FMC-IIC Software Guide

Version 1.03

AVNET® electronics marketing

Revision History

Version	Description	Date
0.01	Preliminary Version	Jan 08, 2010
1.01	Add support for >256 EEPROMs	Jan 12, 2010
1.02	Remove malloc from code	Jun 23, 2010
1.03	New sg_i2c_controller implementation: - add support for 16bit addressing - add new argument for programmable delay	Sep 03, 2010

References

1. XPS IIC Bus Interface – Product Specification
C:\Xilinx\11.4\EDK\hw\XilinxProcessorIPLib\pcores\xps_iic_v2_00_a\doc\xps_iic.pdf

Table of Contents

Table of Figures

Table of Tables

Introduction

The FMC-IIC software library is implemented for use in the EDK development environment.

This library provides a hardware abstraction layer to an I2C controller with general purpose output capability.

Features

The FMC-IIC software library provides the following functionality:

- Reusable by the following software libraries
    - FMC-IPMI
    - FMC-IMAGEOV
    - FMC-DVIDP
- Supports the following hardware implementations
    - XPS_IIC pcore (standard I2C controller in EDK)
    - Custom SG_I2C_CONTROLLER_PLBW pcore
      (picoblaze-based design, implemented in System Generator)
- Supports multiple instantiations

Software Library

Functional Description

The FMC-IIC software library provides a hardware abstraction layer to the I^{2}C controller. This abstraction layer offers the possibility to:

- support many hardware implementations of an I2C Controller
- compile the other layers of the software library on a different processor

This software library assumes that each I2C controller implementation will resemble the following figure.

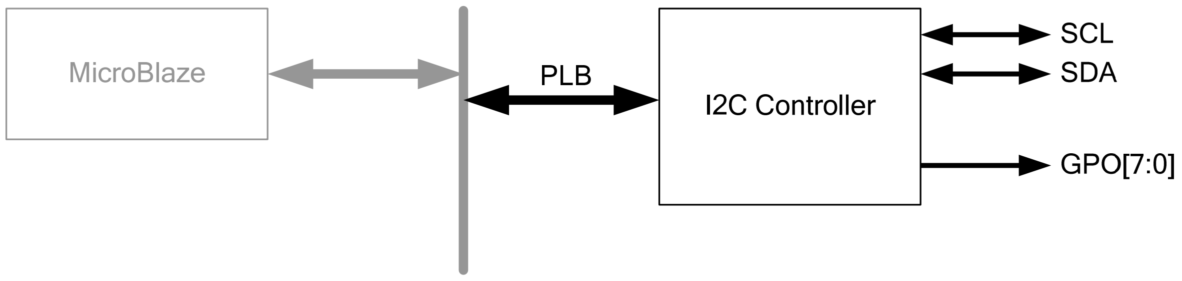

Figure 1 – FMC-IIC – I2C Controller Implementation

The I2C controller will have a PLB slave port, an I2C interface (SCL and SDA), as well as an 8 bit general purpose output port.

This version of the FMC-IIC software library supports two hardware implementations:

- fmc_iic_xps : uses the standard XPS_IIC pcore that ships with Xilinx EDK
- fmc_iic_sg : uses a picoblaze-based I2C controller implemented in Xilinx System Generator

Programming API

The following figure illustrates how an instance of the FMC-IIC software library will look like after being initialized.

