■ Serial Data (SDA)

■ Serial Clock (SCL)

I2C devices are connected to these lines using open collector or open-drain output stages, with pull-up resistors (Rp). A simple master/slave relationship exists between devices. Masters and slaves can operate as either transmitter or receiver. Each slave device connected to the bus is software addressable by a unique 7-bit address. PSoC 4 also supports 10-bit address matching for I2C with firmware support.

15.4.3 Terms and Definitions

Table 15-14 explains the commonly used terms in an I2C communication network.

Table 15-14. Definition of I2C Bus Terminology

Table 15-14. Definition of I²C Bus Terminology

Term	Description
Transmitter	The device that sends data to the bus
Receiver	The device that receives data from the bus
Master	The device that initiates a transfer, generates clock signals, and terminates a transfer
Slave	The device addressed by a master
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	Procedure to synchronize the clock signals of two or more devices

15.4.3.1 Clock Stretching

When a slave device is not yet ready to process data, it may drive a '0' on the SCL line to hold it down. Due to the implementation of the I/O signal interface, the SCL line value will be '0', independent of the values that any other master or slave may be driving on the SCL line. This is known as clock stretching and is the only situation in which a slave drives the SCL line. The master device monitors the SCL line and detects it when it cannot generate a positive clock pulse ('1') on the SCL line. It then reacts by delaying the generation of a positive edge on the SCL line, effectively synchronizing with the slave device that is stretching the clock.

15.4.3.2 Bus Arbitration

The I2C protocol is a multi-master, multi-slave interface. Bus arbitration is implemented on master devices by monitoring the SDA line. Bus collisions are detected when the master observes an SDA line value that is not the same as the value it is driving on the SDA line. For example, when master 1 is driving the value '1' on the SDA line and master 2 is driving the value '0' on the SDA line, the actual line value will be '0' due to the implementation of the I/O signal interface. Master 1 detects the inconsistency and loses control of the bus. Master 2 does not detect any inconsistency and keeps control of the bus.

15.4.4 I2C Modes of Operation

I2C is a synchronous single master, multi-master, multi-slave serial interface. Devices operate in either master mode, slave mode, or master/slave mode. In master/slave mode, the device switches from master to slave mode when it is addressed. Only a single master may be active during a data transfer. The active master is responsible for driving the clock on the SCL line.

Table 15-15 illustrates the I2C modes of operation.

Table 15-15. I^2C Modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus
Multi-master-slave	Simultaneous slave and multi-master operation

Data transfer through the I2C bus follows a specific format. Table 15-16 lists some common bus events that are part of an I2C data transfer. The Write Transfer and Read Transfer sections explain the I2C bus bit format during data transfer.

Table 15-16. I^2C Bus Events Terminology

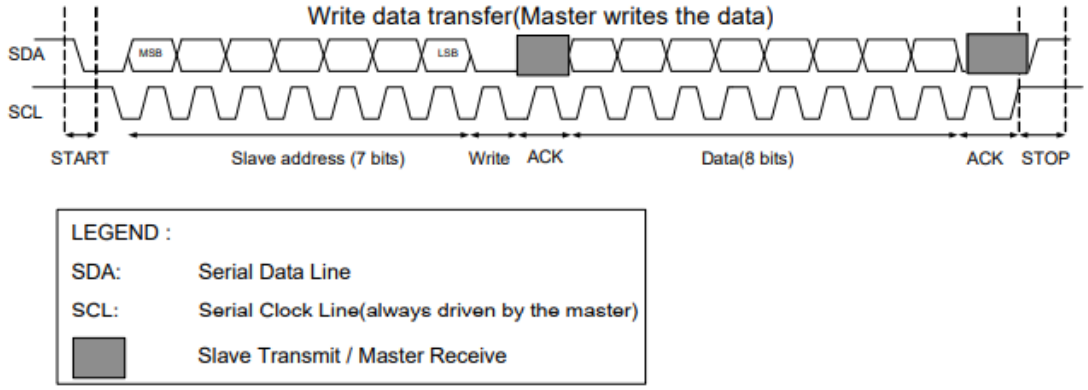
Bus Event	Description
START	A HIGH to LOW transition on the SDA line while SCL is HIGH
STOP	A LOW to HIGH transition on the SDA line while SCL is HIGH
ACK	The receiver pulls the SDA line LOW and it remains LOW during the HIGH period of the clock pulse, after the transmitter transmits each byte. This indicates to the transmitter that the receiver received the byte properly.
NACK	The receiver does not pull the SDA line LOW and it remains HIGH during the HIGH period of clock pulse after the transmitter transmits each byte. This indicates to the transmitter that the receiver received the byte properly.
Repeated START	START condition generated by master at the end of a transfer instead of a STOP condition
DATA	SDA status change while SCL is low (data changing), and no change while SCL is high (data valid)

