Application Report
SPMA073–July 2015

Using Feature Set of I2C Master on TM4C129x Microcontrollers

Amit Ashara

ABSTRACT

The inter-integrated circuit (I2C) is a multi-master, multi-slave, single-ended bus that is typically used for attaching lower speed peripheral ICs to processors and microcontrollers. The type of slave devices range from non-volatile memory to data-acquisition devices like analog-to-digital converters (ADC), sensors, and so forth. This application report demonstrates how to use the feature rich I2C master on the TM4C129x microcontrollers to communicate with a host of slave devices in a system.

Project collateral and source code discussed in this document can be downloaded from the following URL: http://www.ti.com/lit/zip/spma073.

Contents

List of Figures

1 Introduction

The TM4C129x family of devices from Texas Instruments integrates up to 10 independent I2C modules with the following features:

- I2C module with independent master and slave blocks on the same bus without requiring additional pins for separate master and slave functions
- Integrated 8-byte deep FIFO for transmit and receive operations that can be independently assigned to either master or slave block
- Efficient transfer mechanism using uDMA and FIFO that reduces CPU overhead during large data transfers
- Independent uDMA channel for Transmit and Receive operations
- Programmable glitch suppression capability in terms of system clocks which meets the standard requirements
- Support for standard, fast, fast plus and high-speed mode through configurable timer register
- Support for arbitration and clock stretching during master mode initiated transactions
- Support for SMBus clock low timeout, dual-slave address and quick command features
- Highly configurable interrupt mechanism to reduce dependency on polling mechanism

2 Overview of I2C Protocol

Before the usage of I2C on the TM4C129x devices is described, it is important that some of the basic concepts of the I2C protocol and bus are described first. This aids the understanding of the protocol and the debugging issues on the physical bus using an oscilloscope or a logic analyzer.

The description is kept concise from the understanding perspective. For more information, see the UM10204: I2C-Bus Specification and User Manual.

2.1 Physical Layer of I2C

The I2C bus consists of two signals: serial clock (SCL) and serial data (SDA). At the IO level, both SCL and SDA are open-drain as shown in Figure 1.

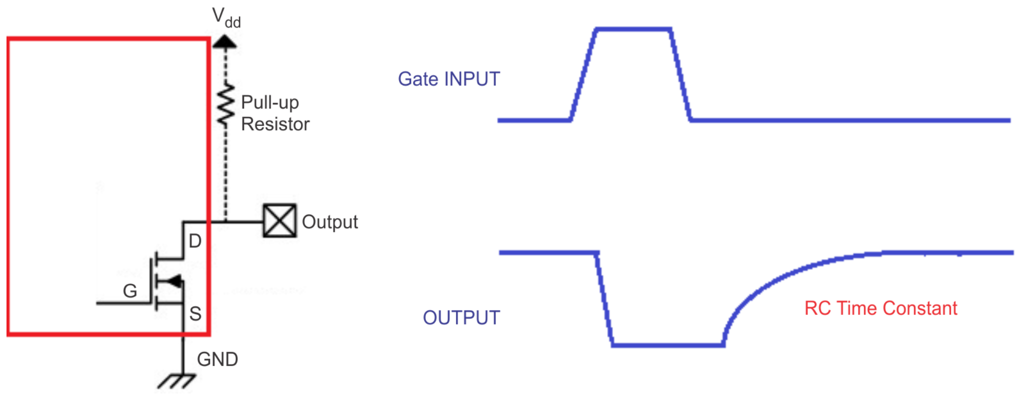


Figure 1. Open Drain Circuit

The output transistor drives low to create a logical 0 on the bus. The output transistor is off to float the pin to create a logical 1 on the bus. Hence, an external pull up to VDD is required. This is required since the bus is bidirectional; if one device on the bus transmits a logical 0 and another device transmits a logical 1, it will not create an electrical contention that could damage the IO. Instead a current path is established from the device transmitting a logical 1 to the device transmitting a logical 0 via the pull up resistor.

The important positive consequence of this are:

- Multiple devices can place a different logical value and not damage the other devices due to excessive current flow. It also has an effect on the workings of the I2C bus itself.
- When this concept is applied to SCL, it provides the slave a mechanism for stretching the clock and establishes a form of flow control.
- When this concept is applied to SDA, it allows the master to detect another master, which is called bus arbitration.

The important negative consequence of this is:

- The rise time of the SCL and SDA are no longer a function of the drive strength but as defined by the RC time constant, where R is the value of the pull-up resistor and C is the value of the parasitic and load capacitance on the bus. This limits the speed of operation.
- The speed of operation requires that the R value be decreased to reduce the RC time constant as parasitic and load capacitance cannot be changed readily, thereby, increasing the current via the R when a logical 0 is applied from a device affecting the current drain on portable power devices.

2.2 Communication Layer of I2C

The I2C bus communicates using the timing of the SCL and SDA; however, there are some concepts that need to be defined first.

- BUS IDLE: The idle state of a bus is defined when both SCL and SDA are high.
- START BIT: The start bit of a I2C frame is defined as SDA driven low while SCL is high.
- STOP BIT: The stop bit of an I2C frame is defined as SDA pulled high while SCL is high.

All bits are signaled between a start and stop bit. Any bit being transmitted by a device (master or slave) is SCL being pulled high and then driven low (pulsing) with the SDA kept at a steady state. It is illegal in I2C for the SDA to change when the SCL is high other than for a START or STOP bit.

Data is always exchanged in a 9-bit format, where 8-bits are used for data exchange and 1-bit is used for acknowledging the transmission by the receiver of the data. The receiver can be a master or a slave.

2.2.1 Address Frame

The first part of an I2C frame is called the address frame where the master presents a slave address. The address frame is comprised of a 7-bit slave address, the 8th bit indicates the direction of transfer and the 9th bit is used to acknowledge the transfer.

When the 8th bit is logical 0, the direction of data is from master to slave and is referred to as master transmit or write and when it is logical 1, the direction of data is from slave to master and is referred to as master receive or read.

The 9th bit is called the acknowledge bit (ACK) and is used by the device receiving the data to indicate it is ready to accept the address. If the slave address exists on the bus, then the device corresponding to it should drive the bus logical 0, which indicates to the master that it is ready to move to the data frame.

