I²C Core User Guide

UG-CORE-I2C-v4.4
February 2023
www.efinixinc.com

Contents

Introduction **3**

Features **3**

Resource Utilization and Performance **4**

Functional Description **5**

Ports 6

I^2C Core Registers 8

I^2C Write and Read Operations 8

IP Manager **12**

Customizing the I^2C **13**

I^2C Example Design **14**

I^2C Testbench **17**

Interface Designer GPIO Block Settings **17**

Revision History **18**

Introduction

The I^2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange between devices. The I^2C core provides an interface between the Trion® FPGA and an I^2C bus.

Use the IP Manager to select IP, customize it, and generate files. The I^2C core has an interactive wizard to help you set parameters. The wizard also has options to create a testbench and/or example design targeting an Efinix® development board.

Features

- Supports native user interface
- Master, slave, and multi-master operations
- Supports 100 kHz and 400 kHz I^2C operation mode
- START, Repeated START and STOP signal generation and detection
- 7-bit slave addressing mode
- Verilog HDL RTL and simulation testbench
- Includes example designs targeting the Trion® T20 BGA256 Development Board and Titanium Ti60 F225 Development Board
- Supports SDA and SCL spike filtering, and SCL clock stretching

FPGA Support

The I^2C core supports all Trion® and Titanium FPGAs.

Resource Utilization and Performance

Note: The resources and performance values provided are just guidance and change depending on the device resource utilization, design congestion, and user design.

Titanium Resource Utilization and Performance

FPGA	Mode	Logic and Adders	Flip-flops	Memory Blocks	DSP48 Blocks	f_{MAX} (MHz)[1]	Efinity® Version[2]
Ti60 F225 C4	Master	217	175	0	0	561	2021.2
	Slave	216	175	0	0	438	

Trion Resource Utilization and Performance

FPGA	Mode	Logic Utilization (LUTs)	Registers	Memory Blocks	Multipliers	f_{MAX} (MHz)[1]	Efinity® Version[2]
T20 BGA256 C4	Master	441	203	0	0	154	2021.1
	Slave	258	198	0	0	210	

(1) Using default parameter settings.
(2) Using Verilog HDL.

Functional Description

The core supports master and slave modes.

The I²C core consists of:

- *I²C master*—Top module wrapper that predefines the I²C core to master mode.
- *I²C slave*—Top module wrapper that predefines the I²C core to slave mode.
- *I²C master/slave controller*—I²C core logic.
- *Microcontroller inbound data register (MIDR)*—TX data register.
- *Microcontroller outbound data register (MODR)*—RX data register.
- *Microcontroller address data register (MADR)*—I²C slave address register.

Figure 1: I²C Master System Block Diagram

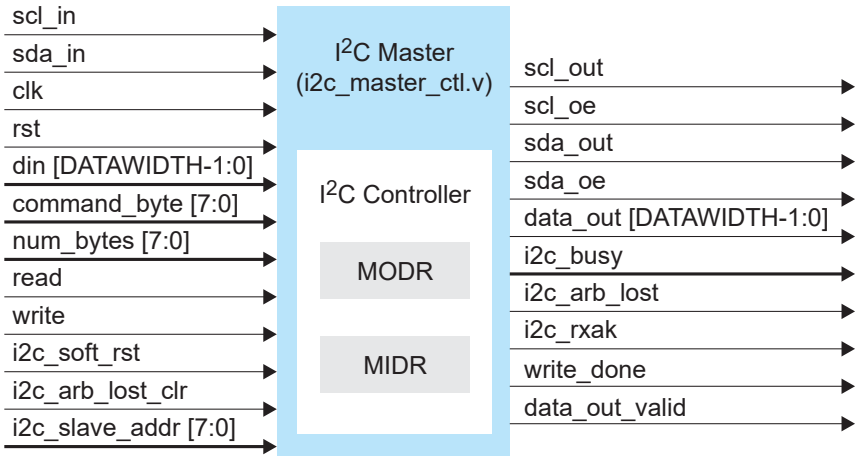
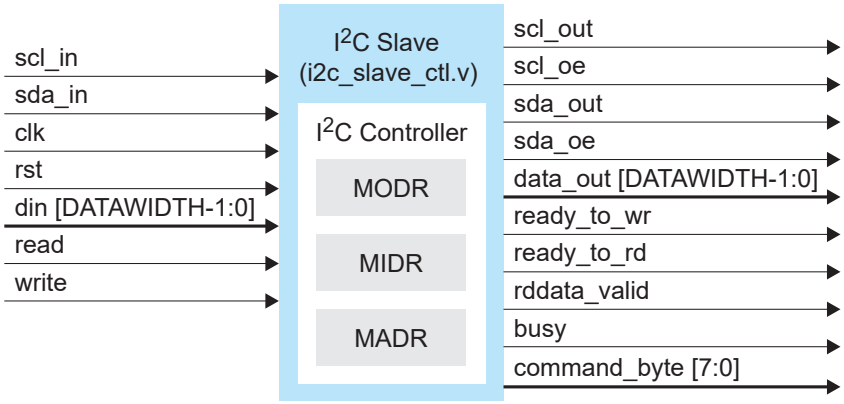