When operating in multi-master mode, the bus should always be checked to see if it is busy; another master may already be communicating with a slave. In this case, the master must wait until the current operation is complete before issuing a START signal (see Table 15-16, Figure 15-23, and Figure 15-24). The master looks for a STOP signal as an indicator that it can start its data transmission.

When operating in multi-master-slave mode, if the master loses arbitration during data transmission, the hardware reverts to slave mode and the received byte generates a slave address interrupt, so that the device is ready to respond to any other master on the bus. With all of these modes, there are two types of transfer - read and write. In write transfer, the master sends data to slave; in read transfer, the master receives data from slave. Write and read transfer examples are available in “Master Mode Transfer Examples” on page 131, “Slave Mode Transfer Examples” on page 133, and “Multi-Master Mode Transfer Example” on page 137.

15.4.4.1 Write Transfer

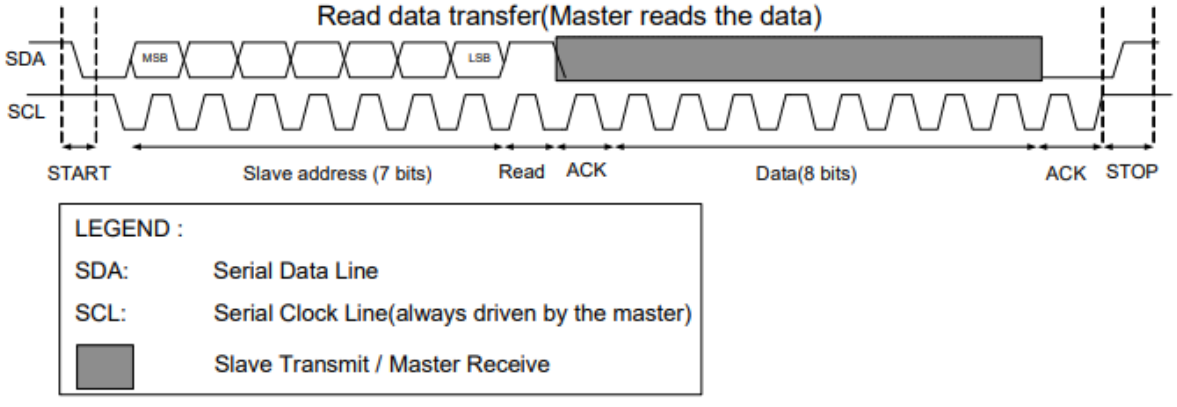
Figure 15-23. Master Write Data Transfer



■ A typical write transfer begins with the master generating a START condition on the I2C bus. The master then writes a 7-bit I2C slave address and a write indicator ('0') after the START condition. The addressed slave transmits an acknowledgement byte by pulling the data line low during the ninth bit time.
■ If the slave address does not match any of the slave devices or if the addressed device does not want to acknowledge the request, it transmits a no acknowledgement (NACK) by not pulling the SDA line low. The absence of an acknowledgement, results in an SDA line value of '1' due to the pull-up resistor implementation.
■ If no acknowledgement is transmitted by the slave, the master may end the write transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt.
■ The master may transmit data to the bus if it receives an acknowledgement. The addressed slave transmits an acknowledgement to confirm the receipt of every byte of data written. Upon receipt of this acknowledgement, the master may transmit another data byte.
■ When the transfer is complete, the master generates a STOP condition.

15.4.4.2 Read Transfer

Figure 15-24. Master Read Data Transfer



■ A typical read transfer begins with the master generating a START condition on the I2C bus. The master then writes a 7-bit I2C slave address and a read indicator ('1') after the START condition. The addressed slave transmits an acknowledgement by pulling the data line low during the ninth bit time.