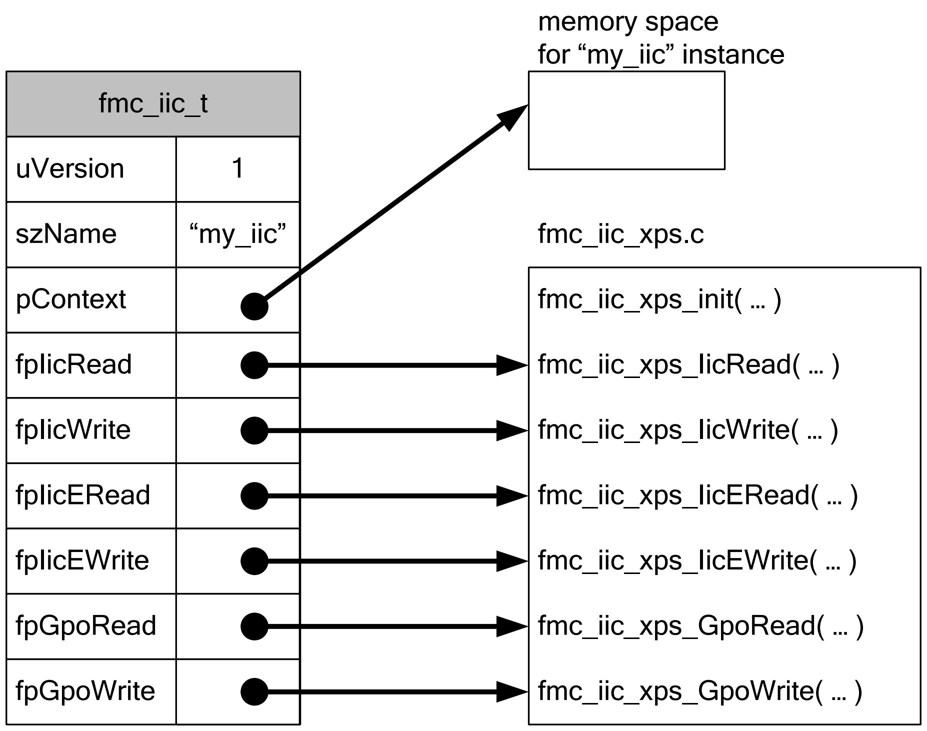

Figure 2 – FMC-IIC – Software Overview

The initialization function specific to an implementation, fmc_iic_xps_init(…) or fmc_iic_pb_init(…), must be called first. The arguments for each of these functions are implementation specific.

The following table lists the initialization functions that are available for each of the implementations supported in this software library.

Table 1 – FMC-IIC – Initialization Function Overview

Function	**Description**
fmc _iic_xps_init	Initialize the FMC-IIC software library for the XPS_IIC implementation.
fmc_iic_sg_init	Initialize the FMC-IIC software library for the SG_I2C_CONTROLLER_PLBW implementation.

Once initialized, the fmc_iic_t data structure provides an abstraction layer that can be used to access the I2C controller in the same manner, regardless of the implementation. The following table describes the contents of the fmc_iic_t data structure.

Table 2 – FMC-IIC – Data Structure

Element	**Description**
fmc_iic_t.uVersion	Software library version
fmc_iic_t.szName	String containing an instantiation specific descriptor
fmc_iic_t.pContext	Pointer to an instantiation specific memory space
fmc_iic_t.fpIicRead	Function pointer to implementation specific IIC Read code. Performs a read transaction on the I2C interface
fmc_iic_t.fpIicWrite	Function pointer to implementation specific IIC Write code. Performs a write transaction on the I2C interface
fmc_iic_t.fpIicERead	Function pointer to implementation specific IIC Read code. The extended version specified a 16 bit address for larger EEPROMs. Performs a read transaction on the I2C interface
fmc_iic_t.fpIicEWrite	Function pointer to implementation specific IIC Extended Write code. The extended version specified a 16 bit address for larger EEPROMs. Performs a write transaction on the I2C interface
fmc_iic_t.fpGpoRead	Function pointer to implementation specific GPO Read code. Reads the current value of the 8 bit GPIO output.
fmc_iic_t.fpGpoWrite	Function pointer to implementation specific GPO Write code. Sets the values of the 8 bit GPIO outputs