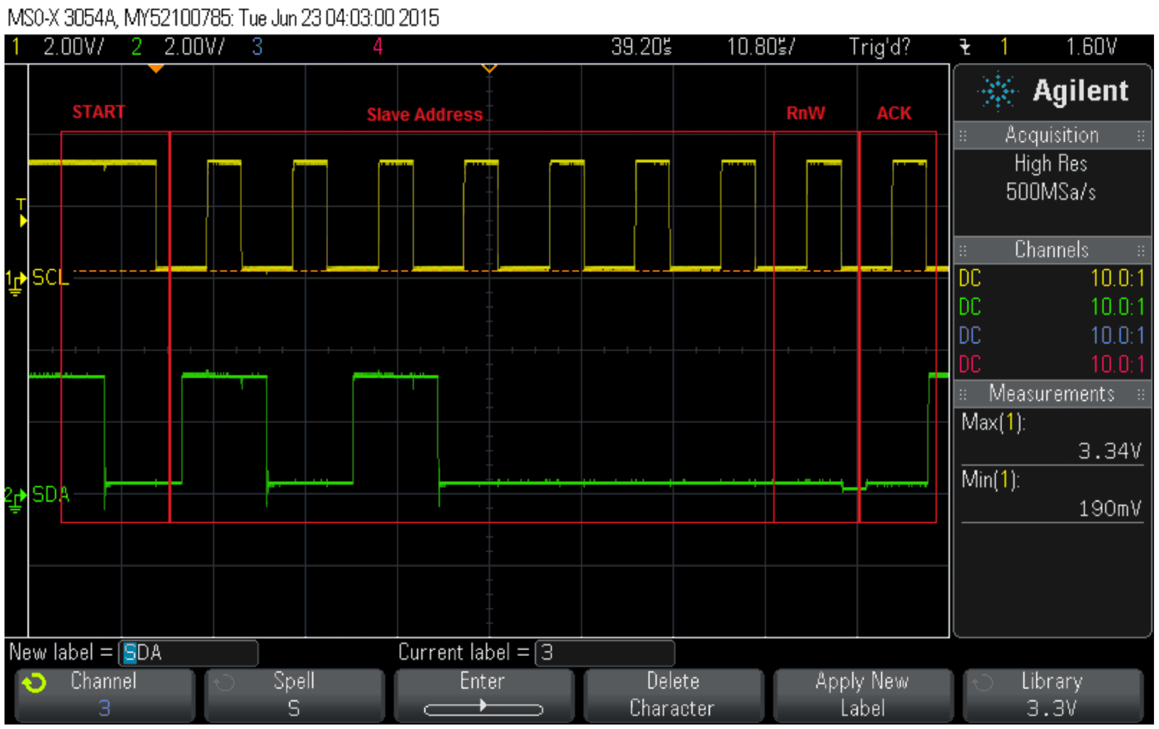


Figure 2. Address Phase ACK

If the slave address does not exist on the bus, then the master reads back a logical 1 (also called NAK) and terminates the transfer with a STOP bit. In some cases, slave devices use the ACK bit to indicate its readiness for further transaction.

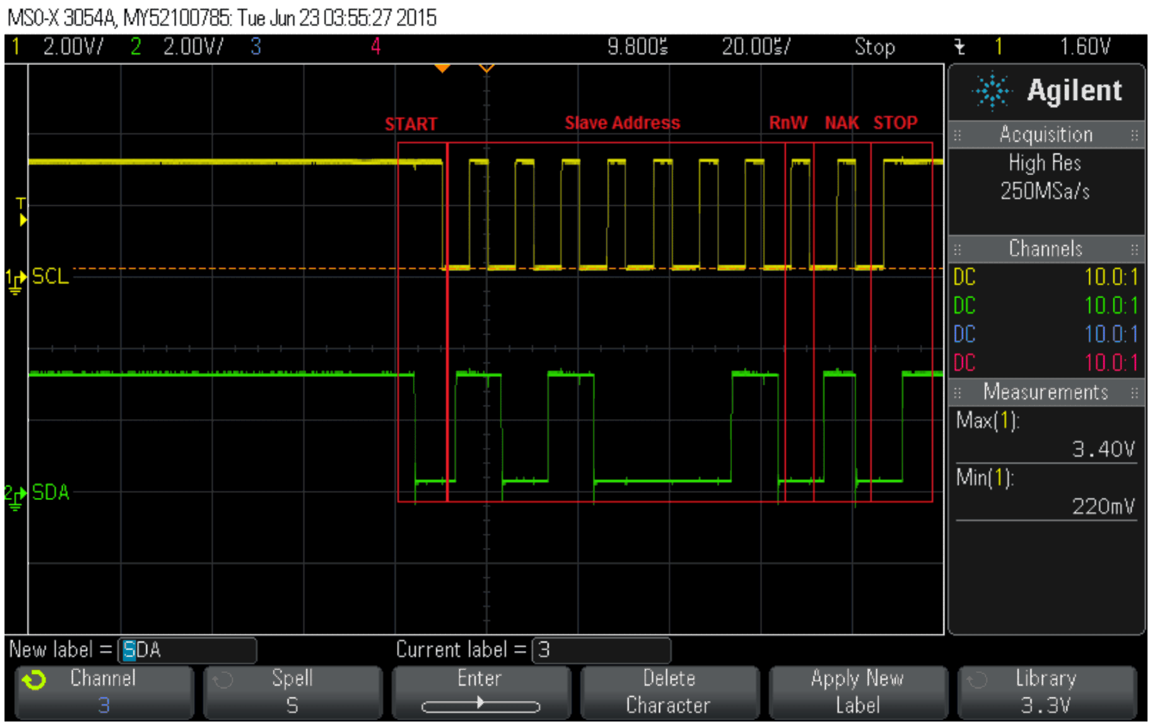


Figure 3. Address Phase NAK

2.2.2 Data Frame Write

After the address frame, the next frame to be sent is the data frame (called a write operation) if a logical 0 is sent in the 8th bit of the address frame. The direction of the transfer is from master to slave for the 8 bits of data frame to follow. The 9th bit is the ACK bit for the data frame and the direction of this bit is from slave to master. If the slave cannot accept the data, it should send a logical 1. The master should stop the I2C frame and the bus must return to idle state. If the slave accepts the data, it should send a logical 0 and the master can send the next byte, stop the frame or change the direction of the bus using repeated start.

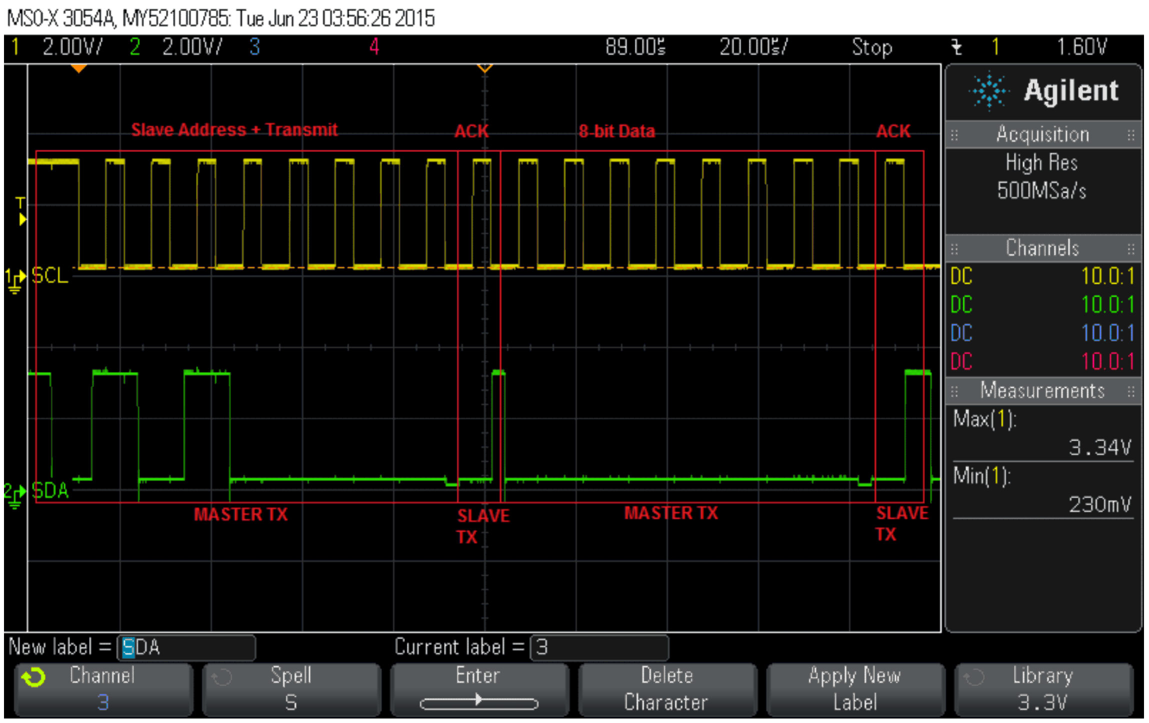


Figure 4. Data Frame Write

2.2.3 Data Frame Read

After the address frame, the next frame to be sent is the data frame (called a read operation) if a logical 1 is sent for the 8th bit in the address frame. The direction of the transfer is from slave to master for the 8 bits of data. The 9th bit is the ACK bit for the data frame and the direction of this bit is from master to slave. If the master cannot accept the data, it should send a logical 1, the master should issue a STOP condition on the I2C frame and the bus should return to idle state. If the master accepts the data, it should send a logical 0 and the master can accept the next byte.