■ If the slave address does not match with that of the connected slave device or if the addressed device does not want to acknowledge the request, a no acknowledgement (NACK) is transmitted by not pulling the SDA line low. The absence of an acknowledgement, results in an SDA line value of '1' due to the pull-up resistor implementation.
■ If no acknowledgement is transmitted by the slave, the master may end the read transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt.
■ If the slave acknowledges the address, it starts transmitting data after the acknowledgement signal. The master transmits an acknowledgement to confirm the receipt of each data byte sent by the slave. Upon receipt of this acknowledgement, the addressed slave may transmit another data byte.
■ The master can send a NACK signal to the slave to stop the slave from sending data bytes. This completes the read transfer.
■ When the transfer is complete, the master generates a STOP condition.

15.4.5 Easy I2C (EZI2C) Protocol

The Easy I2C (EZI2C) protocol is a unique communication scheme built on top of the I2C protocol by Cypress. It uses a software wrapper around the standard I2C protocol to communicate to an I2C slave using indexed memory transfers. This removes the need for CPU intervention at the level of individual frames.

The EZI2C protocol defines an 8-bit address that indexes a memory array (8-bit wide 32 locations) located on the slave device. Five lower bits of the EZ address are used to address these 32 locations. The number of bytes transferred to or from the EZI2C memory array can be found by comparing the EZ address at the START event and the EZ address at the STOP event.

Note The I2C block has a hardware FIFO memory, which is 16 bits wide and 16 locations deep with byte write enable. The access methods for EZ and non-EZ functions are different. In non-EZ mode, the FIFO is split into TXFIFO and RXFIFO. Each has 16-bit wide eight locations. In EZ mode, the FIFO is used as a single memory unit with 8-bit wide 32 locations.

EZI2C has two types of transfers: a data write from the master to an addressed slave memory location, and a read by the master from an addressed slave memory location.