Figure 2: I²C Slave System Block Diagram



Ports

Table 1: I^2C Master Ports

Port	Interface	Direction	Description
scl_in	I^2C	Input	I^2C clock input.
sda_in	I^2C	Input	I^2C data input.
scl_out	I^2C	Output	I^2C clock output.
scl_oe	I^2C	Output	I^2C clock output enable.
sda_out	I^2C	Output	I^2C data output.
sda_oe	I^2C	Output	I^2C data output enable.
clk	System	Input	IP clock.
rst	System	Input	IP reset.
din [DATA_WIDTH-1:0]	System	Input	Write data input.
command_byte [7:0]	System	Input	This 8-bit data is sent to the I^2C slave device during the I^2C command phase.
num_bytes [7:0]	System	Input	Determines the number of data in bytes to be written to the I^2C slave device or read back from the I^2C slave device.
read	System	Input	Assert high for one clock cycle to read data from the I^2C slave device. Assign num_bytes before asserting the read port.
write	System	Input	Assert high for one clock cycle to write data to the I^2C slave device. Assign num_bytes, command_bytes, and din before asserting the write port.
i2c_soft_rst	System	Input	Soft reset the I^2C bus.
i2c_arb_lost_clr	System	Input	Assert high for one clock cycle to clear the i2c_arb_lost port.
i2c_slave_addr [7:0]	System	Input	This 8-bit data is sent to the I^2C slave device during the I^2C header phase. The least significant bit is ignored.
data_out [DATA_WIDTH-1:0]	System	Output	Read data output.
i2c_busy	System	Output	Logic high indicates that the I^2C bus is busy.
i2c_arb_lost	System	Output	Logic high indicates that there is arbitration lost in the I^2C transfer.
i2c_rxak	System	Output	Logic low indicates that the I^2C slave device received and acknowledged the I^2C transfer.
write_done	System	Output	Logic high indicates that I^2C master write data is sent and ready to accept by I^2C slave device.
data_out_valid	System	Output	Logic high indicates that I^2C master read data is valid and ready to read by user.

Table 2: I^2C Slave Ports

Port	Interface	Direction	Description
scl_in	I^2C	Input	I^2C clock input.
sda_in	I^2C	Input	I^2C data input.
scl_out	I^2C	Output	I^2C clock output.
scl_oe	I^2C	Output	I^2C clock output enable.
sda_out	I^2C	Output	I^2C data output.
sda_oe	I^2C	Output	I^2C data output enable.
clk	System	Input	IP clock.
rst	System	Input	IP reset.
din [DATA_WIDTH-1:0]	System	Input	Write data input.
read	System	Input	Assert high for one clock cycle to read data from the I^2C master.
write	System	Input	Assert high for one clock cycle to write data to the I^2C master. Number of data bytes to be sent to the master is equal to DATA_WIDTH/8.
data_out [DATA_WIDTH-1:0]	System	Output	Read data output.
ready_to_wr	System	Output	Logic high indicates that the slave is ready to accept write data from the user.
ready_to_rd	System	Output	Logic high indicates that the slave has read data ready to be read.
rddata_valid	System	Output	Logic high indicates that the read data is valid and ready to be read by the user.
busy	System	Output	Logic high indicates the is busy.
command_bytes [7:0]	System	Output	This 8-bit data is received from I^2C master device during the I2C command phase.

