Application Note: AN00157

How to use the I2C slave library

Required tools and libraries

The code in this application note is known to work on version 14.2.0 of the xTIMEcomposer tools suite, it may work on other versions.

The application depends on the following libraries:

- lib_logging
- lib_i2c

Required hardware

There is no hardware requirement for this application note. It has been designed to run on the simulator using the loopback plugin.

Prerequisites

- This document assumes familarity with I^2C interfaces, the XMOS xCORE architecture, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- For a description of XMOS related terms found in this document please see the XMOS Glossary[1].

[1] http://www.xmos.com/published/glossary

1 Overview

1.1 Introduction

The XMOS I²C library provides software defined, industry-standard, I²C components that allow you to create devices which can be either I²C bus masters or slaves using xCORE GPIO ports.

I²C is a two-wire bus with defined protocols for connecting devices. There is a clock and a data line, both of which are pulled high by external pull-up resistors and driven low by the I²C devices.

The XMOS I²C library includes support for master and slave devices at speeds of up to 400kb/s.

This application note demonstrates how to create an I²C slave device using the XMOS I²C library. It creates an example register file which can be read and written by both I²C masters and a user application on the xCORE.

The application note includes an I²C master which accesses the register file by looping back the clock and data lines between the master and slave. This is done in simulation using a loopback with pull-up enabled.

1.2 Block diagram

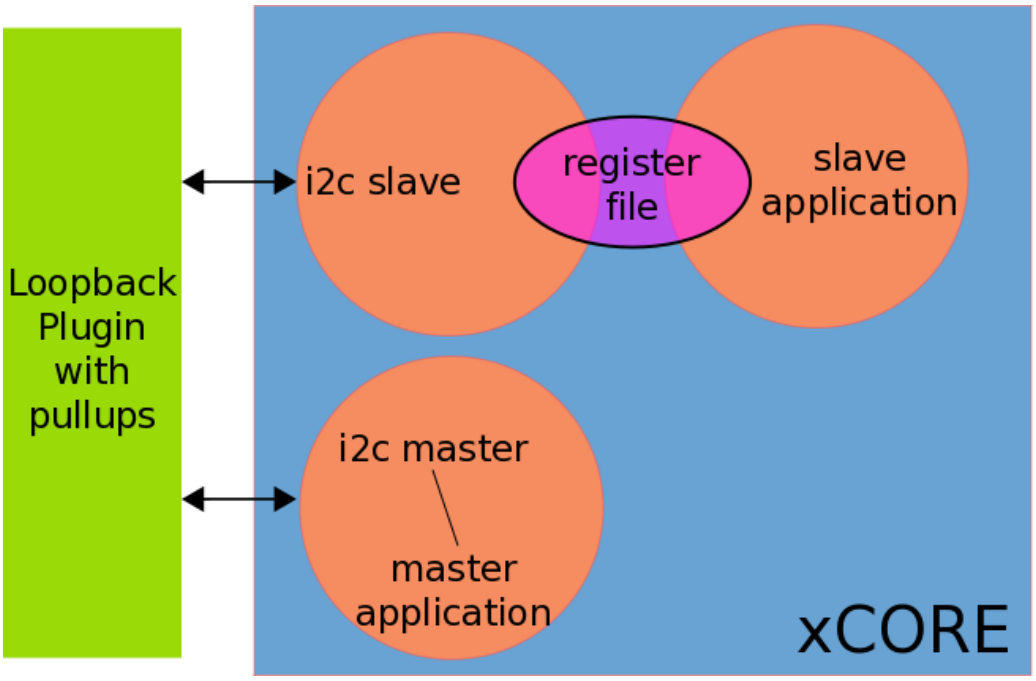

Figure 1: Application block diagram

The application uses three logical cores:

- The first core implements an I²C master and a test application performing register reads and writes.
- The second core is used by the I²C slave.
- The third core is used by an example host application which uses the register file.

The register file is distributed between the two logical cores using it.