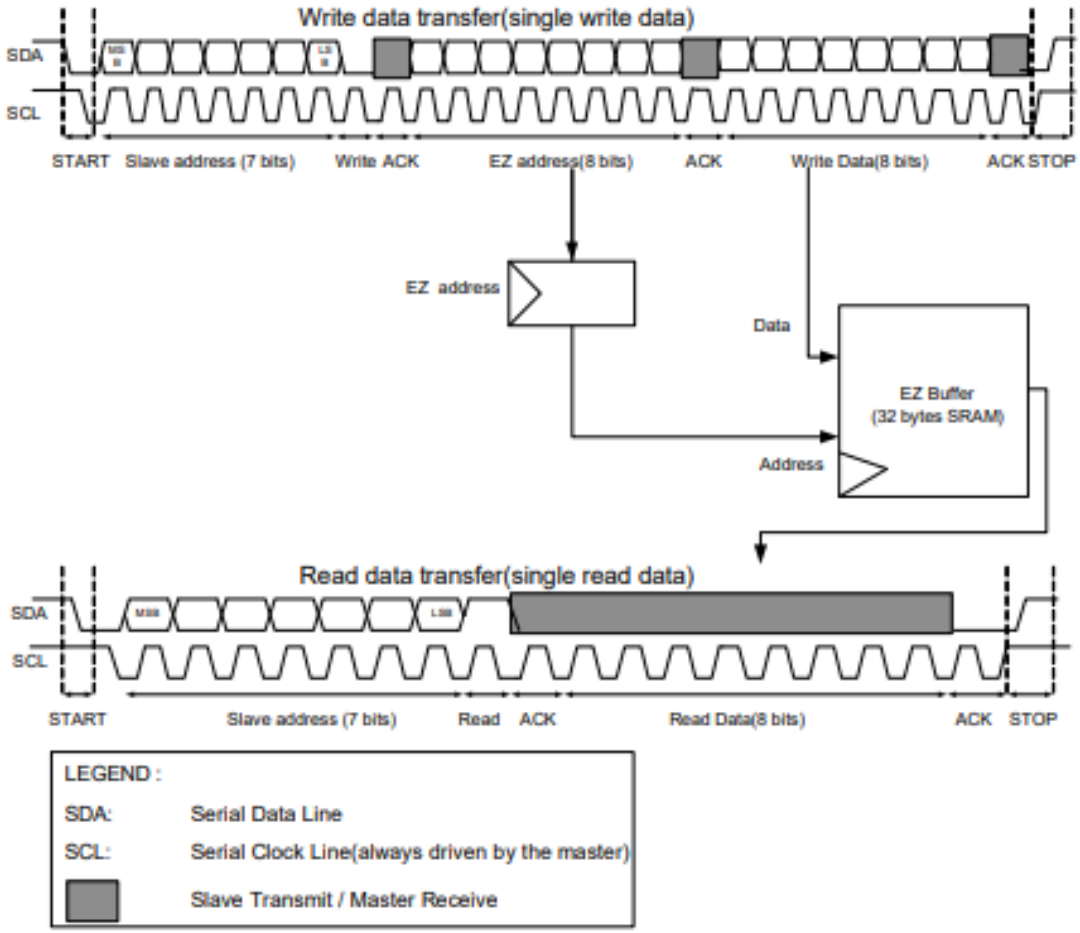
15.4.5.1 Memory Array Write

An EZ write to a memory array index is by means of an I2C write transfer. The first transmitted write data is used to send an EZ address from the master to the slave. The five lowest significant bits of the write data are used as the "new" EZ address at the slave. Any additional write data elements in the write transfer are bytes that are written to the memory array. The EZ address is automatically incremented by the slave as bytes are written into the memory array. If the number of continuous data bytes written to the EZI2C buffer exceeds EZI2C buffer boundary, it overwrites the last location for every subsequent byte.

15.4.5.2 Memory Array Read

An EZ read from a memory array index is by means of an I 2C read transfer. The EZ read relies on an earlier EZ write to have set the EZ address at the slave. The first received read data is the byte from the memory array at the EZ address memory location. The EZ address is automatically incremented as bytes are read from the memory array. The address wraps around to zero when the final memory location is reached.

Figure 15-25. EZI2C Write and Read Data Transfer



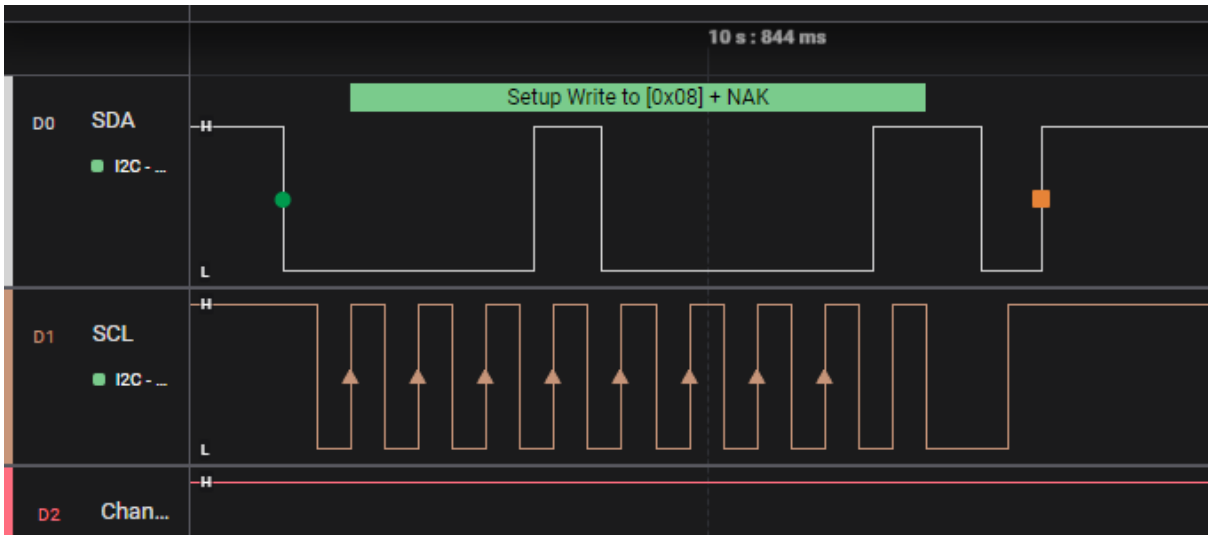

```
/* Interface to internal interrupt component */
#if (I2CS_SCB_IRQ_INTERNAL)
    /**
    * \addtogroup group_interrupt
    * @{
    */

/*******************************************************************************
*****
    * Function Name: I2CS_EnableInt

********************************************************************************
*//**
    *
    *  When using an Internal interrupt, this enables the interrupt in the
NVIC.
    *  When using an external interrupt the API for the interrupt component
must be used to enable the interrupt.
    *

*******************************************************************************
****/
    #define I2CS_EnableInt()     CyIntEnable(I2CS_ISR_NUMBER)
```