Example Usage

The following code excerpt describes how to use the FMC-IIC software library. The example shows how more than one instance of the library can be used to access several IIC chains.

```
#include "fmc_iic.h"
fmc_iic_t iic1;
fmc_iic_t iic2;

Xuint8    iic_device;
Xuint8    iic_address;
Xuint8    iic_data;
int       retval;

// FMC-IIC Initialization (XPS_IIC implementation)
if ( !fmc_iic_xps_init( &iic1, "IIC Chain #1", XPAR_XPS_IIC_0_BASEADDR ) )
{
   xil_printf( "ERROR : Failed to initialize IIC Chain #1\n\r" );
   exit(0);
}
if ( !fmc_iic_xps_init( &iic2, "IIC Chain #2", XPAR_XPS_IIC_1_BASEADDR ) )
{
   xil_printf( "ERROR : Failed to initialize IIC Chain #2\n\r" );
   exit(0);
}

// Read Data from EEPROM (0xA0) on IIC Chain #1
iic_device = 0xA0;
xil_printf( "EEPROM Contents:" );
for ( iic_address = 0; iic_address < 256; iic_address++ )
{
   if ( !(iic_address % 8) )
   {
      xil_printf( "\n\r[0x%02X] ", iic_address );
   }
   retval = iic1.fpIicRead( &iic1, (iic_device>>1), iic_address, &iic_data, 1 );
   if ( retval == 1 )
   {
      xil_printf( "0x%02X ", iic_data );
   }
   else
   {
      xil_printf( "\n\rERROR : Reading from EEPROM device on IIC Chain #1" );
      break;
   }
}
xil_printf( "\n\r" );

// Write Configuration to TFP410 device (0x70) on IIC Chain #2
iic_device = 0x70;
iic_address = 0x08;
retval = iic2.fpIicRead( &iic2, (iic_device>>1), iic_address, &iic_data, 1 );
if (retval < 1 )
{
   xil_printf( "ERROR : Reading from TFP410 device on IIC Chain #2" );
}
iic_data |= 0x01; // PDN# = 1 (power-on TFP410 device)
retval = iic2.fpIicWrite( &iic2, (iic_device>>1), iic_address, &iic_data, 1 );
if (retval < 1 )
{
   xil_printf( "ERROR : Writing to TFP410 device on IIC Chain #2" );
}
```

Implementation Details

FMC_IIC_XPS Implementation

The **fmc_iic_xps** implementation uses the standard XPS_IIC pcore that ships with Xilinx EDK.

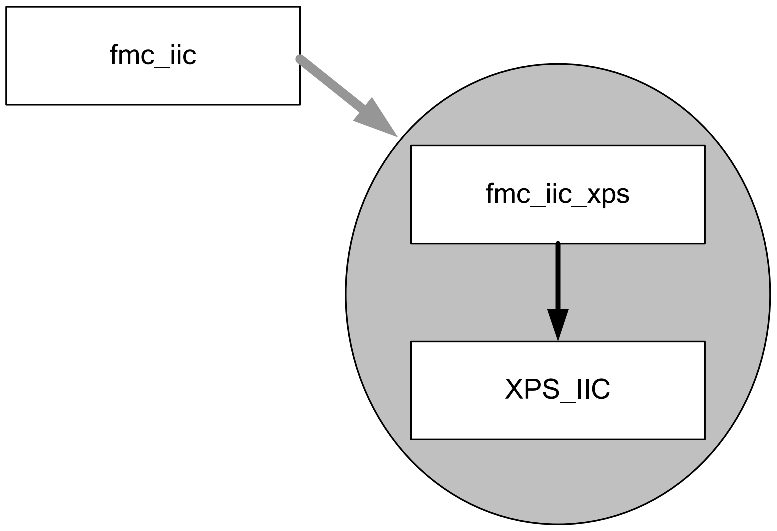

Figure 3 – FMC-IIC – FMC_IIC_XPS Implementation Details

The following .MHS excerpt shows how a XPS_IIC pcore could be instantiated.

```
...

# Top level port
 PORT fmc_ipmi_scl = xps_iic_0_Scl, DIR = IO
 PORT fmc_ipmi_sda = xps_iic_0_Sda, DIR = IO

...

# XPS_IIC pcore instantiation
BEGIN xps_iic
  PARAMETER INSTANCE = xps_iic_0
  PARAMETER HW_VER = 2.00.a
  PARAMETER C_CLK_FREQ = 62500000
  PARAMETER C_GPO_WIDTH = 8
  PARAMETER C_BASEADDR = 0x81010000
  PARAMETER C_HIGHADDR = 0x8161FFFF
  BUS_INTERFACE SPLB = mb_plb
  PORT Scl = xps_iic_0_Scl
  PORT Sda = xps_iic_0_Sda
  PORT Gpo = Gpo7 & Gpo6 & Gpo5 & Gpo4 & Gpo3 & Gpo2 & Gpo1 & Gpo0
END

...
```

For more information on this pcore, please consult the Xilinx documentation [1].

