Application Note: AN00161

How to use the SPI library as SPI slave

This application note shows how to use the SPI library to make the xCORE use an SPI bus as SPI slave. The application provides a register file that can be read and written by the internal application and by the SPI master using a simple command set. The code is run in simulation with an SPI master output looped-back onto the SPI slave input to show the bus functioning.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.0
- XMOS SPI library - Version 3.0.0

Required hardware

This application note is designed to run in simulation so requires no XMOS hardware.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the SPI bus protocol, 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 descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- For the full API listing of the XMOS SPI Device Library please see the library user guide[2].

[1] http://www.xmos.com/published/glossary
[2] http://www.xmos.com/support/libraries/lib_spi

1 Overview

1.1 Introduction

The XMOS SPI library is a library that provides software defined, industry-standard, SPI (serial peripheral interface) components that allows you to control an SPI bus via the xCORE GPIO hardware-response ports. SPI is a four-wire hardware bi-directional serial interface.

The SPI bus can be used by multiple tasks within the xCORE device and (each addressing the same or different slaves) and is compatible with other slave devices on the same bus.

The library includes features such as SPI master and SPI slave modes, supported speeds of up to 100 Mbit, multiple slave device support and support for all configurations of clock polarity and phase.

1.2 Block diagram

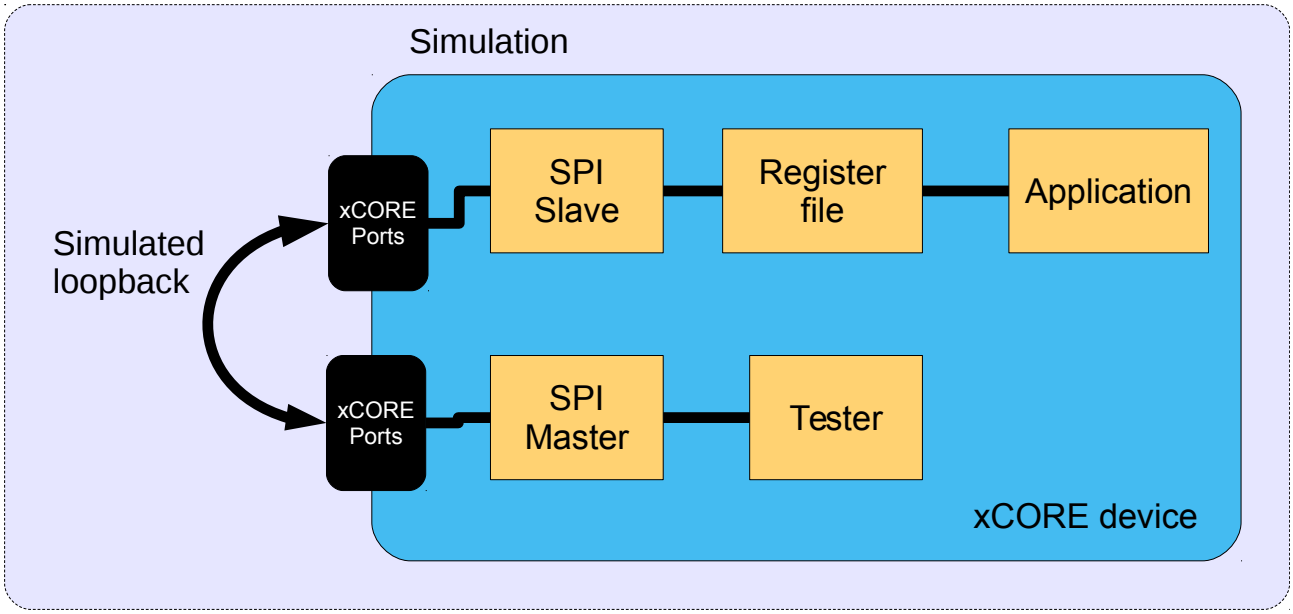


Figure 1: Block diagram of SPI slave application example

2 SPI slave example

The example in this application note uses the XMOS SPI library to act as SPI slave. It maintains a register file which can be read and written by the internal application *or* by the master on the SPI bus. To show the bus functioning the demo application also has a tester component connected to an SPI master bus which is connected (in simulation) to the the SPI slave. This allows the generation of traffic to show the the communication functioning.