I²C Core Registers

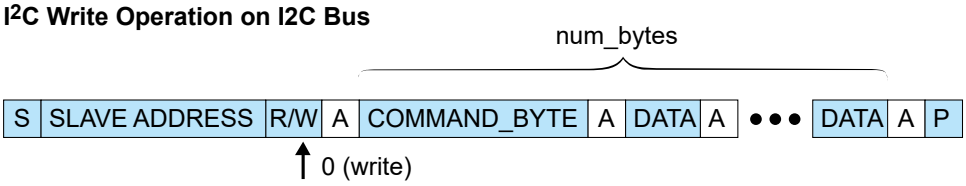
Table 3: I²C Core Registers

Bit	Name	Description
7:0	MIDR	Data byte from command_byte and din ports are written into this data register and transferred out through the I²C bus. When num_bytes is more than 2, the subsequent byte of din is written to MIDR after one byte of data transfer completed.
7:0	MODR	Data received from the I²C transfer is written to this register. This register value is assigned to the data_out port. When there are more than 1 byte of data received, the previous data byte is right-shifted to the least significant bit (LSB) and concatenate with the current received byte to form data_out.
7:0	MADR	This is an I²C slave specific register. Parameter SLAVE_ADDR is assigned to this register. This register value is compared with the I²C header byte send by the I²C master. If these values match, the I²C slave sends ACK the I²C master. Otherwise, it sends NACK to the ²C master. The least significant bit is ignored. Only MADR [7:1] are compared with the header byte.

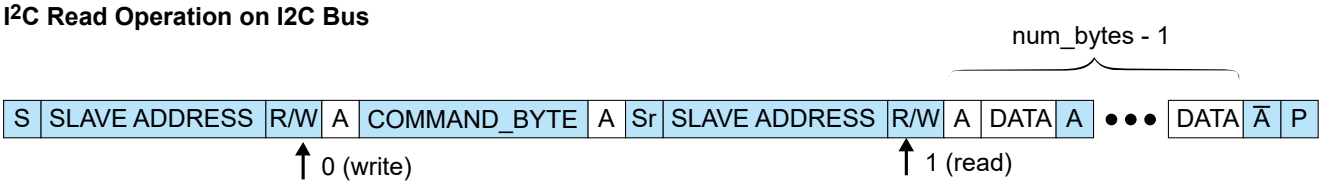
I²C Write and Read Operations

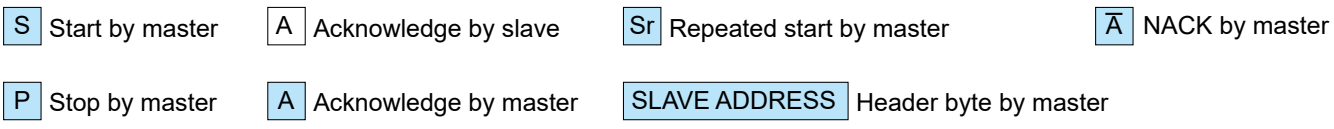
Figure 3: I²C Operations on I²C Bus

I²C Write Operation on I2C Bus



I²C Read Operation on I2C Bus





Performing a Write Operation on I²C Master

1. Ensure the `busy` signal is low.
2. Assign `din`, `command_byte`, `i2c_slave_addr` and `num_bytes`, then assert the `write` signal for one clock cycle.
3. Verify the status of the `busy` signal. If asserted, the I²C master sends out the write data to the I²C slave device.
4. Verify the status of the `write_done` signal. If asserted, the `din` is written completely. If you want to issue multiple I_2C write, insert new `din` value after `write_done` is high.
5. After the `busy` signal goes low, verify `i2c_arb_lost` and `i2c_rxak` signals are low.
6. The write data successfully sent out.