2 How to use I²C slave

2.1 The Makefile

To start using the I²C, you need to add `lib_i2c` to you Makefile:

```
USED_MODULES = .. lib_i2c ...
```

This demo also uses the logging library (`lib_logging`) for the debug_printf function. This is a faster, but restricted version of the C-Standard Library `printf` function. So the Makefile also includes:

```
USED_MODULES = .. lib_logging ..
```

The logging library is configurable at compile-time allowing calls to debug_printf() to be easily enabled or disabled. For the prints to be enabled it is necessary to add the following to the compile flags:

```
XCC_FLAGS = .. -DDEBUG_PRINT_ENABLE=1 ..
```

2.2 Includes

This application requires a few system headers. XMOS xCORE specific defines for declaring and initialising hardware are defined in `xs1.h` and the `_exit()` function which allows the application to terminate once the demo is complete is provided by `syscall.h`.

```
#include <xs1.h>
#include <syscall.h>
```

The I²C library functions are defined in `i2c.h`. This header must be included in your code to use the library. The logging functions are provided by debug_print.h.

```
#include "i2c.h"
#include "debug_print.h"
```

2.3 Allocating hardware resources

An I²C interface requires both a clock and data pin. In this application note there is a slave and a master and each require both a clock and data pin. On an xCORE the pins are controlled by `ports`. The application therefore declares four 1-bit ports:

```
port p_slave_scl = XS1_PORT_1E;
port p_slave_sda = XS1_PORT_1F;

port p_master_scl = XS1_PORT_1G;
port p_master_sda = XS1_PORT_1H;
```

2.4 The register file interfaces

The register file will interact with both the I^2C slave and the user application. The interface to the I^2C slave is defined `i2c.h`. The interface to the user application is a custom interface which is declared in the application:

```
typedef interface register_if {
  /* Set a register value
   */
  void set_register(int regnum, uint8_t data);

  /* Get a register value.
   */
  uint8_t get_register(int regnum);

  /* Get the number of the register that has changed
   * This will also clear the notification
   */
  [[clears_notification]]
  unsigned get_changed_regnum();

  /* Notification from the register file to the application that a register
   * value has changed
   */
  [[notification]]
  slave void register_changed();
} register_if;
```

This interface declares four functions, three of which the user application can use to acces the register file and one which allows the register file to notify the user application that something has changed. The `notification` call means that the user application does not have to waste cycles polling the register file to determine if a register value has changed.

2.5 The register file implementation

The register file has a fixed number of registers that are assumed to start at address 0. The number of registers is controlled by the define:

```
#define NUM_REGISTERS 10
```

The register file implementation fuction is declared as:

```
[[distributable]]
void i2c_slave_register_file(server i2c_slave_callback_if i2c,
                             server register_if app)
```

The xC keyword `distributable` means that this function does not need to consume a whole logical core itself, but the functions it defines can be called in-line on other logical cores that are using its interface functions.

The register file is declared with the `i2c` and app interfaces described above. In both cases the interfaces are declared as `server` so the register file will respond to the function calls but not initiate any. The only exception is the `notification` call where the register file informs the user application that a register has changed.

The register file then declares storage for the registers:

```
uint8_t registers[NUM_REGISTERS];
```

The interface to the I²C slave needs some local state to track which register is being accessed. An I²C master will first perform a write operation with the register number being used and then either write data to that register or read from that register. The `current_regnum` variable is used to record which register is being read/written.

The `changed_regnum` is used by the interface to the user application to record which register was last modified:

```
int current_regnum = -1;
int changed_regnum = -1;
```

And the rest of the register file is a `while(1) { select` loop. All `distributable` functions must be of this form, with declarations followed by a `while(1) { select` loop.

Within the loop the register file handles the functions provided by both interfaces. First the interface to the user application:

```
case app.set_register(int regnum, uint8_t data):
  if (regnum >= 0 && regnum < NUM_REGISTERS) {
    registers[regnum] = data;
  }
  break;
case app.get_register(int regnum) -> uint8_t data:
  if (regnum >= 0 && regnum < NUM_REGISTERS) {
    data = registers[regnum];
  } else {
    data = 0;
  }
  break;
case app.get_changed_regnum() -> unsigned regnum:
  regnum = changed_regnum;
  break;
```

The interface to the application is very straight-forward, with each function validating that the `regnum` specified is valid and then performing the requested operation.