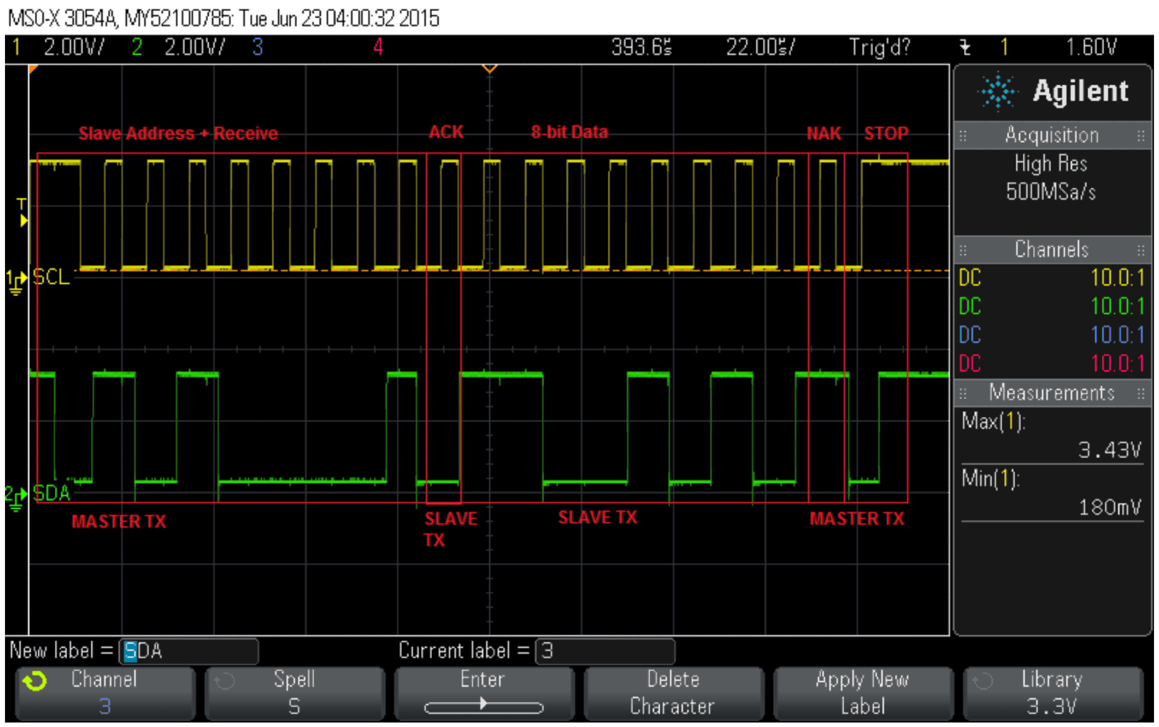


Figure 5. Data Frame Read

3 Circuit Schematic for Example Code

This section discusses the schematic for the slave device (LAPIS semiconductor MR44V064A), which has been used for development of the code examples. The slave device has been interfaced on the booster pack header 1 on the EK-TM4C1294XL Connected Launchpad.

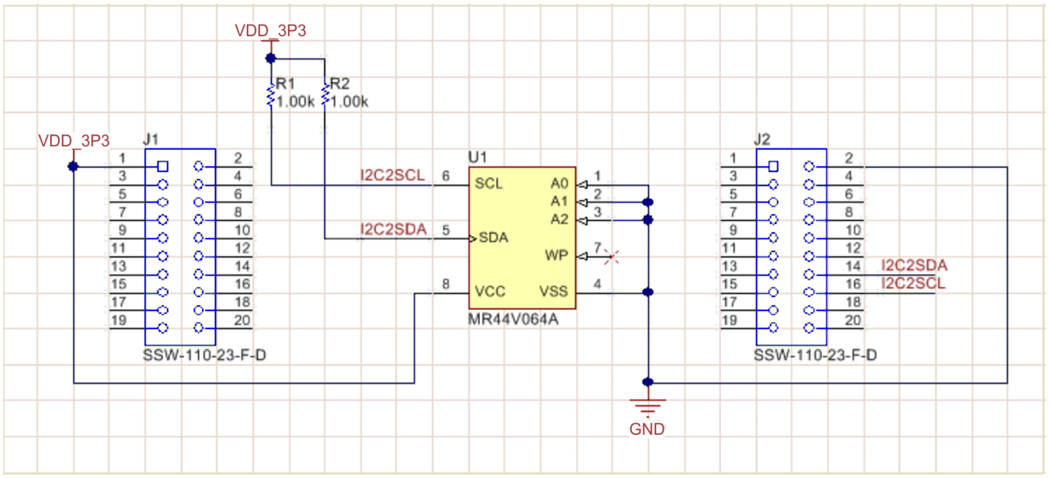


Figure 6. Circuit Schematic

4 Basic I2C Controller Initialization

NOTE: The I2C controller initialization examples that have been described will use I2C module instance 2 for the master function. Other I2C instances may be used but the user must refer to the following section to ensure that the other instances are properly initialized.

To be able to use the I2C module on the TM4C129x family of devices, there are application peripheral interfaces (API's) that need to be called from the TivaWare™ software to correctly initialize the I2C module on the TM4C129x devices.

There are two distinct sets of API's that are required before any operation of the I2C controller for the master function is configured:

- Configuring the IO's for I2C (see Section 4.1)
- Configuring the I2C controller (master function) (see Section 4.2)

Once the initialization has been completed, only the I2C API can be called to perform data transfers from the master function for which the following operations are important:

- Addressing a slave from the I2C master for transmit or receive operations (see Section 4.3)
- Applying the correct command to the I2C master for transmit or receive operations (see Section 4.4)

4.1 Configuration of IO's

The first step of configuration is to enable and configure the GPIO corresponding to the I2C module's SCL and SDA. The master and slave I2C modules are configured in the same manner. All I2C instances use the same API for configuration, except for the parameters that need to be passed to the API which would be instance specific.

Code Flow:

- Enable the Clock to the corresponding GPIO Module using *SysCtlPeripheralEnable* and wait for the peripheral ready using *SysCtlPeripheralReady*.
- Call the *GPIOPinConfigure* API to configure the SCL and SDA port mux for the specific GPIO port and pin.
- Call the *GPIOPinTypeI2C* API to configure the SDA in open drain mode, digital enable and alternate function select in the GPIO port for corresponding SDA pin.
- Call the *GPIOPinTypeI2CSCL* API to configure the SCL for digital enable and alternate function select in the GPIO port for the corresponding SCL pin. The open drain behavior is controlled by the I2C controller; hence, it must not be set in the GPIO Open Drain Select Register (GPIOODR).

```
//
//Enable GPIO for Configuring the I2C Interface Pins
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);

//
// Wait for the Peripheral to be ready for programming
//
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOL)
         || !SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOK));

//
// Configure Pins for I2C2 Master Interface
//
GPIOPinConfigure(GPIO_PL1_I2C2SCL);
GPIOPinConfigure(GPIO_PL0_I2C2SDA);
GPIOPinTypeI2C(GPIO_PORTL_BASE, GPIO_PIN_0);
GPIOPinTypeI2CSCL(GPIO_PORTL_BASE, GPIO_PIN_1);

//
// Configure Pins for I2C3 Slave Interface
//
GPIOPinConfigure(GPIO_PK4_I2C3SCL);
GPIOPinConfigure(GPIO_PK5_I2C3SDA);
GPIOPinTypeI2C(GPIO_PORTK_BASE, GPIO_PIN_5);
GPIOPinTypeI2CSCL(GPIO_PORTL_BASE, GPIO_PIN_4);
```

4.2 Configuring the I2C Controller (Master function)

The next step is configuring the I2C controller master function. Traditional examples of the I2C controller in TM4C have used the polling mechanism. This application report emphasizes the use of interrupts instead of polling.