Figure 4: Write Operation on I²C Master Waveform

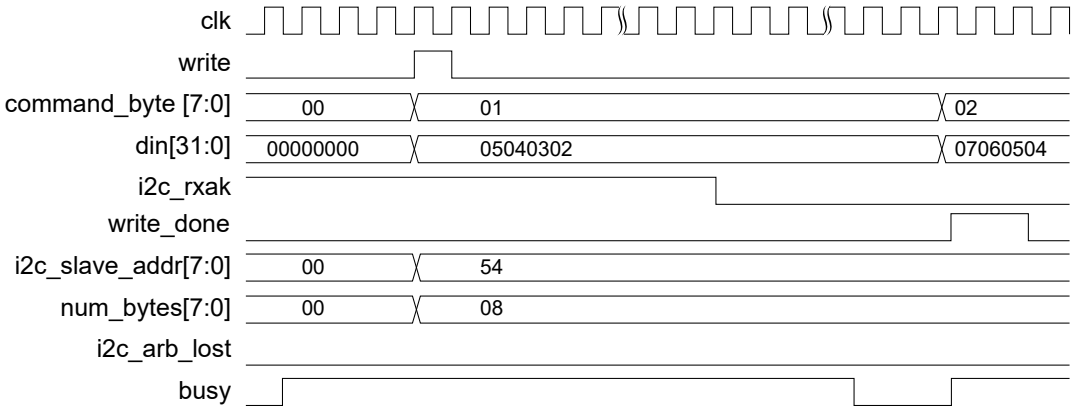
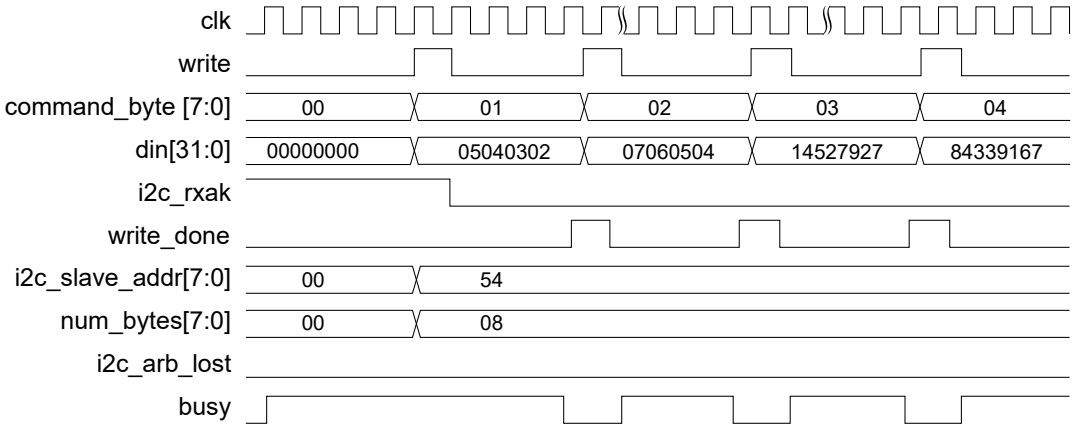


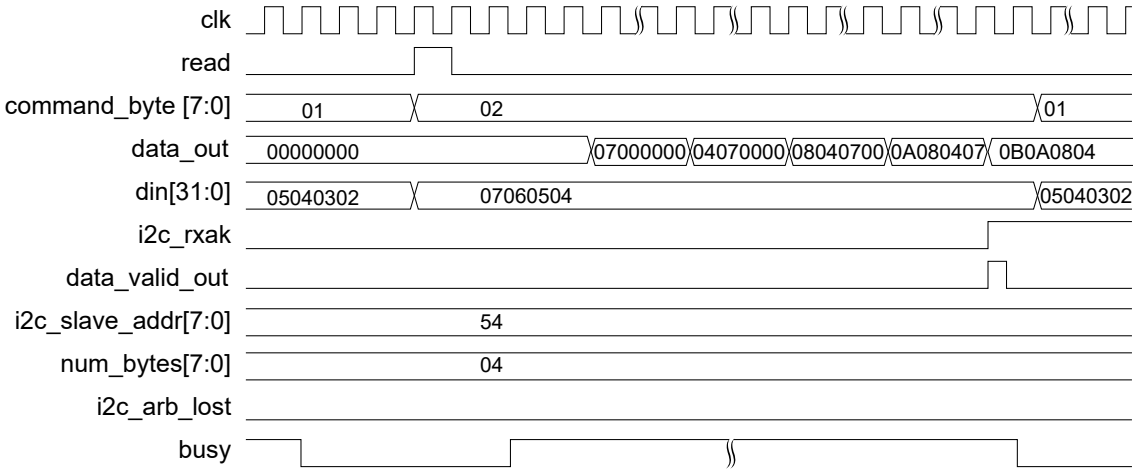
Figure 5: Multiple Write Operation on I²C Master Waveform