It is worth noting the syntax of the `app.get_register` function which is of the form:

```
case app.get_register(int regnum) -> uint8_t data:
```

The `->` operator indicates the result of the `get_register` function is returned in the `data` variable. The assignment to the `data` variable is the data that will be returned.

The I²C slave callback interface declares nine functions that the register file must implement. Of those functions, only five are actually used by the register file, but the others would be used by other I²C slave devices.

```
case i2c.ack_read_request(void) -> i2c_slave_ack_t response:
  // If no register has been selected using a previous write
  // transaction the NACK, otherwise ACK
  if (current_regnum == -1) {
    response = I2C_SLAVE_NACK;
  } else {
    response = I2C_SLAVE_ACK;
  }
  break;
```

```
case i2c.ack_write_request(void) -> i2c_slave_ack_t response:
  // Write requests are always accepted
  response = I2C_SLAVE_ACK;
  break;
```

The register file implements the ack_read_request() and ack_write_request() functions to control whether an ACK or NACK is signaled on the bus for each start of read or write request. All writes are accepted, and reads are only accepted if the register number has been selected first.

```
case i2c.master_sent_data(uint8_t data) -> i2c_slave_ack_t response:
  // The master is trying to write, which will either select a register
  // or write to a previously selected register
  if (current_regnum != -1) {
    registers[current_regnum] = data;
    debug_printf("REGFILE: reg[%d] <- %x\n", current_regnum, data);

    // Inform the user application that the register has changed
    changed_regnum = current_regnum;
    app.register_changed();

    response = I2C_SLAVE_ACK;
  }
  else {
    if (data < NUM_REGISTERS) {
      current_regnum = data;
      debug_printf("REGFILE: select reg[%d]\n", current_regnum);
      response = I2C_SLAVE_ACK;
    } else {
      response = I2C_SLAVE_NACK;
    }
  }
  break;
```

The master_sent_data() interface function is called for each byte received by the I^2C slave. The first byte is assumed to be a register number. Subsequent bytes received before the stop_bit() function is called are assumed to be data to write to the specified register.

If a register is written then the master_sent_data() function also informs the user application if by calling the app.register_changed() notification function after having stored which register was modified in the changed_regnum variable.

```
case i2c.master_requires_data() -> uint8_t data:
  // The master is trying to read, if a register is selected then
  // return the value (other return 0).
  if (current_regnum != -1) {
    data = registers[current_regnum];
    debug_printf("REGFILE: reg[%d] -> %x\n", current_regnum, data);
  } else {
    data = 0;
  }
  break;
```

The `master_requires_data()` interface function is called whenever the I^{2}C slave is required to send a byte to the master. If a register number has been selected first it will read that register, otherwise it returns 0.

```
case i2c.stop_bit():
  // The I2C transaction has completed, clear the regnum
  debug_printf("REGFILE: stop_bit\n");
  current_regnum = -1;
  break;
```

Finally, the `stop_bit()` function is called to indicate that the I^{2}C transaction is finished. The register file clears the `current_regnum` variable to ensure it has to be set for the next transaction.

2.6 Slave application

The slave application gives an example of how the register file could be used. It is passed a `client` interface to the register file so that it can access the registers. It then sits in a `while(1)` loop inverting the data that is written to the register file and printing when it does this:

```
void slave_application(client register_if reg)
{
  // Invert the data of any register that is written
  while (1) {
    select {
      case reg.register_changed():
        unsigned regnum = reg.get_changed_regnum();
        unsigned value = reg.get_register(regnum);
        debug_printf("SLAVE: Change register %d value from %x to %x\n",
          regnum, value, ~value & 0xff);
        reg.set_register(regnum, ~value);
        break;
    }
  }
}
```