Code Flow:

On TM4C129x device there is no System Control API to report the clock frequency.

1. Use the *SysCtlClockFreqSet* API, or hard code a variable with the value of 16 MHz because the default clock, after power up, is a 16 MHz precision oscillator (PIOSC). The clock frequency is required by the I2C API to compute the divider to generate the final I2C SCL clock frequency for different modes of operation.
2. Disable the clock, reset the I2C master module and then enable the clock to the I2C master module using the *SysCtlPeripheralDisable*, *SysCtlPeripheralReset* and *SysCtlPeripheralEnable* APIs. Then, wait for the peripheral ready using *SysCtlPeripheralReady*. This step is required to ensure that any stale state of the bus or the master function (due to a previous run) is removed.
3. Call the *I2CMasterInitExpClk* API to configure the I2C master function for the correct SCL clock frequency. The API uses the I2C base address as the first parameter, the system clock as the second parameter and, based on true or false in the last parameter, configures the SCL clock frequency for 100 KHz or 400 KHz. If the system clock is high enough to support high-speed mode, it will configure the high-speed dividers as well.
4. Enable the interrupt sources by setting the corresponding bits in the I2C master function Interrupt Mask Register. Use the *I2CMasterIntEnableEx* API, whose first parameter is the I2C base address and then the interrupt bits.
5. Enable the interrupt line from the I2C master function to the NVIC using the *IntEnable* API.

```
//
// Setup System Clock for 120MHz
//
ui32SysClock = SysCtlClockFreqSet((SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_XTAL_25MHZ |
                    SYSCTL_CFG_VCO_480), 120000000);

//
// Stop the Clock, Reset and Enable I2C Module
// in Master Function
//
SysCtlPeripheralDisable(SYSCTL_PERIPH_I2C2);
SysCtlPeripheralReset(SYSCTL_PERIPH_I2C2);
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C2);

//
// Wait for the Peripheral to be ready for programming
//
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_I2C2));

//
// Initialize and Configure the Master Module
//
I2CMasterInitExpClk(I2C2_BASE, ui32SysClock, false);

//
// Enable Interrupts for Arbitration Lost, Stop, NAK, Clock Low
// Timeout and Data.
//
I2CMasterIntEnableEx(I2C2_BASE, (I2C_MASTER_INT_ARB_LOST |
        I2C_MASTER_INT_STOP | I2C_MASTER_INT_NACK |
        I2C_MASTER_INT_TIMEOUT | I2C_MASTER_INT_DATA));

//
// Enable the Interrupt in the NVIC from I2C Master
//
IntEnable(INT_I2C2);
```

4.3 Addressing an I2C Slave in TM4C129x

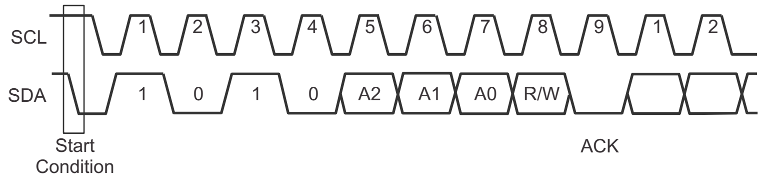
The I2C master function uses the *I2CMasterSlaveAddressSet* API to set the address of the slave device. The first parameter is the I2C base address for the master function, the second parameter is the external slave address (which is explained in detail using the example below) and the third parameter is the direction of transfer. The third parameter is set to “true” for a read operation or “false” for a write operation. In the example below, a call of the function is shown to access a slave device with the slave address as 0x50. Use Figure 7 to figure out the value.

```
//*****************************************************************************
//
// Define for I2C Module
//
//*****************************************************************************
#define SLAVE_ADDRESS_EXT 0x50
#define NUM_OF_I2CBYTES   255


  // Send the Slave Address with RnW as Transmit
  // and First Data Byte. Based on Number of bytes the
  // command would be either START or FINISH
  //
  I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS_EXT, false);
  I2CMasterDataPut(I2C2_BASE, g_ui8MasterTxData[g_ui8MasterBytes++]);
  if(g_ui8MasterBytes == g_ui8MasterBytesLength)
  {
       I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_SEND);
  }
  else
  {
       I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_BURST_SEND_START);
  }
}
```

The top part of Figure 7 shows the address phase as expected by the external slave device for which it would ACK the address phase, and the bottom part shows the I2C Master Slave Address Register (I2CMSA) information from the *Tiva™ TM4C1294NCPDT Microcontroller Data Sheet* (SPMS433). The bits shown as A2, A1 and A0 are based on the tie off done for the slave device pins, which are connected to GND. Therefore, the binary address as expected by the slave device is 1010000. This sequence in hexadecimal format is 0x50.

In the TM4C129x I2CMSA register, the address bits appear in bit position 7 to 1. The *I2CMasterSlaveAddressSet* API takes care of shifting the slave address to the correct position in the register.



	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	RO
Reset	0	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
	Reserved								SA							R/S
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:8	Reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:1	SA	RW	0x00	I^{2}C Slave Address This field specifies bits A6 through A0 of the slave address.
0	R/S	RW	0	Receive/Send The R/S bit specifies if the next master operation is a Receive (High) or Transmit (Low). Value Description 0 Transmit 1 Receive

Figure 7. I2CMSA and Slave Address Representation

4.4 *Data Transfer Commands for I2C Master in TM4C129x*

The I2C master has two modes of operation. In the first mode of operation the CPU or DMA can use the I2C Master Data Register (I2CMDR) to write data to a slave or read data from a slave using a single byte buffer. While the absolute address of the I2CMDR is programmed in the control structure of uDMA, for the CPU to access the data register there are two APIs that are available.

- *I2CDataPut* is used to write data to a slave during a transmit operation. The API expects the first parameter as the I2C base address and the second parameter is the data that must be transmitted
- *I2CDataGet* is used to read data from a slave during a receive operation. The API expects the first parameter as the I2C base address and returns the data read from the I2CMDR register.

The example codes of *ektm4c129_i2c_master_cpu_nonfifo* (see Section 5.1) and ektm4c129_i2c_master_hs (see Section 5.4) uses the first mode of data transfer.

The second mode of operation uses the I2C FIFO Data Register (I2CFIFODATA) to write data to a slave or read data from a slave using the FIFO. As with the first mode, the absolute address of the I2CFIFODATA is programmed in the control structure of the uDMA for DMA based transfers. For the CPU to access the data register, there are four APIs that are available.

- *I2CFIFODataPut* is used to write data to the TX FIFO in blocking mode. The API uses the first parameter of the I2C base address to polls the corresponding I2C FIFO Status Register (I2CFIFOSTATUS) to see if there is space available in the TX FIFO; if at least one byte space is available, the data then writes the second parameter to the I2CFIFODATA register.

- *I2CFIFODataPutNonBlocking* is used to write data to the TX FIFO in non-blocking mode. The API uses the first parameter of the I2C base address to check the corresponding I2CFIFOSTATUS register to see if there is space available in the TX FIFO and writes the second parameter to the I2CFIFODATA register. The API returns a ‘0’ if there is no space available (so it can be retried at a later time) or it returns a ‘1’ if the space was available and the data was written to the I2CFIFODATA register.
- *I2CFIFODataGet* is used to read data from the RX FIFO in blocking mode. The API uses the first parameter of the I2C base address to poll the corresponding I2CFIFOSTATUS register to see if there is data available in the RX FIFO; if at least one byte is available for read, the data is returned by the function.
- *I2CFIFODataGetNonBlocking* is used to read data from the RX FIFO in non-blocking mode. The API uses the first parameter of the I2C base address to check the corresponding I2CFIFOSTATUS register to see if there is data available in the RX FIFO, and writes the data read from the I2CFIFODATA into the buffer pointed by the second parameter. The API returns a ‘0’ if there is no data available (so it can be retried at a later time) or returns a ‘1’ if the data was available and was read from the I2CFIFODATA register.