The application consists of five tasks:

- A task that controls the SPI slave ports
- A task that implements the register file handling calls from the SPI slave component and the application
- An application task that connects to the register file task
- A task that controls the SPI master ports used for testing
- A tester task that outputs commands to the SPI master task

These tasks communicate via the use of xC interfaces.

The following diagram shows the task and communication structure of the application.

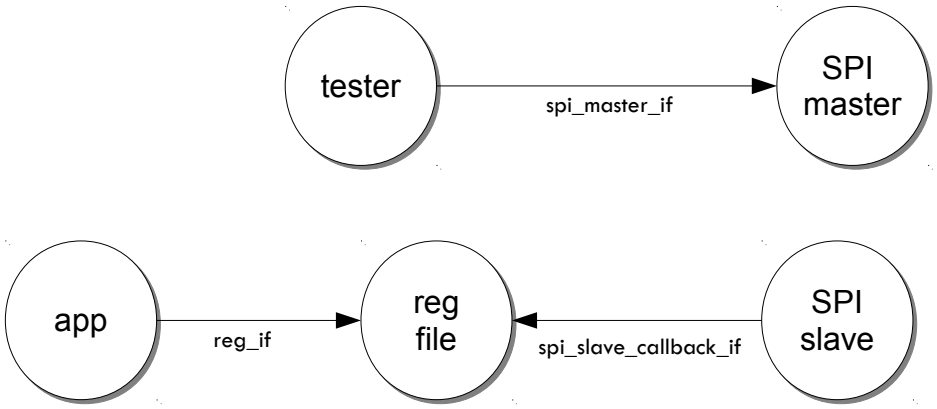


Figure 2: Task diagram of SPI slave example

The SPI slave task communicates via the `spi_slave_callback_if` which the SPI library defines as a set of callbacks a slave will make to the application to handle communication occuring on the SPI bus.

The application also needs to communicate with the register file task to get or set register values. To do this we define a custom communication software interface (`reg_if`).

2.1 Makefile additions for this example

To start using the SPI library, you need to add `lib_spi` to your `Makefile`:

```
USED_MODULES = ... lib_spi
```

You can then access the SPI functions in your source code via the spi.h header file:

```
#include <spi.h>
```

The example also uses the XMOS debug printing library (`lib_logging`). This provides handy debug printing functions:

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

2.2 Declaring ports

The SPI library connects to external pins via xCORE ports. In `main.xc` these are declared as variables of type `port` at the start of the file:

```
in port                 p_sclk = on tile[0]: XS1_PORT_1E;
in port                 p_ss   = on tile[0]: XS1_PORT_1F;
out buffered port:32    p_miso = on tile[0]: XS1_PORT_1G;
in buffered port:32     p_mosi = on tile[0]: XS1_PORT_1H;
clock                   cb     = on tile[0]: XS1_CLKBLK_1;
```

Note that there is also a clock declaration since the slave needs to use an internal clock as well as ports inside the xCORE device.

How the ports (e.g. XS1_PORT_1I) relate to external pins will depend on the exact device being used. See the device datasheet for details.