2.7 Master application

The master application is providing stimulus to test the slave register file and application:

```
void master_application(client interface i2c_master_if i2c, uint8_t device_addr)
```

It is passed the interface to the I^{2}C master and the device address of the slave that it is testing. It then performs a single register write and checks that it completes successfully:

```
reg_result = i2c.write_reg(device_addr, 0x03, 0x12);
if (reg_result != I2C_REGOP_SUCCESS) {
  debug_printf("Write reg 0x03 failed!\n");
}
```

Then it reads the register value back and ensures the read completed correctly and that the data has been inverted:

```
uint8_t data = i2c.read_reg(device_addr, 0x03, reg_result);
if (reg_result != I2C_REGOP_SUCCESS) {
  debug_printf("Read reg 0x03 failed!\n");
}
debug_printf("MASTER: Read from addr 0x%x, 0x%x %s (got 0x%x, expected 0x%x)\n",
  device_addr, 0x03, (data == 0xed) ? "SUCCESS" : "FAILED", data, 0xed);
```

2.8 The application main() function

The `main()` function sets up the tasks in the application.

Firstly, the `interfaces` are declared. In xC interfaces provide a means of concurrent tasks communicating with each other. In this application there is an interface to the I^{2}C slave, an interface between the application and the register file, and an interface to the I^{2}C master.

```
i2c_slave_callback_if i_i2c;
register_if i_reg;
i2c_master_if i2c[1];
```

The address of the I^{2}C slave is declared:

```
uint8_t device_addr = 0x3c;
```

The rest of the `main()` function starts all the tasks in parallel using the xC `par` construct:

```
par {
  i2c_slave_register_file(i_i2c, i_reg);
  i2c_slave(i_i2c, p_slave_scl, p_slave_sda, device_addr);
  slave_application(i_reg);

  i2c_master(i2c, 1, p_master_scl, p_master_sda, 200);
  master_application(i2c[0], device_addr);
}
```

This code starts the I^{2}C slave, register file, slave application, I^{2}C master, and master application. Because the register file is marked as `distributable` it will not actually use a logical core but will be run on the logical cores with the I^{2}C slave and register file. Likewise, the I^{2}C master is marked as `distributable` and will share the logical core with the master application. So the entire system only requires three logical cores.

APPENDIX A - Demo setup

A.1 Using xTIMEcomposer

The demo is designed to run on the XMOS simulator using a loopback plugin providing pull-ups on the pins. In xTIMEcomposer:

- Import and select the AN00157_i2c_slave_example project
- Select `Project -> Build Project`
- Select `Run -> Run Configurations...`
- Select the `New launch configuration` button

Figure 2: New launch configuration

Then on the `Main` tab select: `Run on: simulator`

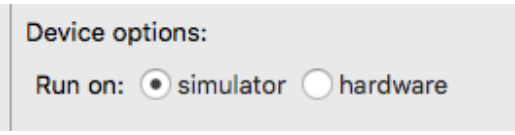

Figure 3: Run on simulator

On the `Simulator` tab enable `Trace to file:`

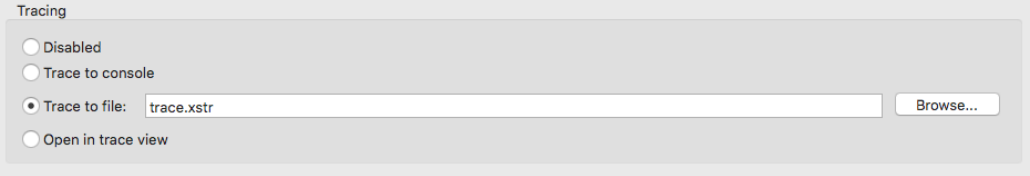

Figure 4: Trace to file

On the Simulator tab, Singal Tracing section, select Enable signal tracing. Then under the Tile Trace Options click Add and on tile[0] enable the Ports checkbox.

