# i2c_devreg

Documentation

# Content

## Table of Contents

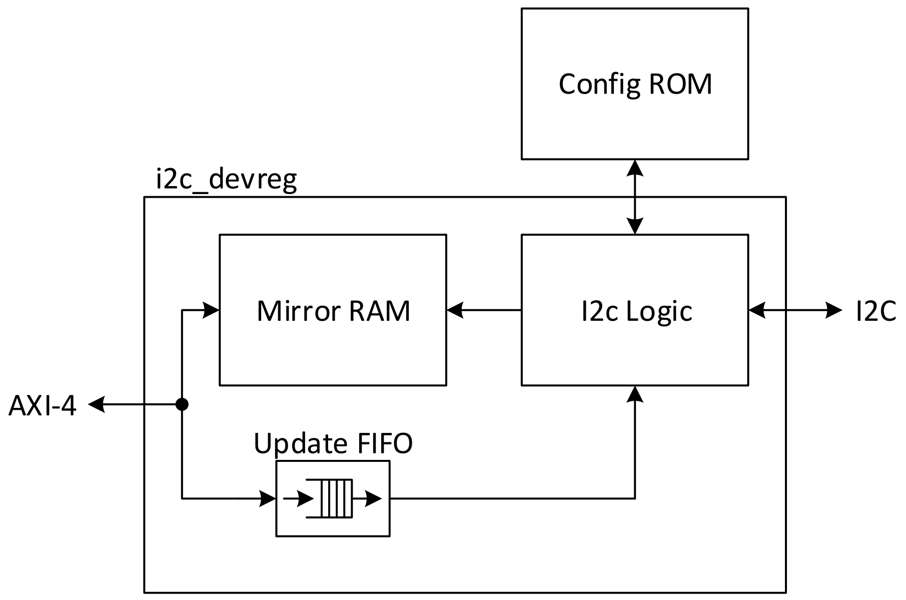
# 1 Introduction

In many systems I2C bus structures grow complex. Many devices are connected to the same bus, sometimes even more devices are connected through I2C multiplexers. For board-management functionality, often multiple registers per device must be read periodically. This leads to many I2C transfers and (depending on the I2C peripheral implementation) more or less significant IRQ load on the CPU. Thigs grow even more complex in multi-master I2C busses where transfers may be aborted in the middle because arbitration is lost and in this case must be restarted later on.

The aim of the *i2c_devreg* component is to offload as much of this load as possible to firmware. The component allows describing all device registers (resp. how they are accessed) in a ROM. They are then updated periodically without software interaction and their values are mirrored to a BRAM where the software can access them with low-latency. Writes from the software are forwarded to the I2C registers.

The configuration ROM containing the information about how to read a specific register is a separate component in order to keep *i2c_devreg* application independent. This Rom contains the device address, multiplexer settings and command bytes for every register. An example for such a ROM together with scripts that allow easily generating the ROM-code is providedinI the *i2c_devreg* repository.

The figure below shows the implementation principle.



If an update is triggered (either by an external input, via the AXI-4 register bank or through the internal timer), the I2C logic iterates through all ROM entries. Each ROM entry describes one I2C device register. For the entries where periodic write or read-back is enabled, the I2C logic executes the transfer required and writes the received value into the Mirror RAM (for read-backs). A transfer may consist of one or more of the following elements:

- Setting up an I2C mux (Mux address byte + 1 data byte)
- Sending command bytes (Device address byte + 1-4 command bytes)
- Receiving / sending data bytes (Device address byte + 1-4 data bytes)

The software can also request immediate read-back of one register or a write to a register. In case of a write, the write data is provided along with the request (and the Mirror RAM is updated with this data to reflect the device register state). The *i2c_devreg* then automatically sets the MUX (if used), sends the device address and preceding command bytes and then sends the data.

Immediate requests from the SW always have higher priority than periodic updates. So if a periodic update is ongoing while the software request an immediate access, the periodic update is paused after the current transfer is completed and continued after all immediate SW requests are executed.