The example codes of e*ktm4c129_i2c_master_cpu_fifo* (see Section 5.2) use the second mode of data transfer in non-blocking mode.

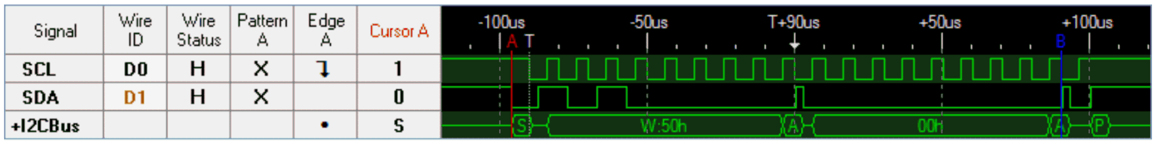
4.5 Commanding Operation of the I2C Master

The I2C master function uses the I2C Master Control/Status Register (I2CMCS) for performing any bus transaction. This register is a write-only register and read of this register will return the status of the I2C master function. Thus, if a user writes to the register in the application code, it will not read back the last command for the I2C master function but the status of the I2C master function resulting from the same.

The *i2c.h* file defines the different commands that can be sent to the I2C master function. From an application code perspective, there are eight major commands that the application can use to control the nature of the transaction from the I2C master.

4.5.1 I2C Transmit Command: I2C_MASTER_CMD_SINGLE_SEND

This command is used when the I2C master function is required to write one byte of data to a slave. As shown in Figure 8 when the I2C_MASTER_CMD_SINGLE_SEND command is sent, the I2C master function sends the start bit (S), slave address of 0x50 with Write Bit clear, transmits the data written to the I2CMDR and sends the stop bit (P).

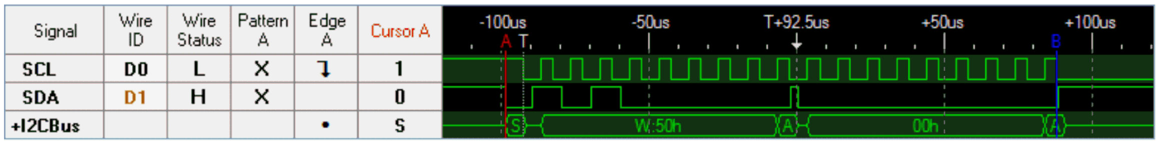


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↿	1
SDA	D1	H	X		0
+I2CBus				•	S

Figure 8. I2C Transaction for I2C_MASTER_CMD_SINGLE_SEND

4.5.2 I2C Transmit Command: I2C_MASTER_CMD_BURST_SEND_START

This command is used when the I2C master function is required to write one byte of data to a slave and own the bus. As shown in Figure 9 when the I2C_MASTER_CMD_BURST_SEND_START command is sent, the I2C master function sends the start bit (S), slave address of 0x50 with Write Bit clear, transmits the data written to the I2CMDR and holds the bus low for further operations.

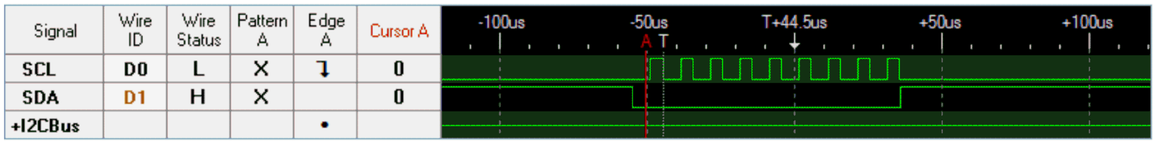


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	L	X	↿	1
SDA	D1	H	X		0
+I2CBus				•	S

Figure 9. I2C Transaction for I2C_MASTER_CMD_BURST_SEND_START

4.5.3 I2C Transmit Command: I2C_MASTER_CMD_BURST_SEND_CONT

This command is used when the I2C master function is required to an additional byte of data to a slave when it owns the bus. As shown in Figure 10 when the I2C_MASTER_CMD_BURST_SEND_CONT command is sent, the I2C master function transmits the data written to the I2CMDR and holds the bus low for further operations.

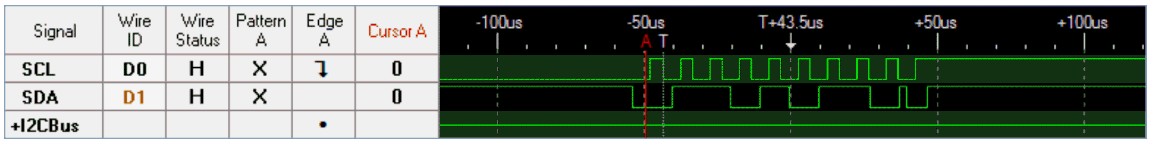


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	L	X	↓	0
SDA	D1	H	X		0
+I2CBus				•	

Figure 10. I2C Transaction for I2C_MASTER_CMD_BURST_SEND_CONT

4.5.4 I2C Transmit Command: I2C_MASTER_CMD_BURST_SEND_FINISH

This command is used when the I2C master function is required to write one byte of data to a slave and release the bus. As shown in Figure 11 when the I2C_MASTER_CMD_BURST_SEND_FINISH command is sent, the I2C master function transmits the data written to the I2CMDR and sends the stop bit (P).

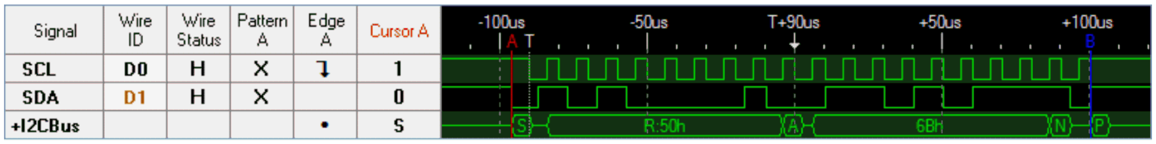


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	0
SDA	D1	H	X		0
+I2CBus				•	

Figure 11. I2C Transaction for I2C_MASTER_CMD_BURST_SEND_FINISH

4.5.5 I2C Receive Command: I2C_MASTER_CMD_SINGLE_RECEIVE

This command is used when the I2C master function is required to read one byte of data from a slave. As shown in Figure 12 when the I2C_MASTER_CMD_SINGLE_RECEIVE command is sent, the I2C master function sends the start bit (S), sets the slave address of 0x50 with Write Bit, receives the data from the slave, writes it to the I2CMDR, asserts a NAK to indicate to the slave that the transaction is completed and sends the stop bit (P).

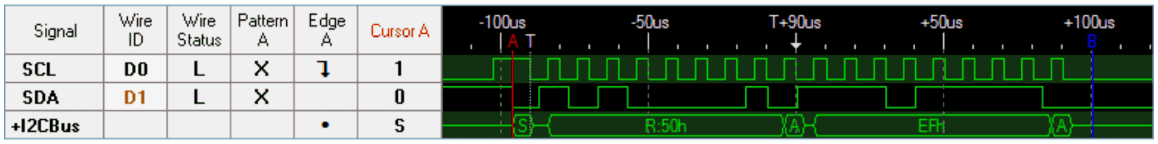


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	1
SDA	D1	H	X		0
+I2CBus				•	S

Figure 12. I2C Transaction for I2C_MASTER_CMD_SINGLE_RECEIVE

4.5.6 I2C Receive Command: I2C_MASTER_CMD_BURST_RECEIVE_START

This command is used when the I2C master function is required to read one byte of data from a slave and own the bus. As shown in Figure 13 when the I2C_MASTER_CMD_BURST_RECEIVE_START command is sent, the I2C master function sends the start bit (S), sets the slave address of 0x50 with Write Bit, receives the data from the slave, sends the data byte for ACK, writes the data to the I2CMDR and holds the bus for further operations.