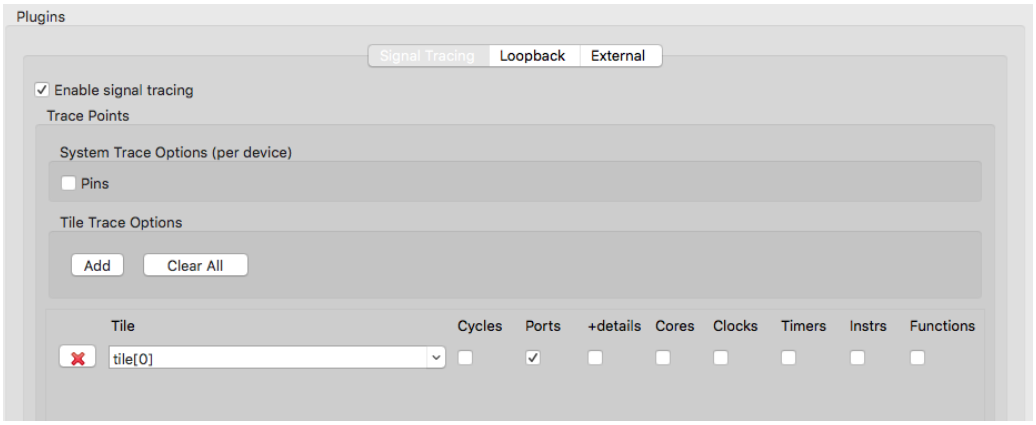

Figure 5: Signal tracing

On the Simulator tab, Loopback section, select Enable pin connections. Then under the Pin Connetions click Add and create a loopback from tile[0], XS1_PORT_1E to tile[0], XS1_PORT_1G and enable pullup. Again, under the Pin Connetions click Add and create a loopback from tile[0], XS1_PORT_1F to tile[0], XS1_PORT_1H and enable pullup.

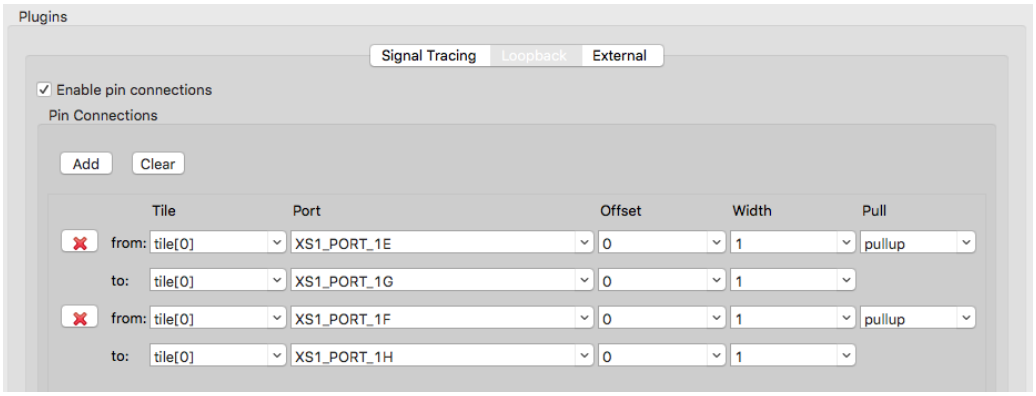

Figure 6: Loopback

Click Apply and then click Run. The application will run and generate the following output:

```
REGFILE: select reg[3]
REGFILE: reg[3] <- 12
SLAVE: Change register 3 value from 12 to ED
REGFILE: stop_bit
REGFILE: select reg[3]
REGFILE: reg[3] -> ED
REGFILE: stop_bit
MASTER: Read from addr 0x3C, 0x3 SUCCESS (got 0xED, expected 0xED)
```

When it finishes, it will switch to the VCD trace viewer and open the signal file created. Hide the console by clicking on the console button on the right-hand side of the window:

Figure 7: Hide console

An existing waveform setup can be loaded using the `Read Session File` button from the `Waves` window:

Figure 8: Read session file

and open the `xvcd.xml` file. This configures the waves to show the clock and data waves and the direction of the ports.