Performing a Read Operation on I²C Master

1. Ensure the `busy` signal is low.
2. Assign `command_byte`,`i2c_slave_addr` and `num_bytes`, then assert the `read` signal for one clock cycle.
3. Verify the status of the `busy` signal. If asserted, the I²C master reads from the `command_byte` value of the I²C slave device.
4. When `data_out_valid` signal is asserted, the `data_out` is a valid read data.
5. After the `busy` signal is low, verify the `i2c_arb_lost` signal is low, and the `i2c_rxak` signal is high.

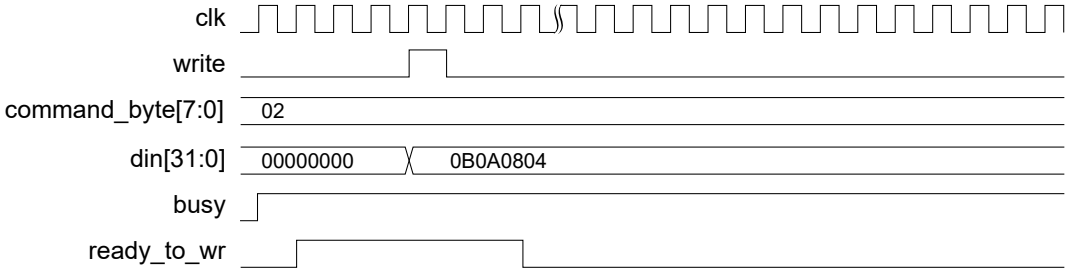
Figure 6: Read Operation on I²C Master Waveform



Performing a Write Operation on I²C Slave

1. To send the data back to the I²C master, you must provide the write data based on the `command_byte` value received.
2. Wait for the `ready_to_wr` signal to go high, then assert the `write` signal for one clock cycle.
3. Verify the status of the `busy` signal. If asserted, the I²C slave sends out the write data to the I²C master.
4. Verify the status of the `busy` signal. The write transfer is complete when the `busy` signal is low.

Figure 7: Write Operation on I²C Slave Waveform

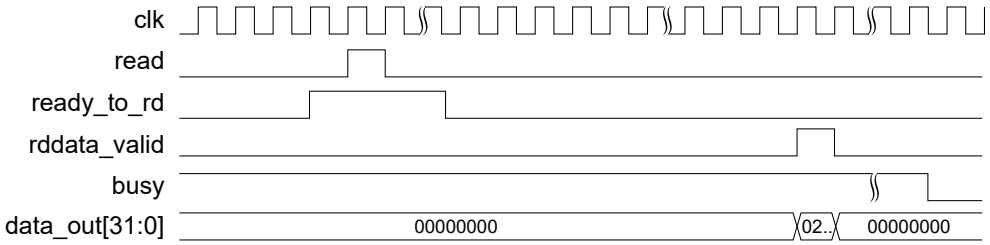


Performing a Read Operation on I²C Slave

1. Wait for the `ready_to_rd` signal to go high, then assert the `read signal` for one clock cycle.
2. Verify the status of the `busy` signal. If asserted, the I²C slave reads out the data transferred by the I²C master.
3. Verify the status of the `rddata_valid` signal. If asserted, the `data_out` is a valid read data.
4. Verify the status of the `busy` signal. The read transfer is complete when the `busy` signal is low.

Note: The I²C slave drops the additional byte if the I²C master sends more than DATA_BYTE_WIDTH/8 bytes of data.

Figure 8: Read Operation on I²C Slave Waveform



IP Manager

The Efinity® IP Manager is an interactive wizard that helps you customize and generate Efinix® IP cores. The IP Manager performs validation checks on the parameters you set to ensure that your selections are valid. When you generate the IP core, you can optionally generate an example design targeting an Efinix development board and/or a testbench. This wizard is helpful in situations in which you use several IP cores, multiple instances of an IP core with different parameters, or the same IP core for different projects.

Note: Not all Efinix IP cores include an example design or a testbench.