FMC_IIC_SG Implementation

The **fmc_iic_sg** implementation uses a custom pcore that was implemented in Xilinx System Generator. This pcore uses a Picoblaze-based implementation.

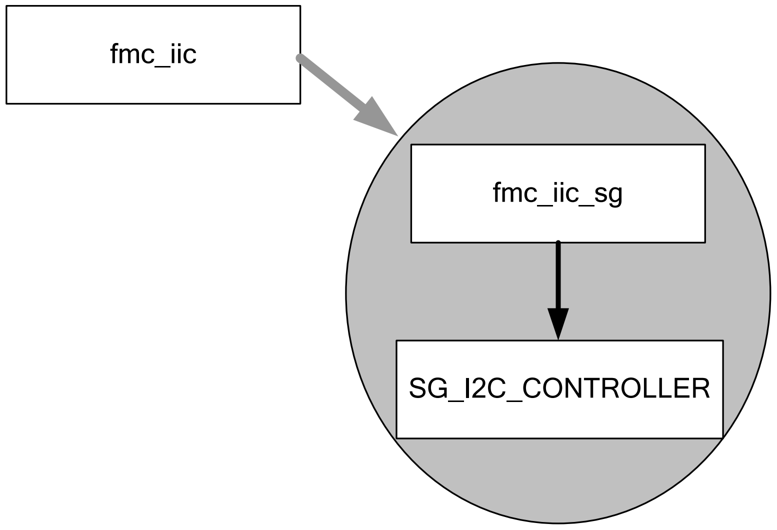

Figure 4 – FMC-IIC – FMC_IIC_SG Implementation Details

The following .MHS excerpt shows how a SG_I2C_CONTROLLER pcore could be instantiated.

```
...

# Top level port
 PORT fmc_ipmi_scl = sg_i2c_controller_plbw_0_Scl, DIR = O
 PORT fmc_ipmi_sda = sg_i2c_controller_plbw_0_Sda, DIR = IO

...

# SG_I2C_CONTROLLER pcore instantiation
BEGIN sg_i2c_controller_plbw
  PARAMETER INSTANCE = sg_i2c_controller_plbw_0
  PARAMETER HW_VER = 1.01.a
  PARAMETER C_BASEADDR = 0x81010000
  PARAMETER C_HIGHADDR = 0x8161FFFF
  BUS_INTERFACE SPLB = mb_plb
  PORT i2c_scl = sg_i2c_controller_plbw_0_Scl
  PORT i2c_sda = sg_i2c_controller_plbw_0_Sda
  PORT gpio_out8_o = Gpo7 & Gpo6 & Gpo5 & Gpo4 & Gpo3 & Gpo2 & Gpo1 & Gpo0
END

...
```

The PicoBlaze-based I2C controller provides two modes of operation:

- static configuration (executed unconditionally at power-up)
- dynamic configuration (executed using a command port)

The static configuration consists of the following:

- … TBD (?) …

The dynamic configuration is available using a Command Port as illustrated in the following block diagram:

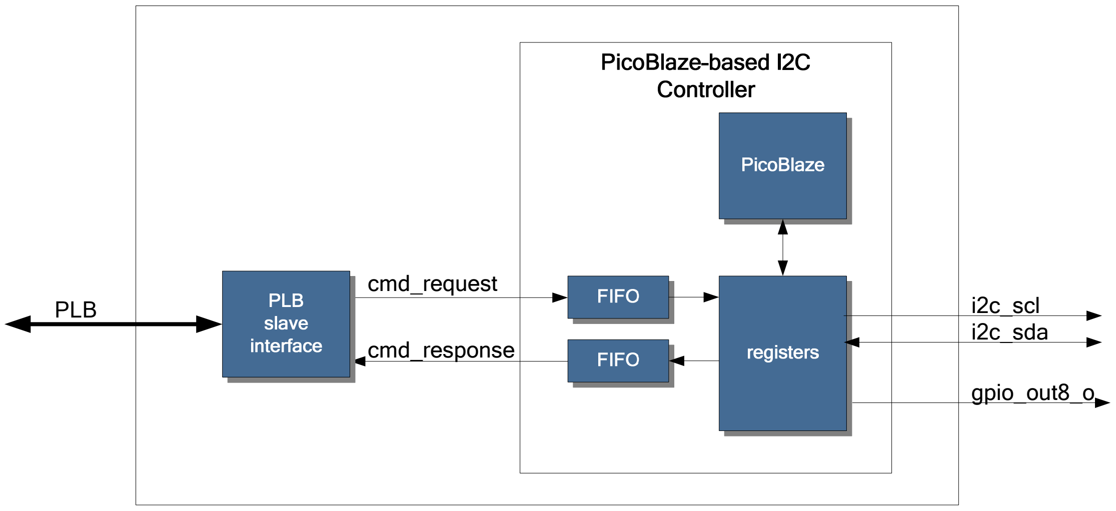

Figure 5 – SG_I2C_CONTROLLER – Block Diagram

The Command Port is implemented with two 32 bit FIFOs. A Command is initiated by sending a request to the PicoBlaze via the cmd_request FIFO. The PicoBlaze sends a response to each request in the cmd_response FIFO.

The data format for cmd_request FIFO is defined as follows:

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x01 : I2C register write 0x02 : I2C register read 0x08 : Programmable delay value 0x10 : Start Bit 0x11 : Stop Bit 0x12 : Write Byte 0x13 : Write Last Byte 0x14 : Read Byte (ack asserted) 0x15 : Read Last Byte (ack not asserted)
[23:16]	cmd_param1[7:0]	parameter 1 (different for each cmd_id)
[15:8]	cmd_param2[7:0]	parameter 2 (different for each cmd_id)
[7:0]	cmd_param3[7:0]	parameter 3 (different for each cmd_id)

Table 3 – SG_I2C_CONTROLLER – Command Request

When an unknown cmd_id has been detected, the PicoBlaze will respond 0xDEADBEEF in the cmd_response FIFO.,

When an I2C error occurs, the PicoBlaze will respond 0xDEADCAFE in the cmd_response FIFO.

When an I2C write or read is successful, the cmd_response data format is defined as follows:

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	cmd_return1[7:0]	return value 1 (different for each cmd_id)
[15:8]	cmd_return2[7:0]	return value 2 (different for each cmd_id)
[7:0]	cmd_return3[7:0]	return value 3 (different for each cmd_id)

Table 4 – SG_I2C_CONTROLLER – Command Response

The following sub-sections describe the format of the command request and response for each command

I2C_REG_WRITE

This command is used to generate a complete I2C write transaction, including:

- start bit
- write byte (device, write mode)
- write byte (register address)
- write byte (register data)
- stop bit

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x01 : I2C register write
[23:16]	i2c_device_addr[7:0]	I2C device address
[15:8]	i2c_register addr[7:0]	I2C register address
[7:0]	I2c_register_data[7:0]	I2C register data

Table 5 – I2C_REG_WRITE – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	same as cmd_request[23:16]
[15:8]	i2c_register addr[7:0]	same as cmd_request[15:8]
[7:0]	i2c_register_data[7:0]	same as cmd_request[7:0]

Table 6 – I2C_REG_WRITE – Command Response

I2C_REG_READ

This command is used to generate a complete I2C read transaction, including:

- start bit
- write byte (device, write mode)
- write byte (register address)
- start bit
- write byte (device, read mode)
- read byte (register data)
- stop bit