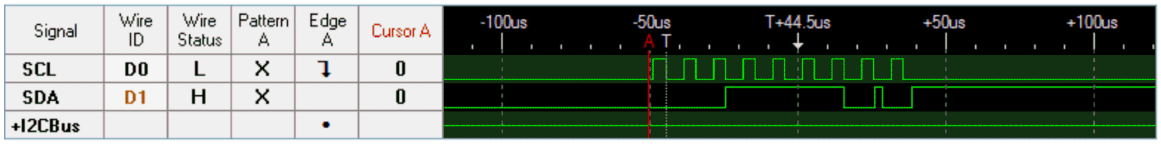


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	L	X	↓	1
SDA	D1	L	X		0
+I2CBus				•	S

Figure 13. I2C Transaction for I2C_MASTER_CMD_BURST_RECEIVE_START

4.5.7 I2C Receive Command: I2C_MASTER_CMD_BURST_RECEIVE_CONT

This command is used when the I2C master function is required to read one byte of data from a slave when it owns the bus. As shown in Figure 14 when the I2C_MASTER_CMD_BURST_RECEIVE_CONT command is sent, the I2C master function receives the data from the slave, sends ACK for the data byte, writes it to the I2CMDR and holds the bus for further operations.

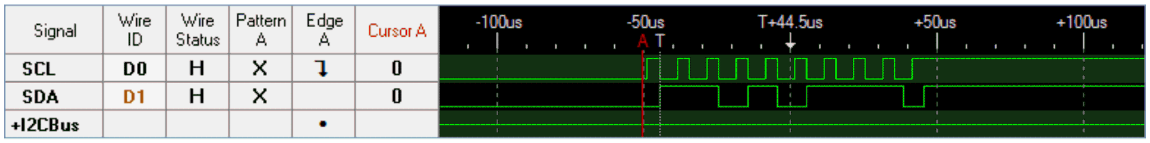


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	L	X	↧	0
SDA	D1	H	X		0
+I2CBus				•	

Figure 14. I2C Transaction for I2C_MASTER_CMD_BURST_RECEIVE_CONT

4.5.8 I2C Receive Command: I2C_MASTER_CMD_BURST_RECEIVE_FINISH

This command is used when the I2C master function is required to read one byte of data from a slave when it owns the bus and then release the bus. As shown in Figure 15 when the I2C_MASTER_CMD_BURST_RECEIVE_FINISH command is sent, the I2C master receives the data from the slave, sends a NAK to indicate that the transaction has completed and sends the stop bit (P).



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↧	0
SDA	D1	H	X		0
+I2CBus				•	

Figure 15. I2C Transaction for I2C_MASTER_CMD_BURST_RECEIVE_FINISH

5 I2C Master Function: Interrupt, uDMA and FIFO Operation

The I2C master supports both the polling and interrupt driven mechanisms for performing bus transactions. While the polling mechanism is simpler, the action of polling requires precious CPU cycles performing status checks. The TM4C129x family of devices now supports an exhaustive interrupt mechanism that an application code may use to free precious CPU clocks for other device activities.

The TM4C129x family of devices includes an 8-byte deep transmit and receive FIFO with configurable trigger threshold. The FIFO enables the CPU to setup a bulk transfer with fewer interrupts per I2C transactions leaving the CPU for other device activities.

The FIFO can be coupled with the efficient uDMA so that the CPU can be notified of a transaction completion, thus, reducing the overall CPU clocks managing a data transaction.

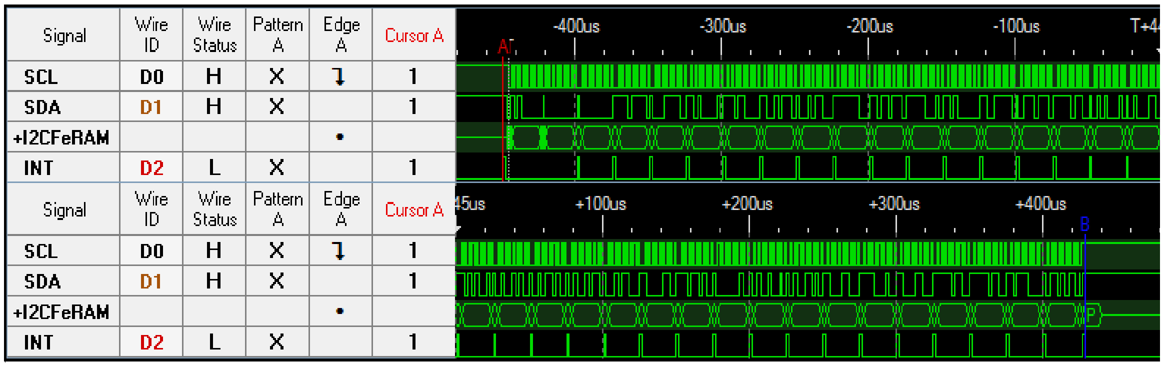
The code examples in the following subsections illustrate each of the features so that the application developer can leverage the I2C features of the TM4C129x devices for improving overall CPU bandwidth for essential system functions.

Each of the code examples mentioned below have logic analyzer captures associated with them, which shows both transmit and receive plots. The plot contains the SCL and SDA IO for the I2C master function and a GPIO toggle for interrupt, which is made high when the interrupt handler is called and made low on exit.

5.1 Master Function for CPU With Non-FIFO Transaction

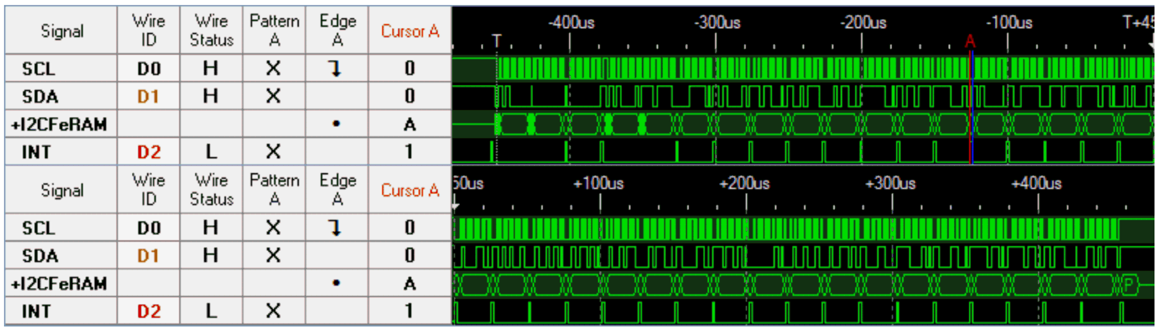
ektm4c129_i2c_master_cpu_nonfifo is a simple code example in which the CPU uses the traditional I2CMDR to access the slave for write and read operations with interrupts.

Figure 16 and Figure 17 show the transmit and receive operations. As can be seen for every data transaction, the CPU is being interrupted.