Generating a Core with the IP Manager

The following steps explain how to customize an IP core with the IP Configuration wizard.

1. Open the IP Catalog.
2. Choose an IP core and click **Next**. The **IP Configuration** wizard opens.
3. Enter the module name in the **Module Name** box.

   Note: You cannot generate the core without a module name.
4. Customize the IP core using the options shown in the wizard. For detailed information on the options, refer to the IP core's user guide or on-line help.
5. (Optional) In the **Deliverables** tab, specify whether to generate an IP core example design targeting an Efinix® development board and/or testbench. For SoCs, you can also optionally generate embedded software example code. These options are turned on by default.
6. (Optional) In the **Summary** tab, review your selections.
7. Click **Generate** to generate the IP core and other selected deliverables.
8. In the **Review configuration generation** dialog box, click **Generate**. The Console in the **Summary** tab shows the generation status.

   Note: You can disable the **Review configuration generation** dialog box by turning off the **Show Confirmation Box** option in the wizard.
9. When generation finishes, the wizard displays the **Generation Success** dialog box. Click **OK** to close the wizard.

The wizard adds the IP to your project and displays it under **IP** in the Project pane.

Generated Files

The IP Manager generates these files and directories:

- **<module name>_define.vh**—Contains the customized parameters.
- **<module name>_tmpl.v**—Verilog HDL instantiation template.
- **<module name>_tmpl.vhd**—VHDL instantiation template.
- **<module name>.v**—IP source code.
- **settings.json**—Configuration file.
- **<kit name>_devkit**—Has generated RTL, example design, and Efinity® project targeting a specific development board.
- **Testbench**—Contains generated RTL and testbench files.

Note: Refer to the IP Manager chapter of the Efinity® Software User Guide for more information about the Efinity® IP Manager.

Customizing the I^2C

The core has parameters so you can customize its function. You set the parameters in the General tab of the core's IP Configuration window.

Table 4: I^2C Core Master Parameters when I2C Controller Mode is MASTER

Parameters	Options	Description
I2C Data Transfer Speed	100 kHz normal mode, 400 kHz fast mode	I^2C data transfer speed. Default = 100 kHz normal mode
Data Width	8, 16, 24, 32	Data width for the user interface data input/output bus. Default = 32
Core Clock Frequency (MHz)	50, 100, 150	Core clock frequency. Default = 100
SDA/SCL Spike Filtering Cycle	1 - 15	SDA/SCL spike filtering logic to filter out signal spike in clock cycle with reference to core clock frequency. Default = 2

Table 5: I^2C Core Master Parameters when I2C Controller Mode is Slave

Parameters	Range	Description
8-bit I2C Slave Address	–	Slave address for the I^2C slave. The least significant bit is ignored. Default = 84 (decimal)
I2C Data Transfer Speed	0, 1	I^2C data transfer speed. Default = 100 kHz normal mode
Data Width	8, 16, 24, 32	Data width for the user interface data input/output bus. Default = 32
Core Clock Frequency (MHz)	50, 100, 150	Core clock frequency in MHz. Default = 100
SDA/SCL Spike Filtering Cycle	1 - 15	SDA/SCL spike filtering logic to filter out signal spike in clock cycle with reference to core clock frequency. Default = 2

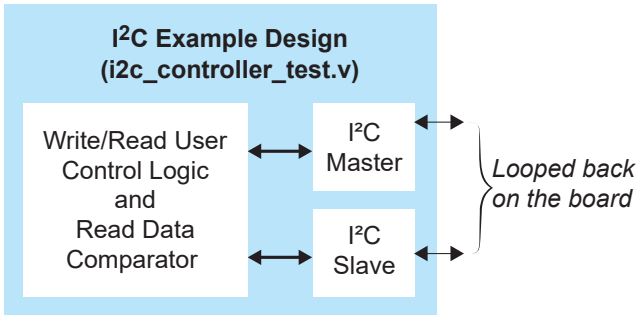
I²C Example Design

You can choose to generate the example design when generating the core in the IP Manager Configuration window. Compile the example design project and download the **.hex** or **.bit** file to your board.

Important: Efinix tested the example design generated with the default parameter options only.

The example designs the target the Trion® T20 BGA256 Development Board and Titanium Ti60 F225 Development Board.