This application also has an SPI master interface on different ports:

```
out buffered port:32   p_test_sclk  = on tile[0]: XS1_PORT_1I;
out port               p_test_ss[1] = on tile[0]: {XS1_PORT_1J};
in buffered port:32    p_test_miso  = on tile[0]: XS1_PORT_1K;
out buffered port:32   p_test_mosi  = on tile[0]: XS1_PORT_1L;
```

2.3 The application main() function

Below is the source code for the main function of this application, which is taken from the source file main.xc

```
int main(void) {
  interface spi_slave_callback_if i_spi;
  interface reg_if i_reg;
  interface spi_master_if i_spi_master[1];
  par {
    on tile[0]: spi_slave(i_spi, p_sclk, p_mosi, p_miso, p_ss, cb, SPI_MODE_0,
                          SPI_TRANSFER_SIZE_8);
    on tile[0]: reg_file(i_spi, i_reg);
    on tile[0]: app(i_reg);

    // These tasks are not part of the application but a test harness to
    // provide SPI master data which is expected to be looped back in
    // simulation to the SPI slave ports.
    on tile[0]: tester(i_spi_master[0]);
    on tile[0]: spi_master(i_spi_master, 1,
                           p_test_sclk, p_test_mosi, p_test_miso, p_test_ss,
                           1, null);
  }
  return 0;
}
```

Looking at this in a more detail you can see the following:

- The par functionality describes running five separate tasks in parallel; three are for the main application and two are for the tester.
- The spi_slave task controls the application SPI bus and takes the ports it will use as arguments.
- The reg_file task is connected to the app task and the spi_slave task.
- The spi_slave task has an argument for the mode is expects - in this case Mode 0 (see the SPI library user guide for more details on modes)
- The spi_slave task also has an argument SPI_TRANSFER_SIZE_8 which specifies the size of data chunk it will use when making callbacks to the application.
- The spi_master task controls the test SPI bus and takes different ports to the SPI slave bus as arguments. For details on using SPI master see application note AN00160.

2.4 The reg_file() function

The reg_file function is the main logic of this example. It will respond to calls from the application and the SPI slave bus whilst maintaining a set of register values.

The function is marked as [[distributable]] which means it only responds to calls from other tasks. The main reason for this is so that the reg_file task itself does not need a logical core of its own it can use the logical core of the task that calls it. See the XMOS programming guide for details of distributable tasks.

The function takes two arguments, the interface connections to the application task and the SPI slave task:

```
[[distributable]]
void reg_file(server spi_slave_callback_if i_spi,
              server reg_if i_reg)
{
```

The reg_if interface has been defined in main.xc earlier. It defines the functions that the app may call in the reg_file tasks:

```
typedef interface reg_if {
  uint8_t get_reg(uint8_t regnum);
  void set_reg(uint8_t regnum, uint8_t value);
} reg_if;
```

In this case we have two functions - one for reading a register value and one for writing a register value.

The reg_file function first declares its state - an array to hold register value, a state variable to hold what stage of an SPI transaction it is in and the currently addressed register by the SPI bus.

```
/* This array holds the register values */
uint8_t reg_data[NUM_REG] = {0};

/* This variable holds the current state of the register file with respect
 * to the SPI bus (i.e. what stage of the transaction over SPI it is at).
 */
enum reg_state_t state = IDLE;

/* This variable holds the current register being addressed over the SPI
 * bus.
 */
uint8_t addr = 0;
```

The state variable is just an integer from the following enum type defined earlier in the file:

```
enum reg_state_t {
  WRITE_REG = 0,
  READ_REG = 1,
  WRITE_REG_DATA,
  READ_REG_DATA,
  IDLE
};
```

The implemented protocol on the SPI bus is as follows:

- The master will start a transaction (assert slave select)
- It will then send a byte of either a 0 for a write or a 1 for a read.
- It will then send the address of the register to read/write
- It will then send or receive the value of the register

To implement the protocol logic the `reg_file` tasks must continually react to events from the SPI slave tasks keeping track of its state, updating registers and supplying the correct outputs. This is done via a `while (1)` loop with an xC `select` statement inside it. A `select` statement will wait and then react to various events or calls from different tasks - see the XMOS programming guide for more details.