The parts of an access (see bullet list above) are executed as follows: First the mux is set up (if required). After setting up the mux, a stop followed by a start condition is transmitted on the I2C bus. This is required since most multiplexers only switch the bus if it is idle. Then the command bytes are transmitted. Write data is transmitted directly after the command bytes, for reading data a repeated start condition is required first to change the direction of tranfers.

All parts of an access to a device register (see bullet list above) are chained together using I2C repeated-start conditions (not stop followed by start). This ensures that no other master can take over the bus in the middle.

The *i2c_devreg* component is fully multi-master capable. If it loses arbitration during a device register access, it immediately stops transmitting anything to the bus and retries executing the same access after the bus becomes free.

In case a device is not accessible (does not acknowledge), the *i2c_devreg* retries executing the same access a second time. If the second access fails too, the corresponding entry in the Mirror RAM is set to 0xFFFFFFFF and an error flag is set before normal operation is continued. This allows using the component even in cases where slave devices are not always present.

## 1.1 Feature List

- Multi-master capable
- Handling of one stage of I2C muxes
- Up to 4 command bytes prior to register access
- Up to 4 data bytes read in one access
- Periodic update of all registers
  - Triggered by internal timer
  - Triggered by external port
  - Triggered by register access
  - Can include writes (not only reads)
- Immediate actions can be requested by software
  - Read a specific register
  - Write a specific register
- Bus busy timeout
  - If not transfer is going on for a configurable time, the bus is regarded as free
- Scripted generation of configuration ROM
  - Including C-header file for SW

## 1.2 Functional Description

### 1.2.1 I2C Transfers

#### 1.2.1.1 General

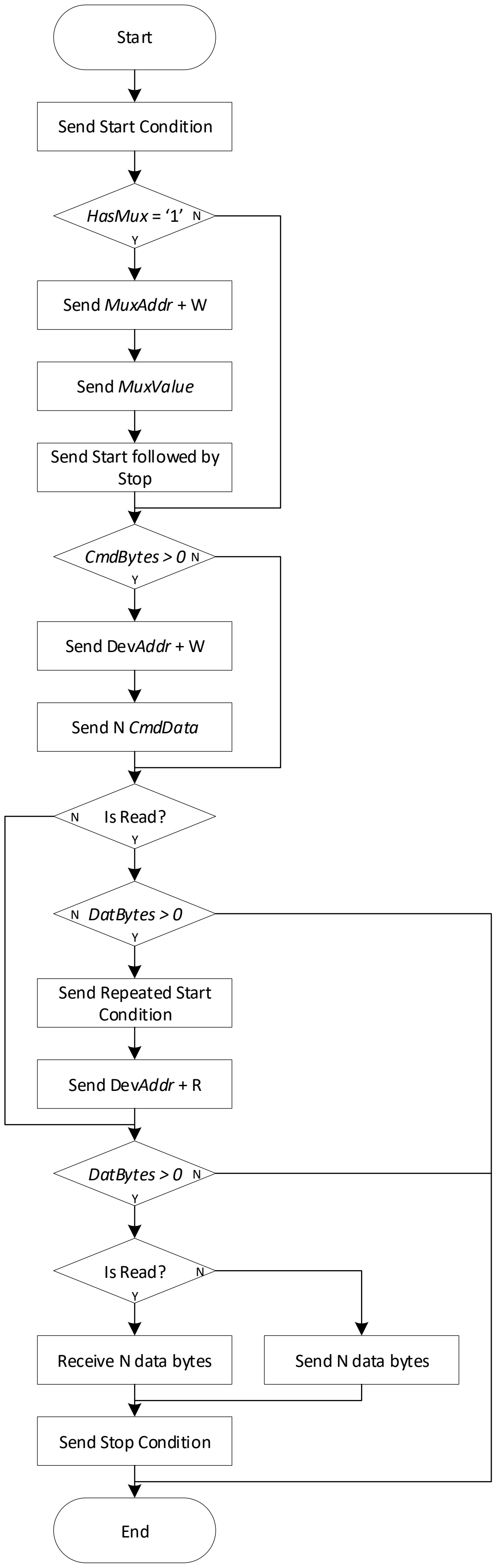
All I2C transfers are treated as MSB first with both meanings of MSB:

- Bytes are transferred with the most significant bit first
- If multiple bytes are transferred, the most significant byte is transferred first
    - 0x1234 means 0x12 is transferred first, 0x34 is transferred second
    - This applies to reads and writes

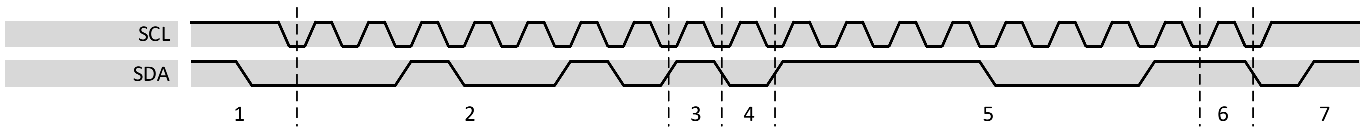
The I2C protocol itself is not explained in detail. If required, refer to one of the protocol specifications available online (google for "I2C specification").

#### 1.2.1.2 FSM

The diagram on the next page schematically shows the state machine that is executed when executing a transfer. The state machine is the same for automatic updates or SW triggered reads/writes.


Start
Send Start Condition
HasMux = '1'
N
Y
Send MuxAddr + W
Send MuxValue
Send Start followed by Stop
CmdBytes > 0
N
Y
Send DevAddr + W
Send N CmdData
N
Is Read?
Y
N
DatBytes > 0
Y
Send Repeated Start Condition
Send DevAddr + R
DatBytes > 0
N
Y
Is Read?
N
Y
Receive N data bytes
Send N data bytes
Send Stop Condition
End

### 1.2.1.3 1 Byte Read, No Mux, No Command Bytes

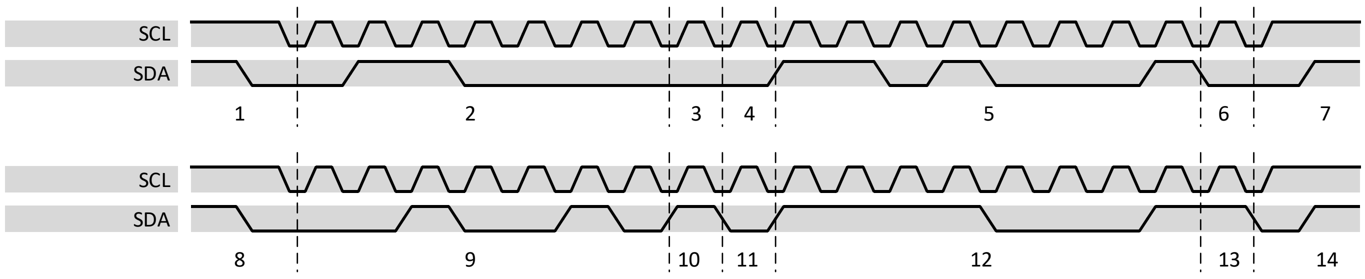


1. Start condition
2. 7-bit device address (0x12 in this case)
3. R/W bit (1 = read in this case)
4. Slave sends ACK to acknowledge address
5. Data byte (0xF1 in this case)
6. Master sends NACK since it is the last byte read
7. Stop condition

This corresponds to the following configuration ROM entry:

*HasMux* = 0
*DevAddr* = 0x12
*DatBytes* = 1

#### 1.2.1.4 1 Byte Read, With Mux, No Command Bytes

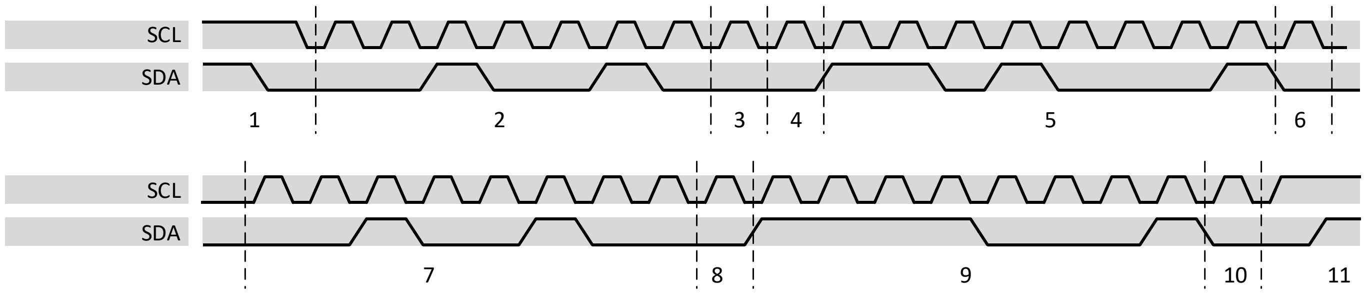


1. Start condition
2. 7-bit mux address (0x30 in this case)
3. R/W bit (0 = write because mux register must be written)
4. Mux sends ACK to acknowledge address
5. Mux data byte (0xD1 in this case)
6. Mux sends ACK to acknowledge data
7. Stop condition
8. Start condition
9. 7-bit device address (0x12 in this case)
10. R/W bit (1 = read in this case)
11. Slave sends ACK to acknowledge address
12. Data byte (0xF1 in this case)
13. Master sends NACK since it is the last byte read
14. Stop condition

This corresponds to the following configuration ROM entry:

*HasMux* = 1
*MuxAddr* = 0x30
*MuxValue* = 0xD1
*DevAddr* = 0x12
*DatBytes* = 1

### 1.2.1.5 2 Byte Write, No Mux, 1 Command Byte



1. Start condition
2. 7-bit device address (0x12 in this case)
3. R/W bit (0 = write in this case)
4. Slave sends ACK to acknowledge address
5. Command byte (0xD1 in this case)
6. Slave sends ACK to acknowledge command byte
7. First data byte follows without repeated start since we are still writing (0x24 in this case)
8. Slave sends ACK to acknowledge first data byte
9. Second data byte (0xF1 in this case)
10. Slave sends ACK to acknowledge second data byte
11. Stop condition

This corresponds to the following configuration ROM entry:

*HasMux* = 0
*CmdBytes* = 2
*CmdData* = 0xD1
*DevAddr* = 0x12
*DatBytes* = 1

In the case of this write, the data written is 0x24F1

## 1.2.2 Configuration ROM

### 1.2.2.1 Setup

To generate a project specific configuration ROM, follow the steps below:

1. Copy thefolder *<i2c_devreg>/example_rom* to a project specific location *<loc>*
2. Modify the constant *FW_LIB_PATH* in *<loc>/scripts/package.tcl* to point to the *Libraries/Firmware* directory that contains the PSI firmware libraries in your project.
3. Modify the file *<loc>/scripts/example_generator.py* in to describe the device registers to access
4. Run "*python3 example_generator.py"* to generate the HDL code and the C-header file
5. In the Vivado TCL console, navigate to *<loc>/scripts*
6. In the Vivado TCL console, run "*source ./package.tcl"* to re-package the IP-Core
7. The IP-core containing the configuration ROM is now ready to use
8. When writing software, include the header-file *<loc>/inc/*.h>* to guarantee consistence with the configuration ROM

Whenever you need to change the device registers accessed, re-run steps 38.

### 1.2.2.2 Config ROM Entry

Each entry in the configuration ROM has the following fields:

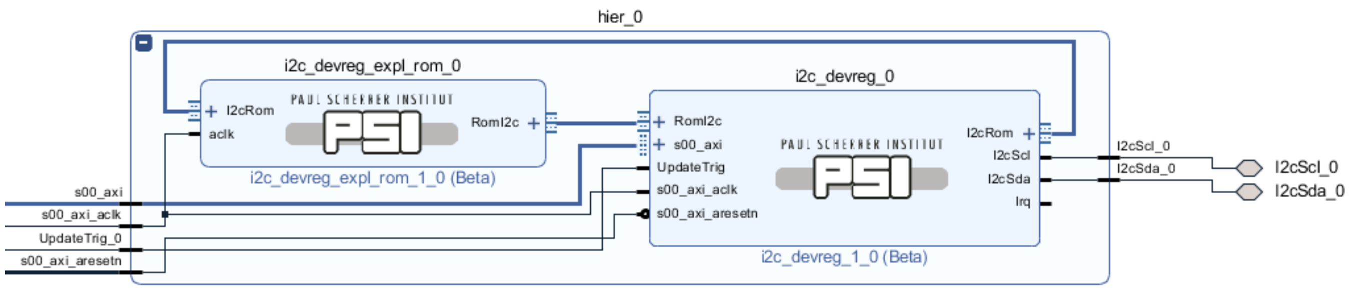
| | |
|---|---|
| **AutoRead** | If set to '1' the register is read during automatic update cycles. If set to '0', it is skipped. |
| **AutoWrite** | If set to '1' the register is written during automatic update cycles. This can be used if a register access requires a more complex sequence than covered by the command-bytes. |
| **HasMux** | If set to '1', a single byte access to configure an I2C multiplexer is executed prior to device access. If set to '0', no mux access is done and *MuxAddr* and *MuxValue* don't have any effect. |
| **MuxAddr** | I2C address of the multiplexer (only used if *HasMux* = '1' ) |
| **MuxValue** | Value to write to the I2C mux (only used if *HasMux* = '1') |
| **DevAddr** | I2C address of the device to access. |
| **CmdBytes** | Number of bytes to write prior to data access. For reads, a repeated start is done between the command bytes and the data. For writes, data bytes are sent after command bytes directly (without repeated start). |
| **CmdData** | Command bytes to send. Only the lowest *N* bytes are used. They are transmitted MSB first. |
| **DatBytes** | Number of data bytes to transfers. |

# 2 Interfaces

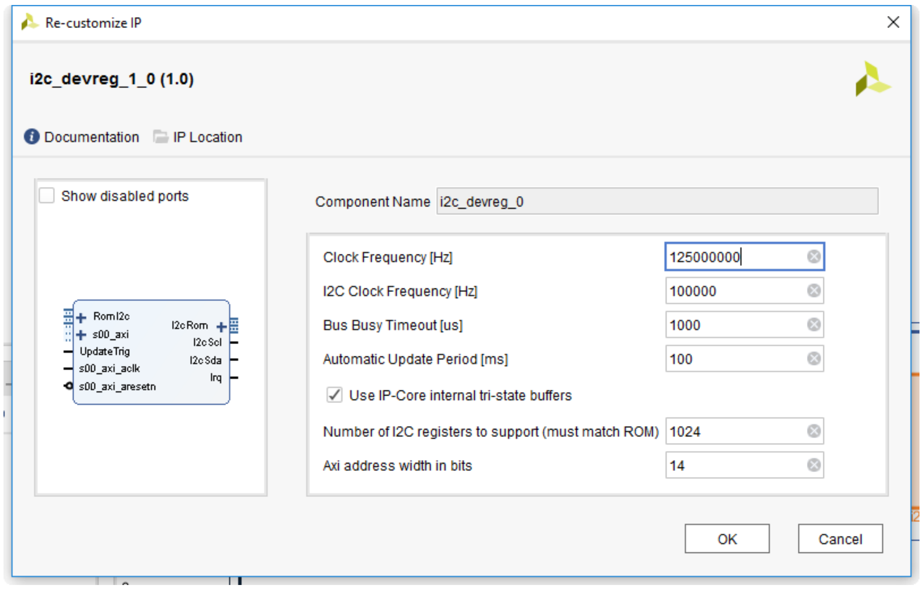
## 2.1 Vivado GUI

### 2.1.1 General Setup

The *i2c_devreg* IP is connected with the configuration ROM as shown in the picture below. It is recommended to put this pair into a hierarchy since for the system it represents one single component.