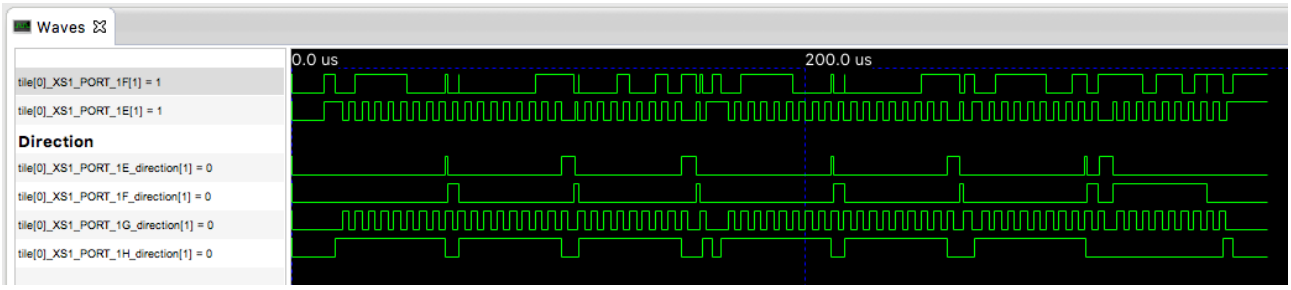

Figure 9: Waves

A.2 On the command-line

On the command-line run change to the application note folder and run:

```
xmake
```

Then run `xsim` with the following arguments:

```
xsim --trace-to trace.txt --plugin LoopbackPort.so
  "-pullup -port tile[0] XS1_PORT_1E 1 0 -port tile[0] XS1_PORT_1G 1 0
   -pullup -port tile[0] XS1_PORT_1F 1 0 -port tile[0] XS1_PORT_1H 1 0"
   bin/AN00157_i2c_slave_example.xe
```

If you would like to look at the waveforms to further understand how the I^2C is working then add the following to the xsim command arguments before the name of the binary:

```
--vcd-tracing "-o trace.vcd -tile tile[0] -ports"
```

The VCD file can be opened in xTIMEcomposer or any other VCD viewer.

APPENDIX B - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

XMOS I^2C Library

http://www.xmos.com/support/libraries/lib_i2c

APPENDIX C - Full source code listing

C.1 Source code for main.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <xs1.h>
#include <syscall.h>

#include "i2c.h"
#include "debug_print.h"

port p_slave_scl = XS1_PORT_1E;
port p_slave_sda = XS1_PORT_1F;

port p_master_scl = XS1_PORT_1G;
port p_master_sda = XS1_PORT_1H;

/*
 * Interface definition between user application and I2C slave register file
 */
typedef interface register_if {
  /* Set a register value
   */
  void set_register(int regnum, uint8_t data);

  /* Get a register value.
   */
  uint8_t get_register(int regnum);

  /* Get the number of the register that has changed
   * This will also clear the notification
   */
  [[clears_notification]]
  unsigned get_changed_regnum();

  /* Notification from the register file to the application that a register
   * value has changed
   */
  [[notification]]
  slave void register_changed();
} register_if;

#define NUM_REGISTERS 10

[[distributable]]
void i2c_slave_register_file(server i2c_slave_callback_if i2c,
                             server register_if app)
{
  uint8_t registers[NUM_REGISTERS];

  // This variable is set to -1 if no current register has been selected.
  // If the I2C master does a write transaction to select the register then
  // the variable will be updated to the register the master wants to
  // read/update.
  int current_regnum = -1;
  int changed_regnum = -1;
  while (1) {
    select {

    // Handle application requests to get/set register values.
    case app.set_register(int regnum, uint8_t data):
      if (regnum >= 0 && regnum < NUM_REGISTERS) {
        registers[regnum] = data;
      }
      break;
    case app.get_register(int regnum) -> uint8_t data:
      if (regnum >= 0 && regnum < NUM_REGISTERS) {
        data = registers[regnum];
      } else {
        data = 0;
      }
      break;
    case app.get_changed_regnum() -> unsigned regnum:
```