The following cases in the main loop of the function handle this:

```
while (1) {
    select {
    /* These cases react to the SPI slave bus. A write from the bus will
     * update the state of the transaction. A read from the bus will get
     * sent the data from the currently addressed register. */
    case i_spi.master_ends_transaction(void):
        state = IDLE;
        break;
    case i_spi.master_requires_data(void) -> uint32_t data:
        data = reg_data[addr];
        break;
    case i_spi.master_supplied_data(uint32_t datum, uint32_t valid_bits):
        switch (state) {
        case IDLE:
            if (datum == WRITE_REG || datum == READ_REG)
                state = datum;
            break;
        case READ_REG:
            if (datum < NUM_REG) {
                addr = datum;
                state = READ_REG_DATA;
            } else {
                state = IDLE;
            }
            break;
        case READ_REG_DATA:
            // Do nothing with master data during a read data operation.
            break;
        case WRITE_REG:
            if (datum < NUM_REG) {
                addr = datum;
                state = WRITE_REG_DATA;
            } else {
                state = IDLE;
            }
            break;
        case WRITE_REG_DATA:
            reg_data[addr] = datum;
            break;
        }
        break;
```

We can see that the slave will always send the value of the currently addressed register on every data transfer (this is allowable in the described protocol).

When the SPI master supplies some data to the slave then what happens depends on the current state - either the state variable is updated, the currently addressed register is updated or a register value is updated. This state machine will implement the previously described protocol.

The main select also needs to react to request from the application. The following cases implement this:

```
/* The following cases respond to the application when it
 * requests to get/set a register.
 */
case i_reg.get_reg(uint8_t regnum) -> uint8_t value:
    value = reg_data[regnum];
    break;
case i_reg.set_reg(uint8_t regnum, uint8_t value):
    reg_data[regnum] = value;
    break;
```

2.5 The app() function

The app task represents a sample application tasks that uses the register file. In this demo, it doesn't do much - it simple sets one register and repeatedly polls the value of another register and prints out its value:

```
void app(client reg_if reg) {
    reg.set_reg(0, 0xfe);
    debug_printf("APP: Set register 1 to 0xFE\n");
    while (1) {
        delay_microseconds(20);
        debug_printf("APP: Register 1 is 0x%x\n", reg.get_reg(1));
    }
}
```

Note that the debug_printf function comes from the debug_print.h header supplied by lib_logging. It is a low memory debug printing function that will print out messages to the console in the xTIMEcomposer (either using JTAG or xSCOPE to communicate to the host via the debug adaptor).

2.6 The tester() function

The tester function will send some test data to the SPI master bus. It does this using the SPI master interface to communicate with the SPI master task:

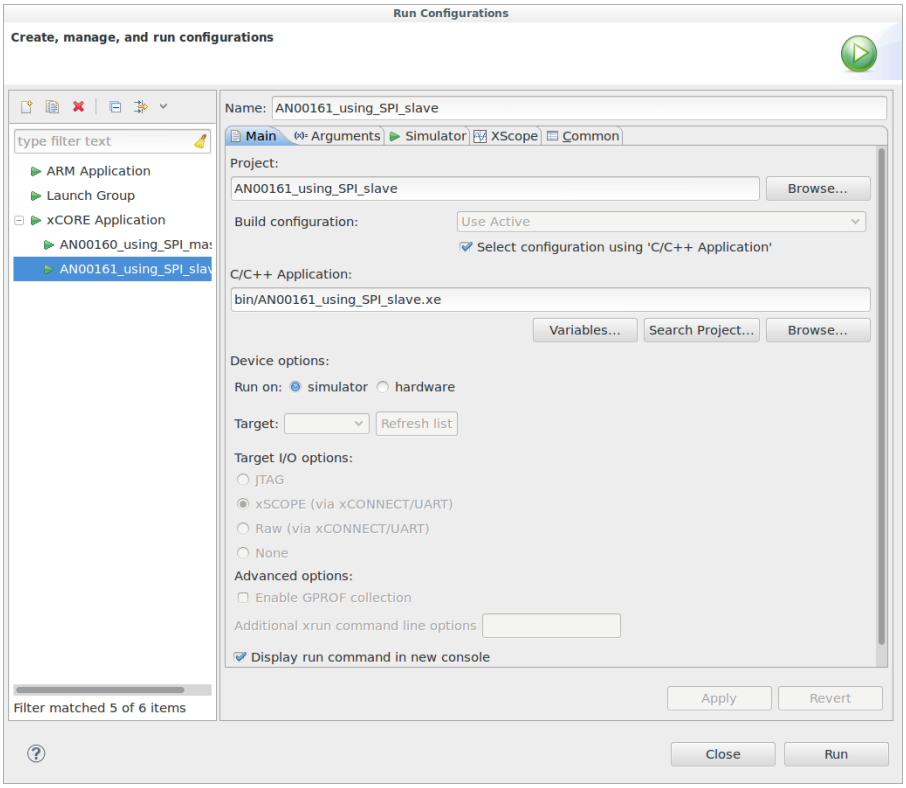
```
void tester(client spi_master_if spi)
{
    delay_microseconds(45);
    uint8_t val;
    spi.begin_transaction(0, 100, SPI_MODE_0);
    spi.transfer8(READ_REG); // READ command
    spi.transfer8(0); // REGISTER 0
    val = spi.transfer8(0); // DATA
    spi.end_transaction(100);
    debug_printf("SPI MASTER: Read register 0: 0x%x\n", val);

    spi.begin_transaction(0, 100, SPI_MODE_0);
    spi.transfer8(WRITE_REG); // WRITE command
    spi.transfer8(1); // REGISTER 1
    spi.transfer8(0xac); // DATA
    spi.end_transaction(100);
    debug_printf("SPI MASTER: Set register 1 to 0xAC\n");
}
```

2.7 Setting up the run configuration for the application

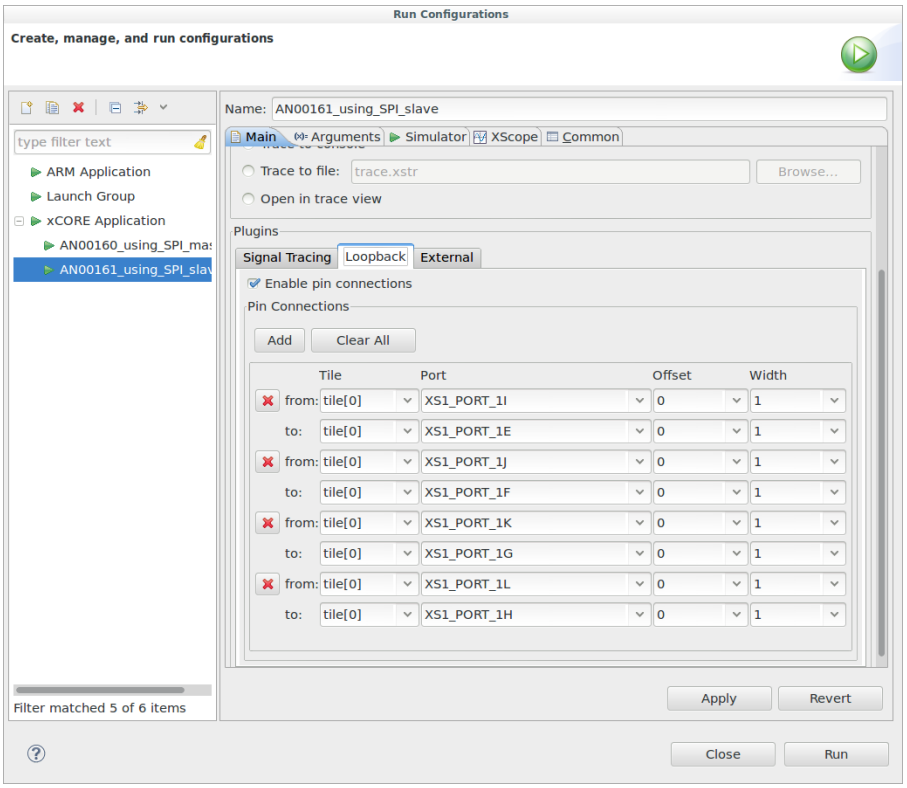
To run the application binary in the simulator, first the application must be built by pressing the **Build** button in the xTIMEcomposer. This will create the `AN00161_using_SPI_slave.xe` binary in the `bin` folder of the project. The xTIMEcomposer may have to import the SPI library if you do not already have it in your workspace; this will occur automatically on build.