## 2.1.2 IP Configuration



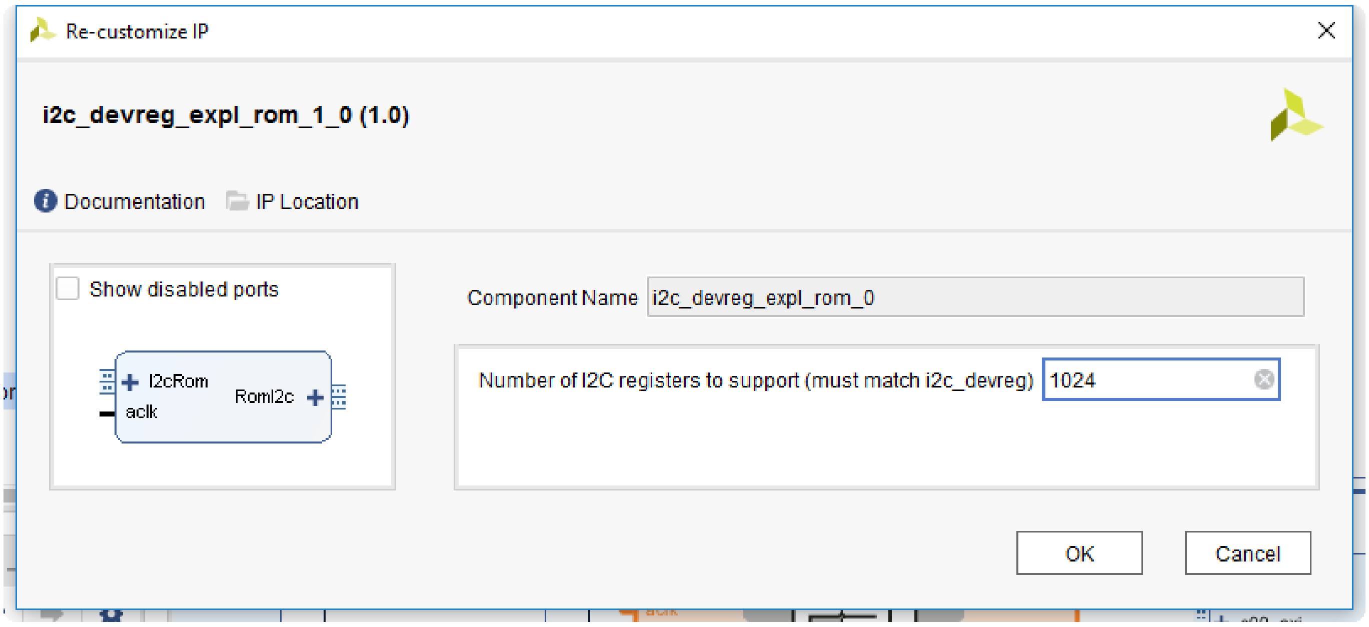
Component Name: i2c_devreg_0

Clock Frequency [Hz]: 125000000
I2C Clock Frequency [Hz]: 100000
Bus Busy Timeout [us]: 1000
Automatic Update Period [ms]: 100
[✓] Use IP-Core internal tri-state buffers
Number of I2C registers to support (must match ROM): 1024
Axi address width in bits: 14

This page contains the following parameters

- Clock Frequency
  - Frequency of the *s00_axi_aclk* port. This is used to generate the correct I2C SCL frequency.
- I2C Clock Frequency
  - Frequency of the I2C SCL signal to generate
- Bus Busy Timeout
  - If no transactions are detected on the I2C bus for this time, it is regarded as free. Even if there was a start condition detected that was not matched by any stop-condition. This can happen if devices are reset while a transaction is ongoing.
- Automatic Update Period
  - If no update is triggered by the *UpdateTrig* port or over the register bank for this time, a timer expires that triggers the update internally.
- Use IP-Core internal tri-state buffers
  - If this checkbox is selected, the tri-state buffers are included in the IP-Core and only SDA/SCL signals are exported. Choose this option if the IP-Core is connected to package pins directly.
  - If the checkbox is not selected, input (I), output (O) and tristate (T) signals are exported and the tristate buffer must be instantiated externally. Use this option if the signals are not connected to package pins directly (e.g. because multiple *i2c_devreg* components shall access the same bus).
- Number of I2C registers to support
  - Number of configuration ROM entries that are supported. The ROM must be configured with the same number. Use a value of at least 16.
- Axi address width in bits
  - Number of AXI address bits used. Choose at least $ceil(\log_2(NumberOfI2cRegisters)) + 3$

### 2.1.3 Configuration ROM


Re-customize IP

i2c_devreg_expl_rom_1_0 (1.0)

Component Name i2c_devreg_expl_rom_0

Number of I2C registers to support (must match i2c_devreg) 1024

The only thing to configure is the number of I2C registers supported. This value must match the setting in the *i2c_devreg* IP-Core.