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	1
SDA	D1	H	X		1
+I2CFeRAM				•	
INT	D2	L	X		1
Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	1
SDA	D1	H	X		1
+I2CFeRAM				•	
INT	D2	L	X		1

Figure 16. Transmit With CPU and Non FIFO



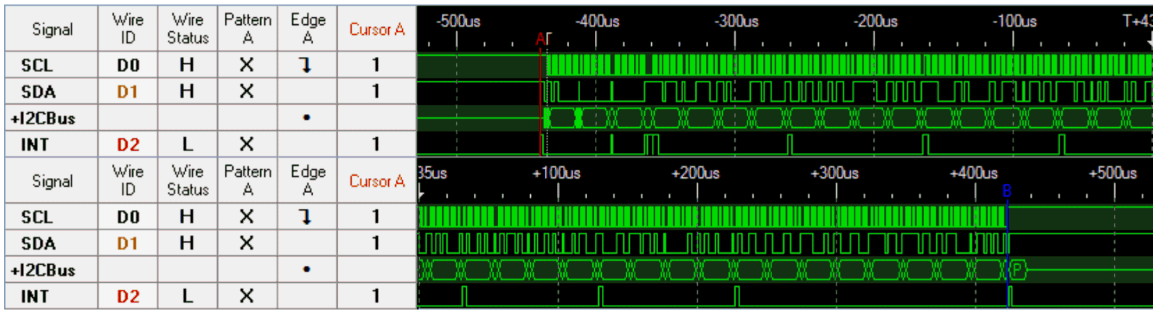
Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	0
SDA	D1	H	X		0
+I2CFeRAM				•	A
INT	D2	L	X		1
Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	0
SDA	D1	H	X		0
+I2CFeRAM				•	A
INT	D2	L	X		1

Figure 17. Receive With CPU and Non FIFO

5.2 Master Function for CPU With FIFO

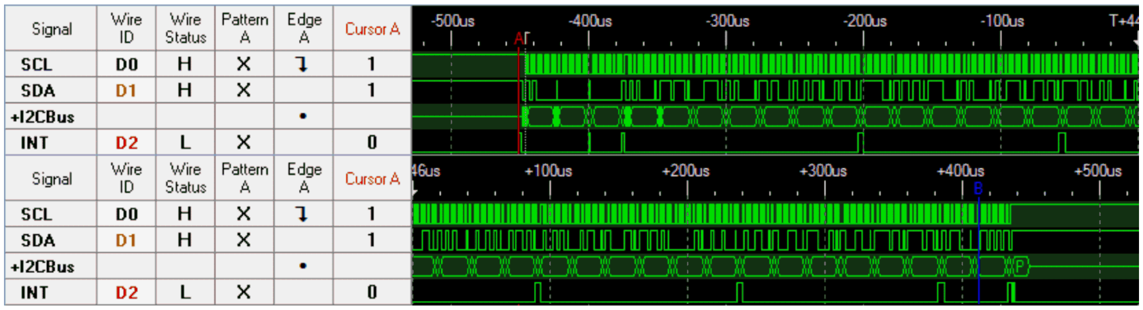
ektm4c129_i2c_master_cpu_fifo is a simple code example in which the CPU uses the I2CFIFODATA to access transmit and receive FIFO to perform write and read operations to a slave with interrupts.

Figure 18 and Figure 19 show the transmit and receive operations. As can be seen, the CPU is now being interrupted less than before; during the interrupt, the CPU can write or read the threshold of the data to the FIFO. However, this is still suboptimal as the buffer management has to be done by the CPU.



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X		1
Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X		1

Figure 18. Transmit With CPU and FIFO



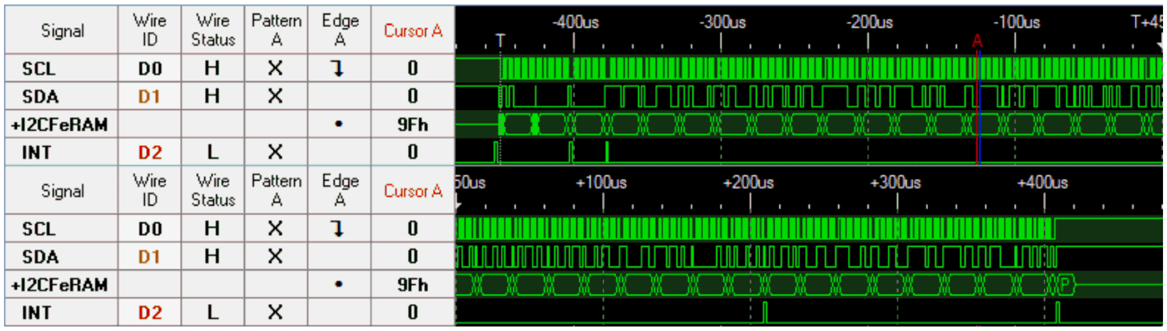
Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X		0
Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X		0

Figure 19. Receive With CPU and FIFO

5.3 Master Function for uDMA With FIFO

ektm4c129_i2c_master_udma_fifo is a simple code example where the CPU initializes and configures the uDMA to use the I2CFIFODATA to access transmit and receive FIFO to perform write and read operations to a slave with interrupt for the CPU on bus transaction completion. An important consideration in the code is setting the arbitration size of the uDMA based on the threshold of the FIFO triggers. The code itself has the optimal settings for the arbitration size as per the threshold of the TX and RX FIFOs.

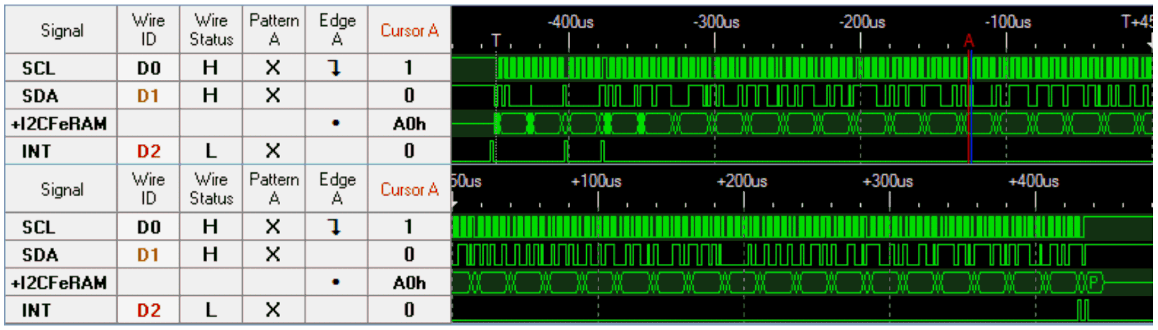
Figure 20 and Figure 21 show the transmit and receive operations. As can be seen, the CPU is now being interrupted only when the data transfer is completed by the uDMA and for handling bus events. This is an optimal usage of the FIFO when the uDMA performs the buffer management and only informing the CPU of a transaction completion.



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	0
SDA	D1	H	X		0
+I2CFeRAM				•	9Fh
INT	D2	L	X		0

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	0
SDA	D1	H	X		0
+I2CFeRAM				•	9Fh
INT	D2	L	X		0

Figure 20. Transmit With uDMA and FIFO



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	1
SDA	D1	H	X		0
+I2CFeRAM				•	A0h
INT	D2	L	X		0

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↓	1
SDA	D1	H	X		0
+I2CFeRAM				•	A0h
INT	D2	L	X		0

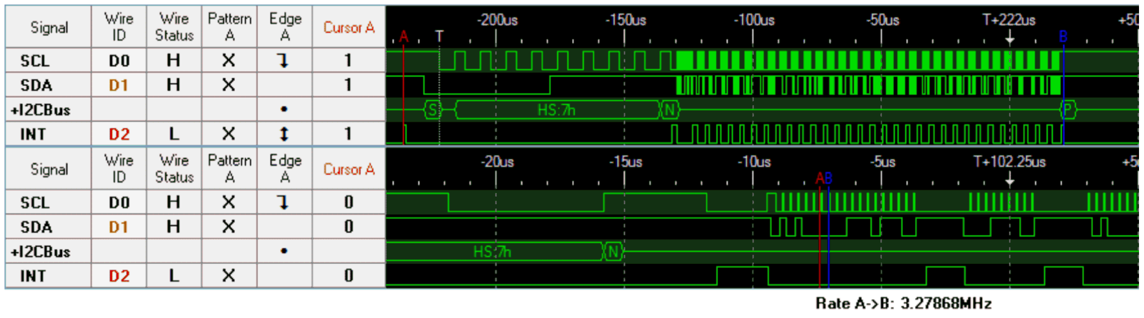
Figure 21. Receive With uDMA and FIFO

5.4 *Master Function for High-Speed Operation*

ektm4c129_i2c_master_hs is a simple code example where the CPU uses the traditional I2CMDR to access the slave for write and read operations with interrupts. The main transactions in this example are performed in high-speed mode reducing the overall time spent by the CPU for write and read bus transactions.