Figure 9: Example Design Block Diagram



The example design flow consists of the following steps:

1. The user control logic asserts the `write` signal with `din`, `command_byte`, and `num_bytes` assigned to the I²C master.
2. The I²C master sends data to the I²C slave, once complete, the user control logic asserts `write_done` signal.
3. Once the `ready_to_rd` signal is high, the user control logic asserts the `read` signal to the I²C slave and start receiving the data from I²C master.
4. Once the `rddata_valid` signal is high, the user control logic compares the read data from the I²C slave with the write data written from I²C master.
5. The user control logic asserts the read signal with `command_byte` and `num_bytes` assigned to the I²C master.
6. The I²C master sends `command_byte` to the I²C slave.
7. Once the `ready_to wr` signal is high, the user control logic asserts the write signal with `din` to the I²C slave.
8. The I2C slave sends data to the I²C master.
9. Once the `data_out_valid` signal is high, the user control logic compares the read data from the I²C master with the write data written from I²C slave.
10. Once the `busy` signal is low, the example design operation is completed.

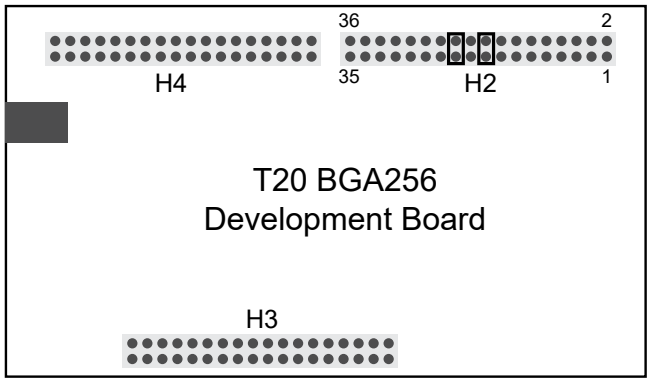
Trion® T20 BGA256 Development Board

External jumpers are required to connect the I²C SDA and SCL ports between master and slave at the Trion® T20 BGA256 Development Board. The following table describes the external jumper requirements for the example design.

Table 6: External Jumper for Trion® T20 BGA256 Development Board

Connection Port	Header	Jumper Setting
SDA	H2	Connect pins 17 and 18
SCL	H2	Connect pins 21 and 22

Figure 10: Jumper Connection Diagram



The LED displays the first data byte that the slave or master receive sequentially from LED D3, D4, D5 and D6 continuously.

Titanium Ti60 F225 Development Board

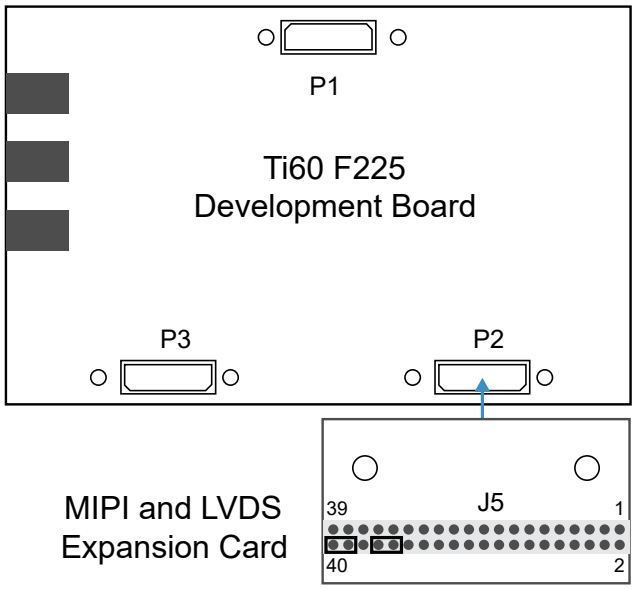
External jumpers are required to connect the I²C SDA and SCL ports between master and slave at the Titanium Ti60 F225 Development Board through the MIPI and LVDS Expansion Daughter Card. Connect the P3 header of the daughter card to the P2 header of the Titanium Ti60 F225 Development Board.

The following table describes the external jumper requirements at the MIPI and LVDS Expansion Daughter Card for the example design.