## 2.2 Ports

| Name | Width | Direction | Description |
|---|---|---|---|
| ***Clock & Reset*** | | | |
| S00_axi_aclk | 1 | In | Clock input |
| S00_aresetn | 1 | In | Trigger automatic update |
| S00_axi | - | - | AXI-4 MM register bank interface |
| ***Parallel Control Signals*** | | | |
| UpdateTrig | 1 | In | Trigger automatic update (pulse high for 1 clock cycle) |
| Irq | 1 | Out | Level sensitive IRQ output |
| ***I2C Interface (with internal tri-state buffers)*** | | | |
| I2cScl | 1 | Bidir | I2C SCL Line (present if internal tristate-buffers are used) |
| I2cSda | 1 | Bidir | I2C SDA Line (present if internal tristate-buffers are used) |
| ***I2C Interface (with external tri-state buffers)*** | | | |
| I2cScl_I | 1 | In | I2C SCL Input (if internal tristate-buffers are not used) |
| I2cScl_O | 1 | Out | I2C SCL Output (if internal tristate-buffers are not used) |
| I2cScl_T | 1 | Out | I2C SCL Tristate (if internal tristate-buffers are not used) |
| I2cSda_I | 1 | In | I2C SDA Input (if internal tristate-buffers are not used) |
| I2cSda_O | 1 | Out | I2C SDA Output (if internal tristate-buffers are not used) |
| I2cSda_T | 1 | Out | I2C SDA Tristate (if internal tristate-buffers are not used) |
| ***Communication with configuration ROM*** | | | |
| I2cRom | - | Out | AXI-S interface for communication from *i2c_devreg* to config-ROM |
| RomI2c | - | In | AXI-S interface for communication from config-ROM to *i2c_Devreg* |

## 2.3 Drivers

For accessing the IP-Core from software, always use the drivers provided and the header file generated with the configuration ROM. Do not access the IP-Core registers directly except for debugging purposes.

Below is a code sample about how to use the code:

```
#include <example_rom_regs.h>
#include <xparameters.h>
#include <stdint.h>
#include <i2c_devreg.h>

...

void foo() {
   uint32_t data;
   I2cDevReg_RegGet(XPAR_I2C_DEVREG_0_BASEADDR, EXAMPLE_I2C_LM73_TEMP, *data);
}
```

The constant *EXAMPLE_I2C_LM73_TEMP* is generated by the ROM generation script and defined in the header-file *example_rom_regs.h*.

## 2.4 Register Bank

### 2.4.1 Overview

| Byte Address Offset | Name | Description |
|---|---|---|
| 0x000 | UPD_ENA | Enable automatic update |
| 0x004 | UPD_TRIG | Trigger automatic update |
| 0x008 | IRQ_ENA | Interrupt enable register |
| 0x00C | IRQ_VEC | Interrupt vector register |
| 0x010 | BUS_BUSY | I2C bus busy detection |
| 0x014 | ACC_FAIL | Access fail detection |
| 0x018 | FIFO_STATE | SW transaction FIFO state |
| 0x01C | UPD_ONGOING | Automatic update state |
| 0x020 | FORCE_READ | Force Readback |
| 0x040 - …. | MEM_OFS | Mirroring memory offset |

### 2.4.2 Register Descriptions

#### 2.4.2.1 UPD_ENA – Enable automatic update (0x00)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| ENA | 0 | RW | 0 | 1 Automatic update is enabled<br>0 Automatic update is disabled |

If automatic update is disabled, none of the update triggers have any effect. Neither a write to *UPD_TRIG*, nor pulsing the *UpdateTrig* port or the internal update timer.