For both transmit and receive, Figure 22 shows the entire transmission and Figure 23 shows the zoomed in version with the baud rate for high speed.

Note that just like the other examples of using FIFO, CPU bandwidth for performing other system tasks can be improved while reducing the overall transaction time for the I2C bus transaction.

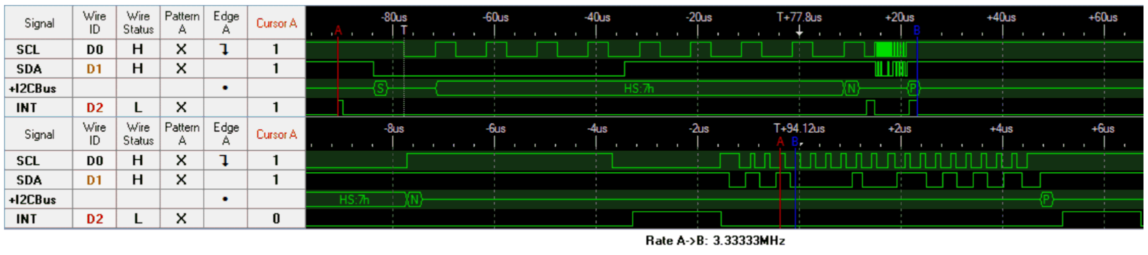


Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X	↕	1

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	0
SDA	D1	H	X		0
+I2CBus				•	
INT	D2	L	X		0

Rate A->B: 3.27868MHz

Figure 22. Transmit in HS Mode



Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X		1

Signal	Wire ID	Wire Status	Pattern A	Edge A	Cursor A
SCL	D0	H	X	↴	1
SDA	D1	H	X		1
+I2CBus				•	
INT	D2	L	X		0

Rate A->B: 3.33333MHz

Figure 23. Receive in HS Mode

6 TM4C129x I2C Glitch Filter Capability

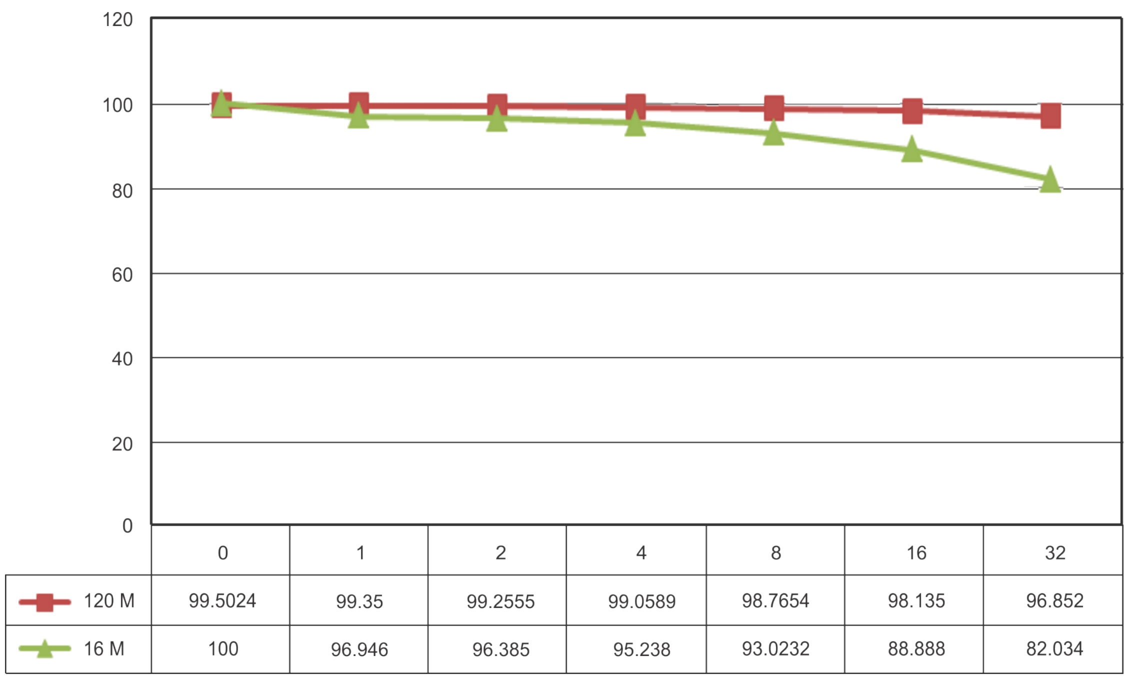
An important feature on the TM4C12x device is the I2C glitch filter that allows the application code to robustly handle transient noises that may affect the master. This feature is applicable for both the master and slave function of the TM4C12x devices. The major difference of this function on the TM4C129x when compared to the TM4C123x family of the devices is that the feature is enabled or disabled without having the requirement to set or clear control bit.

The I2C specification requirement for glitch suppression is 50 ns for standard (fast and fast plus mode) and 10 ns for high speed mode. The TM4C129x glitch filters work on the system clock and, based on the system clock frequency, the value assigned to the filter must be adjusted to the nearest available setting in the glitch filter register I2C Master Timer Period Register (I2CMTPR). For example, if the system clock is 120 MHz (8.33 ns), then for the standard mode glitch suppression of 50 ns would be a value of 6 system clocks. The nearest available value is 8 system clocks. Similarly for high-speed mode, a glitch suppression of 10 ns would be 1.2 system clocks, for which the value of 2 has to be programmed in the I2CMTPR.

The I2C glitch filter on the TM4C129x devices uses the API I2CMasterGlitchFilterConfigSet. The first parameter for the API is the base address of the I2C module and the second parameter is the filter setting. By setting the value of the filter as 0 (I2C_MASTER_GLITCH_FILTER_DISABLED), the glitch filter is automatically disabled.

```
//
// Enable the Glitch Filter. Writting a value 0 will
// disable the glitch filter
// I2C_MASTER_GLITCH_FILTER_DISABLED
// I2C_MASTER_GLITCH_FILTER_1
// I2C_MASTER_GLITCH_FILTER_2 : Ideal Value when in HS Mode
//                              for 120MHz clock
// I2C_MASTER_GLITCH_FILTER_4
// I2C_MASTER_GLITCH_FILTER_8 : Ideal Value when in Std,
//                              Fast, Fast+ for 120MHz clock
// I2C_MASTER_GLITCH_FILTER_16
// I2C_MASTER_GLITCH_FILTER_32
//
I2CMasterGlitchFilterConfigSet(I2C2_BASE, I2C_MASTER_GLITCH_FILTER_8);
```

However, the I2C glitch filter on the TM4C129x has an effect on the baud rate. Figure 24 shows the actual baud rate for a system clock of 120 MHz and 16 MHz when the device is programmed for 100 KHz operation. This can be compensated by overriding the TPR value in the I2CMTPR register after the initialization API's have been called.



	0	1	2	4	8	16	32
🟥 120 M	99.5024	99.35	99.2555	99.0589	98.7654	98.135	96.852
▲ 16 M	100	96.946	96.385	95.238	93.0232	88.888	82.034

Figure 24. Glitch Filter v/s Baud Rate

7 Conclusion

The new features of FIFO and uDMA for the I2C controller allow you to substantially improve the performance of the application in terms of CPU execution cycle allocation to the other system tasks, while ensuring that the I2C, in conjunction with the uDMA and FIFO, have better management of the data and control transfer from the peripheral devices in the end system.

8 References

- *Tiva™ TM4C1294NCPDT Microcontroller Data Sheet* (SPMS433)
- UM10204: I2C-Bus Specification and User Manual
- TivaWare software
- MR44V064A *64k (8,192-Word × 8-Bit) Ferroelectric Random Access Memory (FeRAM) Data Sheet* - LAPIS Semiconductor

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