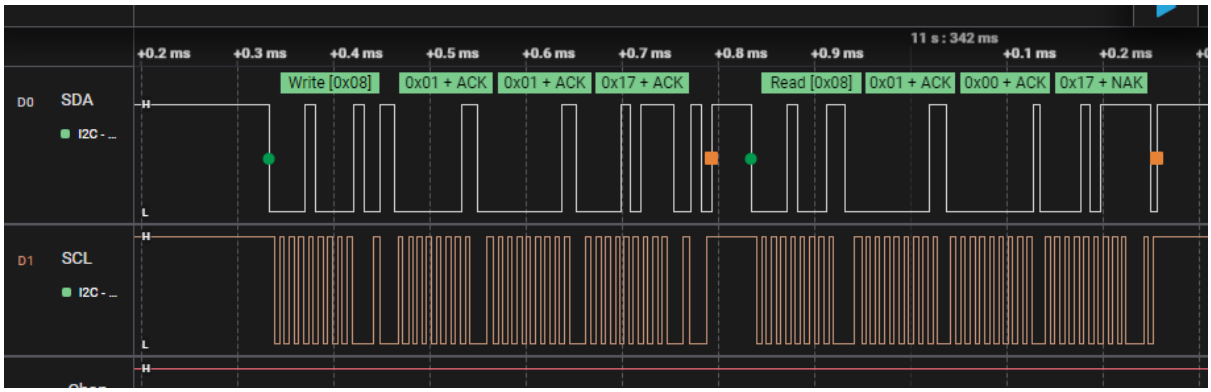
```
main.c | main.h | LED_RED.c | I2CS_I2C.h | I2CS_I2C_SLAVE.c | I2CS.h | I2CS.c | I2CS_I2C.c | cyfitter.h

861
862
863 /***************************************
864 *        Registers Constants
865 ***************************************/
866
867 #if (I2CS_SCB_IRQ_INTERNAL)
868     #define I2CS_ISR_NUMBER     ((uint8) I2CS_SCB_IRQ__INTC_NUMBER)
869     #define I2CS_ISR_PRIORITY   ((uint8) I2CS_SCB_IRQ__INTC_PRIOR_NUM)
870 #endif /* (I2CS_SCB_IRQ_INTERNAL) */
```

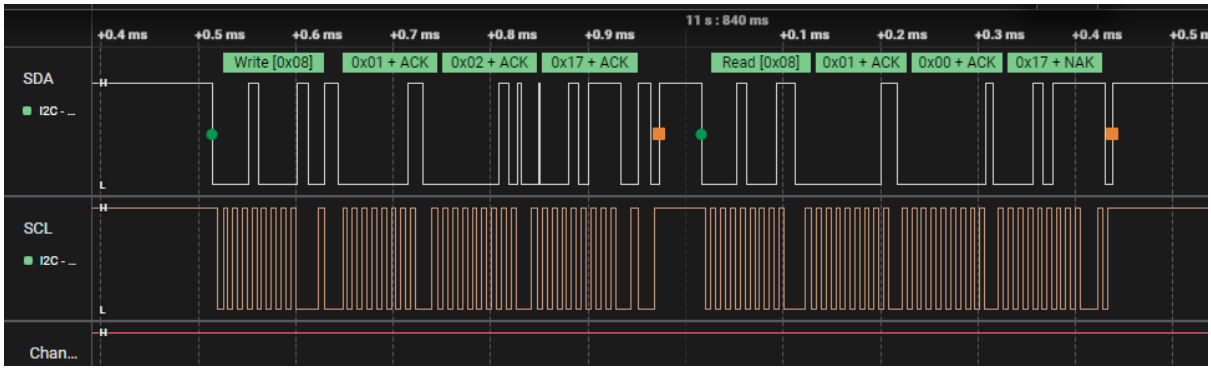
I had two psoc4 boards and
I flashed I2C Master CE code into cy8ckit-042 and flashed I2C slave CE into cy8ckit-044
the result is below: -