#### 2.4.2.2 UPD_TRIG – Trigger automatic update (0x04)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| TRIG | 0 | W | - | 1 Trigger an automatic update cycle<br>0 No effect |

The automatic update is only triggered if *UPD_ENA* = 1.

#### 2.4.2.3 IRQ_ENA – Interrupt enable register (0x08)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| UPD | 0 | RW | 0 | 1 Fire IRQ when an update is completed<br>0 No action if an update is completed |
| FIFO_EMPTY | 8 | RW | 0 | 1 Fire IRQ when the FIFO becomes empty (i.e. all SW triggered operations are completed)<br>0 No action if the FIFO becomes empty |

#### 2.4.2.4 IRQ_VEC – Interrupt enable register (0x0C)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| UPD | 0 | RCW1 | 0 | 1 An update cycle completed<br>0 No update cycle completed since last clearing |
| FIFO_EMPTY | 8 | RCW1 | 0 | 1 FIFO became empty<br>0 FIFO did not become empty since last clearing |

Each bit must be cleared manually by writing a one to it.

Whenever a bit in *IRQ_VEC* is set and the corresponding bit in *IRQ_ENA* is set too, an IRQ is fired.

Bits in *IRQ_VEC* are set if the corresponding event occurs independently of *IRQ_ENA*. So when setting a bit in *IRQ_ENA* it is a good idea to clear the corresponding bit in *IRQ_VEC* first.

### 2.4.2.5 BUS_BUSY – Trigger automatic update (0x10)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| BUSY | 0 | R | - | 1 I2C bus is busy<br>0 I2C bus is free |

Busy means busy by another master or the *i2c_devreg* itself. However, during multiple accesses of the *i2c_devreg* the bus busy bit may pulse low as the bus is free between two accesses for a very short time. So don't rely on this bit staying constantly low during a whole automatic update cycle.

### 2.4.2.6 ACC_FAIL – Access fail detection (0x14)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| FAIL | 0 | RCW1 | - | 1 At least one access failed<br>0 No access failed |

When a device does not acknowledge an access twice in a row, the access is regarded as failed. In this case, the corresponding entry in the mirroring RAM is set to 0xFFFFFFFF and the *ACC_FAIL* flag is set. To clear the *ACC_FAIL* flag, write a 1 to it.

### 2.4.2.7 FIFO_STATE – SW transaction FIFO state (0x18)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| EMPTY | 0 | R | - | 1 FIFO is empty (all transactions executed)<br>0 FIFO is not empty |
| FULL | 8 | R | - | 1 FIFO is full<br>0 FIFO is not full |

All SW triggered transactions are stored in a FIFO. These are enforced reads (write to *FORCE_READ*) and writes (write to mirror memory). When the FIFO is full, no more of these operations are allowed until there is space in the FIFO.

Entries are removed from the FIFO if they are completed. So the FIFO being empty means that all SW triggered operations are completed.

### 2.4.2.8 UPD_ONGOING – Automatic update state (0x1C)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| ONGOING | 0 | R | - | 1 Automatic update is ongoing<br>0 No automatic update is ongoing |

### 2.4.2.9 FORCE_READ – Force readback (0x20)

| Field | Bit(s) | Type | Reset | Description |
|---|---|---|---|---|
| ADDR | N-1…0 | W | - | Index of the register entry to update |

If this register is written, the I2C register described in the ROM entry with the given index is read and the mirroring RAM entry is updated.

### 2.4.2.10 MEM_OFFS – Mirroring memory offset (0x40 … )

The mirroring memory is placed at this offset. Reading from such an address means reading from the mirroring RAM, writing to such an address means placing a write access in the FIFO. When the access is executed, the mirroring RAM entry is updated with the corresponding value.

The configuration ROM / mirroring RAM index to be updated is calculated by the formula below:

$$Index = (Addr - MEM_{OFFS})/4$$

So an access to 0x40 means updating Index 0.

An access to 0x58 means updating Index 6.