Table 7: External Jumper for MIPI and LVDS Expansion Daughter Card

Connection Port	Header	Jumper Setting
SDA	J5	Connect pins 32 and 34
SCL	J5	Connect pins 38 and 40

Figure 11: Jumper Connection Diagram



The LED displays the first data byte that the slave or master received in sequentially from LEDs D16 green and LED D17 white to LED D16 red and LED D17 yellow continuously.

Table 8: Trion® Example Design Implementation

FPGA	Mode	Logic Utilization (LUTs)	Registers	Memory Blocks	Multipliers	f_{MAX} (MHz)[3]	Efinity® Version[4]
T20 BGA256 C4	Master	723	414	0	0	95	2021.1
	Slave	722	414	0	0	93	

Table 9: Titanium Example Design Implementation

FPGA	Mode	Logic and Adders	Flip-flops	Memory Blocks	DSP48 Blocks	f_{MAX} (MHz)[3]	Efinity® Version[4]
Ti60 F225 C4	Master	703	414	0	0	332	2021.2
	Slave	700	414	0	0	339	

[3] Using default parameter settings.
[4] Using Verilog HDL.

I²C Testbench

You can choose to generate the testbench when generating the core in the IP Manager Configuration window.

Note: You must include all **.v** files generated in the **/testbench** directory in your simulation.

Efinix provides a simulation script for you to run the testbench quickly using the Modelsim software. To run the Modelsim testbench script, run `vsim -do modelsim.do` in a terminal application. You must have Modelsim installed in your computer to use this script.

The testbench provides read and write tests. Each test case indicates a pass or fail results for the register read/write tests. After running the simulation, the test prints the following message indicating the pass/fail results:

```
Slave received the command byte from Master 01
Slave received the data byte from Master 040302
Slave received the command byte from Master 02
Slave received the data byte from Master 060504
Master received the data byte from Slave, 4
```

Note: If you want to use your own testbench file, add the following line in your testbench file, *instancename*_**tb.v**:

```
`define SIM
```

Interface Designer GPIO Block Settings

The I²C SCL and SDA are bidirectional ports. When using the I²C core to communicate with I²C devices outside of the Trion® FPGA, set the GPIO block as follows:

1. In the Interface Designer, create a new GPIO block.
2. In the GPIO **Block Editor**, set the **Mode** to **inout**.
3. Select **weak pullup** in the **Pull Option** drop-down list.

Revision History

Table 10: Revision History

Date	Version	Description
February 2023	4.4	Added note about the resource and performance values in the resource and utilization table are for guidance only.
January 2022	4.3	Updated resource utilization table. (DOC-700)
October 2021	4.2	Added note to state that the f_{MAX} in Resource Utilization and Performance, and Example Design Implementation tables were based on default parameter settings. Updated design example target board to production Titanium Ti60 F225 Development Board and updated Resource Utilization and Performance, and Example Design Implementation tables. (DOC-553)
September 2021	4.1	Removed num_bytes [7:0] port possible values limitation.
June 2021	4.0	Added note about including all **.v** generated in testbench folder is required for simulation. Added write_done, data_out_valid, and slv_command_byte ports. Updated resource utilization and performance table. Updated example design output and implementation table. Added support for Titanium FPGAs and example design for Titanium Ti60 F225 Development Board. Added multiple write on master waveform. Updated for Efinity v2021.1.
December 2020	3.0	Added busy signal to the I^2C slave controller. Updated core name to I^2C core. Updated user guide for Efinix® IP Manager which includes added IP Manager topics, updated parameters, and user guide structure.
July 2020	2.0	Updated for I^2C Master/Slave Controller core v2.0. Added support for SDA and SCL spike filtering and SCL clock stretching. Updated LUTs utilization for master and slave mode in resource utilization and performance. Added MASTER_I2C_FAST_MODE, MASTER_CLOCK_FREQ, MASTER_SPIKE_FILTER_CYCLE, SLAVE_I2C_FAST_MODE, SLAVE_CLOCK_FREQ, and SLAVE_SPIKE_FILTER_CYCLE parameters. Updated LUTs and f_{MAX} in example design implementation. Updated example design block diagram to remove clock divider block. I^2C I2C core v2.0 supports core clock frequency of 150 Mhz and 100 Mhz and clock divider is no longer needed.
May 2020	1.0	Initial release.