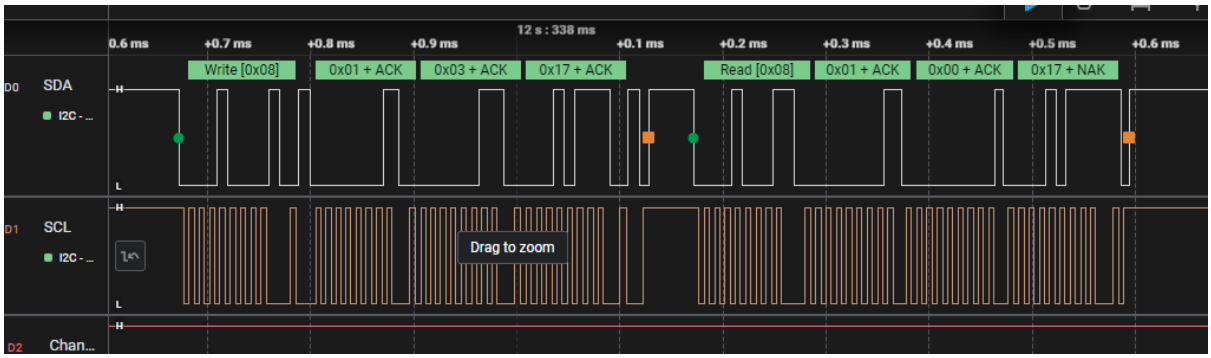
I have followed CE222306 CE which I kept in below folder
C:\Users\sidra\Documents\I2C_learning

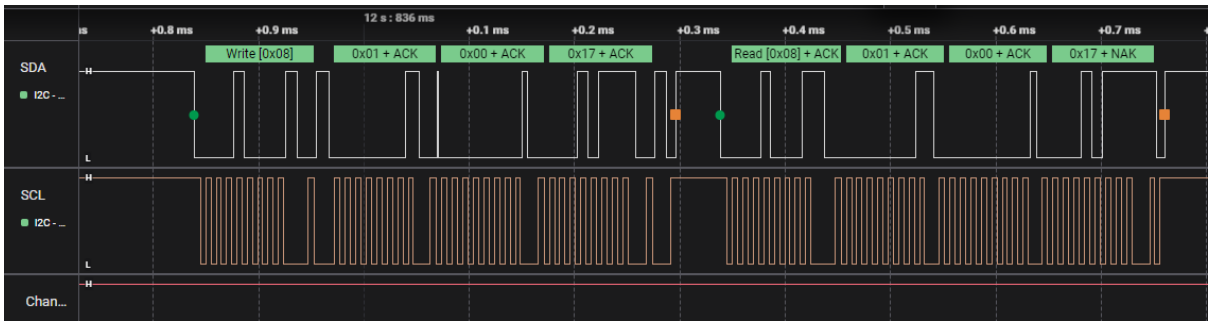
Same goes with architecture trm and SCB register part of register trm also I2C_pdl document

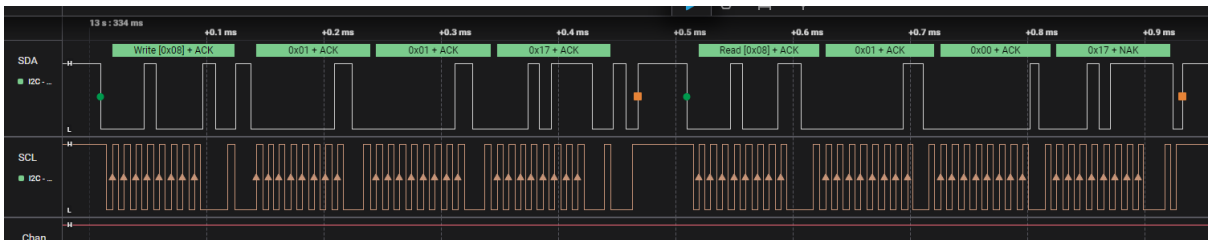












write to 0x08 nak

write to 0x08 ack data: 0x01 0x01 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x02 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x03 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x00 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x01 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x02 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x03 0x17

read to 0x08 ack data: 0x01 0x00 0x17

write to 0x08 ack data: 0x01 0x00 0x17

read to 0x08 ack data: 0x01 0x00 0x17