Then a *Run Configuration* needs to be set up. This can be done by selecting the **Run ▶ Run Configurations..** menu. You can create a new run configuration by right clicking on the **xCORE application** group in the left hand pane and **new**. However, in this example a run configuration has already been created for you.

Looking at this run configuration, you can see the simulator has been selected under **Device Options:**:



In the **Loopback** tab of the **Simulator** tab of the run configuration. The **Enable pin connections** check box has been enabled and four loopback connections have been added connecting the SPI master ports to the corresponding SPI slave ports:



2.8 Running the application

By clicking on the **Run** icon (a green arrow) in the Edit Perspective of the xTIMEcomposer or by clicking the **Run** button in the run configuration dialog, the program will run. In the console window you will get this output:

```
APP: Set register 1 to 0xFE
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
SPI MASTER: Read register 0: 0xFE
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
APP: Register 1 is 0x0
SPI MASTER: Set register 1 to 0xAC
APP: Register 1 is 0xAC
APP: Register 1 is 0xAC
APP: Register 1 is 0xAC
...
```

You can terminate the program by clicking the **Terminate** button in the Console window (the red square).

The console shows the debug print output from the application and tester. You can see that the SPI bus can read registers that the application has set and the application can read registers that the SPI bus has set.

APPENDIX A - 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 SPI Library

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

APPENDIX B - Full source code listing

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <xs1.h>
#include <spi.h>
#include <stdint.h>
#include <timer.h>
#include <debug_print.h>
#include <platform.h>

/* These ports are used for the SPI slave task */
in port                 p_sclk = on tile[0]: XS1_PORT_1E;
in port                 p_ss   = on tile[0]: XS1_PORT_1F;
out buffered port:32    p_miso = on tile[0]: XS1_PORT_1G;
in buffered port:32     p_mosi = on tile[0]: XS1_PORT_1H;
clock                   cb     = on tile[0]: XS1_CLKBLK_1;

/* These ports are used for the SPI master task which is
   used to test the SPI slave (via simulator loopback). */
out buffered port:32   p_test_sclk  = on tile[0]: XS1_PORT_1I;
out port               p_test_ss[1] = on tile[0]: {XS1_PORT_1J};
in buffered port:32    p_test_miso  = on tile[0]: XS1_PORT_1K;
out buffered port:32   p_test_mosi  = on tile[0]: XS1_PORT_1L;

/* Interface to communicate between the register file tasks
   and the application. */
typedef interface reg_if {
  uint8_t get_reg(uint8_t regnum);
  void set_reg(uint8_t regnum, uint8_t value);
} reg_if;

#define NUM_REG 5

enum reg_state_t {
  WRITE_REG = 0,
  READ_REG = 1,
  WRITE_REG_DATA,
  READ_REG_DATA,
  IDLE
};

/* This application implements a register file which can be accessed by
 * the application on chip as well as over the SPI bus.
 *
 * To do this, the reg_file task reacts to both the SPI slave component or
 * requests from the application.
 *
 * Over SPI, the master is expected to assert the slave select line and
 * then send either a 0 for writing a register or a 1 for reading a register in
 * the first byte. The next byte is expected to contain the register number
 * to read/write. The final byte either receives or transmits the register
 * value.
 */
[[distributable]]
void reg_file(server spi_slave_callback_if i_spi,
              server reg_if i_reg)
{
  /* This array holds the register values */
```

```
uint8_t reg_data[NUM_REG] = {0};

/* This variable holds the current state of the register file with respect
 * to the SPI bus (i.e. what stage of the transaction over SPI it is at).
 */
enum reg_state_t state = IDLE;