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x02 : I2C register read
[23:16]	i2c_device_addr[7:0]	I2C device address
[15:8]	i2c_register addr[7:0]	I2C register address
[7:0]	I2c_register_data[7:0]	unused

Table 7 – I2C_REG_READ – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	same as cmd_request[23:16]
[15:8]	i2c_register addr[7:0]	same as cmd_request[15:8]
[7:0]	i2c_register_data[7:0]	data read from I2C device

Table 8 – I2C_REG_READ – Command Response

DELAY_VALUE

This command is used to change the programmable delay value that is used to create the delays in the I2C timing.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x08 : Change Programmable Delay Value
[23:16]	i2c_device_addr[7:0]	Delay Value
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 9 – DELAY_VALUE – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	same as cmd_request[23:16]
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 10 – DELAY_VALUE – Command Response

I2C_SEND_START

This command is used to generate a start bit sequency in the I2C protocol.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x10 : Send I2C Start Bit
[23:16]	i2c_device_addr[7:0]	unused
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 11 – I2C_SEND_START – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	0x00
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 12 – I2C_SEND_START – Command Response

I2C_SEND_STOP

This command is used to generate a stop bit sequence in the I2C protocol.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x11 : Send I2C Stop Bit
[23:16]	i2c_device_addr[7:0]	unused
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 13 – I2C_SEND_STOP – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	0x00
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 14 – I2C_SEND_STOP – Command Response

I2C_WRITE_BYTE

This command is used to write an 8 bit data sequence in the I2C protocol.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x12 : Write I2C Byte
[23:16]	i2c_device_addr[7:0]	data byte to send
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 15 – I2C_WRITE_BYTE – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	same as cmd_request[23:16]
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 16 – I2C_WRITE_BYTE – Command Response

I2C_WRITE_LAST

This command is used to write the last 8 bit data sequence in the I2C protocol. The stop bit sequence is generated to finish the transaction.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x13 : Write Last I2C Byte
[23:16]	i2c_device_addr[7:0]	data byte to send
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 17 – I2C_WRITE_LAST – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	same as cmd_request[23:16]
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 18 – I2C_WRITE_LAST – Command Response

I2C_READ_BYTE

This command is used to read an 8 bit data sequence in the I2C protocol. The ack is asserted to indicate that another read is requested.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x14 : Write I2C Byte
[23:16]	i2c_device_addr[7:0]	unused
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 19 – I2C_READ_BYTE – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	data byte read
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 20 – I2C_READ_BYTE – Command Response

I2C_READ_LAST

This command is used to read the last 8 bit data sequence in the I2C protocol. The ack is not asserted to indicate that this is the last byte read. The stop bit sequence is generated to finish the transaction.

cmd_request		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	0x15 : Write Last I2C Byte
[23:16]	i2c_device_addr[7:0]	unused
[15:8]	i2c_register addr[7:0]	unused
[7:0]	I2c_register_data[7:0]	unused

Table 21 – I2C_READ_LAST – Command Request

cmd_response		
Bits	**Name**	**Description**
[31:24]	cmd_id[7:0]	same as cmd_request[31:24]
[23:16]	i2c_device_addr[7:0]	data byte read
[15:8]	i2c_register addr[7:0]	0x00
[7:0]	i2c_register_data[7:0]	0x00

Table 22 – I2C_READ_LAST – Command Response

Known Limitations

The following limitations exist with the FMC-IIC software library.

Table 23 – FMC-IIC – Known Limitations

Implementation	Limitations
fmc_iic_xps	Long timeouts[1] will occur under following conditions: - I2C device does not respond
	Does not support the particular timing of the following devices[2]: - AD9887 - OmniVision image sensors
fmc_iic_pb	Does not support multi-master capability - SCL cannot be tri-stated, since it is an output only port

[1] These long timeouts make it unpractical to perform a scan of all possible I2C devices on the I2C chain. The fmc_iic_sg implementation overcomes this limitation.
[2] This known limitation is overcome by the fmc_iic_sg implementation. This was the main motivator for adding support for different I2C controller implementations.