```
      regnum = changed_regnum;
      break;

    // Handle I2C slave transactions
    case i2c.start_read_request(void):
      break;
    case i2c.ack_read_request(void) -> i2c_slave_ack_t response:
      // If no register has been selected using a previous write
      // transaction the NACK, otherwise ACK
      if (current_regnum == -1) {
        response = I2C_SLAVE_NACK;
      } else {
        response = I2C_SLAVE_ACK;
      }
      break;
    case i2c.start_write_request(void):
      break;
    case i2c.ack_write_request(void) -> i2c_slave_ack_t response:
      // Write requests are always accepted
      response = I2C_SLAVE_ACK;
      break;
    case i2c.start_master_write(void):
      break;
    case i2c.master_sent_data(uint8_t data) -> i2c_slave_ack_t response:
      // The master is trying to write, which will either select a register
      // or write to a previously selected register
      if (current_regnum != -1) {
        registers[current_regnum] = data;
        debug_printf("REGFILE: reg[%d] <- %x\n", current_regnum, data);

        // Inform the user application that the register has changed
        changed_regnum = current_regnum;
        app.register_changed();

        response = I2C_SLAVE_ACK;
      }
      else {
        if (data < NUM_REGISTERS) {
          current_regnum = data;
          debug_printf("REGFILE: select reg[%d]\n", current_regnum);
          response = I2C_SLAVE_ACK;
        } else {
          response = I2C_SLAVE_NACK;
        }
      }
      break;
    case i2c.start_master_read(void):
      break;
    case i2c.master_requires_data() -> uint8_t data:
      // The master is trying to read, if a register is selected then
      // return the value (other return 0).
      if (current_regnum != -1) {
        data = registers[current_regnum];
        debug_printf("REGFILE: reg[%d] -> %x\n", current_regnum, data);
      } else {
        data = 0;
      }
      break;
    case i2c.stop_bit():
      // The I2C transaction has completed, clear the regnum
      debug_printf("REGFILE: stop_bit\n");
      current_regnum = -1;
      break;
    } // select
  }
}

void slave_application(client register_if reg)
{
  // Invert the data of any register that is written
  while (1) {
    select {
      case reg.register_changed():
        unsigned regnum = reg.get_changed_regnum();
        unsigned value = reg.get_register(regnum);
```

```
        debug_printf("SLAVE: Change register %d value from %x to %x\n",
          regnum, value, ~value & 0xff);
        reg.set_register(regnum, ~value);
        break;
    }
  }
}

void master_application(client interface i2c_master_if i2c, uint8_t device_addr)
{
  i2c_regop_res_t reg_result;

  // Write a single register
  reg_result = i2c.write_reg(device_addr, 0x03, 0x12);
  if (reg_result != I2C_REGOP_SUCCESS) {
    debug_printf("Write reg 0x03 failed!\n");
  }

  // Read a single register and check the result
  uint8_t data = i2c.read_reg(device_addr, 0x03, reg_result);
  if (reg_result != I2C_REGOP_SUCCESS) {
    debug_printf("Read reg 0x03 failed!\n");
  }
  debug_printf("MASTER: Read from addr 0x%x, 0x%x %s (got 0x%x, expected 0x%x)\n",
    device_addr, 0x03, (data == 0xed) ? "SUCCESS" : "FAILED", data, 0xed);

  // Test finished
  _exit(0);
}

int main() {
  i2c_slave_callback_if i_i2c;
  register_if i_reg;
  i2c_master_if i2c[1];
  uint8_t device_addr = 0x3c;

  par {
    i2c_slave_register_file(i_i2c, i_reg);
    i2c_slave(i_i2c, p_slave_scl, p_slave_sda, device_addr);
    slave_application(i_reg);

    i2c_master(i2c, 1, p_master_scl, p_master_sda, 200);
    master_application(i2c[0], device_addr);
  }
  return 0;
}
```