/* This variable holds the current register being addressed over the SPI
 * bus.
 */
uint8_t addr = 0;

while (1) {
    select {
    /* These cases react to the SPI slave bus. A write from the bus will
     * update the state of the transaction. A read from the bus will get
     * sent the data from the currently addressed register. */
    case i_spi.master_ends_transaction(void):
        state = IDLE;
        break;
    case i_spi.master_requires_data(void) -> uint32_t data:
        data = reg_data[addr];
        break;
    case i_spi.master_supplied_data(uint32_t datum, uint32_t valid_bits):
        switch (state) {
        case IDLE:
            if (datum == WRITE_REG || datum == READ_REG)
                state = datum;
            break;
        case READ_REG:
            if (datum < NUM_REG) {
                addr = datum;
                state = READ_REG_DATA;
            } else {
                state = IDLE;
            }
            break;
        case READ_REG_DATA:
            // Do nothing with master data during a read data operation.
            break;
        case WRITE_REG:
            if (datum < NUM_REG) {
                addr = datum;
                state = WRITE_REG_DATA;
            } else {
                state = IDLE;
            }
            break;
        case WRITE_REG_DATA:
            reg_data[addr] = datum;
            break;
        }
        break;

    /* The following cases respond to the application when it
     * requests to get/set a register.
     */
    case i_reg.get_reg(uint8_t regnum) -> uint8_t value:
```

```
          value = reg_data[regnum];
          break;
      case i_reg.set_reg(uint8_t regnum, uint8_t value):
          reg_data[regnum] = value;
          break;
      }
  }
}

/* This dummy application just sets a register value and then continually
 * polls and prints out another register value. It uses an interface connection
 * to the reg_file tasks to get/set registers.
 */
void app(client reg_if reg) {
    reg.set_reg(0, 0xfe);
    debug_printf("APP: Set register 1 to 0xFE\n");
    while (1) {
        delay_microseconds(20);
        debug_printf("APP: Register 1 is 0x%x\n", reg.get_reg(1));
    }
}

/* The tester task is not part of the application but just sends commands
 * over an SPI master bus on the same chip. The simulation can then loopback
 * the SPI ports onto the SPI slave bus to provide test traffic. This loopback
 * needs to be set up in the run configuration of the application.
 */
void tester(client spi_master_if spi)
{
    delay_microseconds(45);
    uint8_t val;
    spi.begin_transaction(0, 100, SPI_MODE_0);
    spi.transfer8(READ_REG); // READ command
    spi.transfer8(0); // REGISTER 0
    val = spi.transfer8(0); // DATA
    spi.end_transaction(100);
    debug_printf("SPI MASTER: Read register 0: 0x%x\n", val);

    spi.begin_transaction(0, 100, SPI_MODE_0);
    spi.transfer8(WRITE_REG); // WRITE command
    spi.transfer8(1); // REGISTER 1
    spi.transfer8(0xac); // DATA
    spi.end_transaction(100);
    debug_printf("SPI MASTER: Set register 1 to 0xAC\n");
}

int main(void) {
  interface spi_slave_callback_if i_spi;
  interface reg_if i_reg;
  interface spi_master_if i_spi_master[1];
  par {
    on tile[0]: spi_slave(i_spi, p_sclk, p_mosi, p_miso, p_ss, cb, SPI_MODE_0,
                          SPI_TRANSFER_SIZE_8);
    on tile[0]: reg_file(i_spi, i_reg);
    on tile[0]: app(i_reg);

    // These tasks are not part of the application but a test harness to
    // provide SPI master data which is expected to be looped back in
```

```
    // simulation to the SPI slave ports.
    on tile[0]: tester(i_spi_master[0]);
    on tile[0]: spi_master(i_spi_master, 1,
                           p_test_sclk, p_test_mosi, p_test_miso, p_test_ss,
                           1, null);
  }
  return 0;
}
```