Application Note: AN00160

How to use the SPI library as SPI master

This application note shows how to use the SPI library to make the xCORE drive an SPI bus as SPI master. The application is the simplest example of setting up the library and performing a couple of transactions. The code can then be run in simulation to see the outputted waveforms.

The note covers both the synchronous and asynchronous use of the SPI master components provided from the library.

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

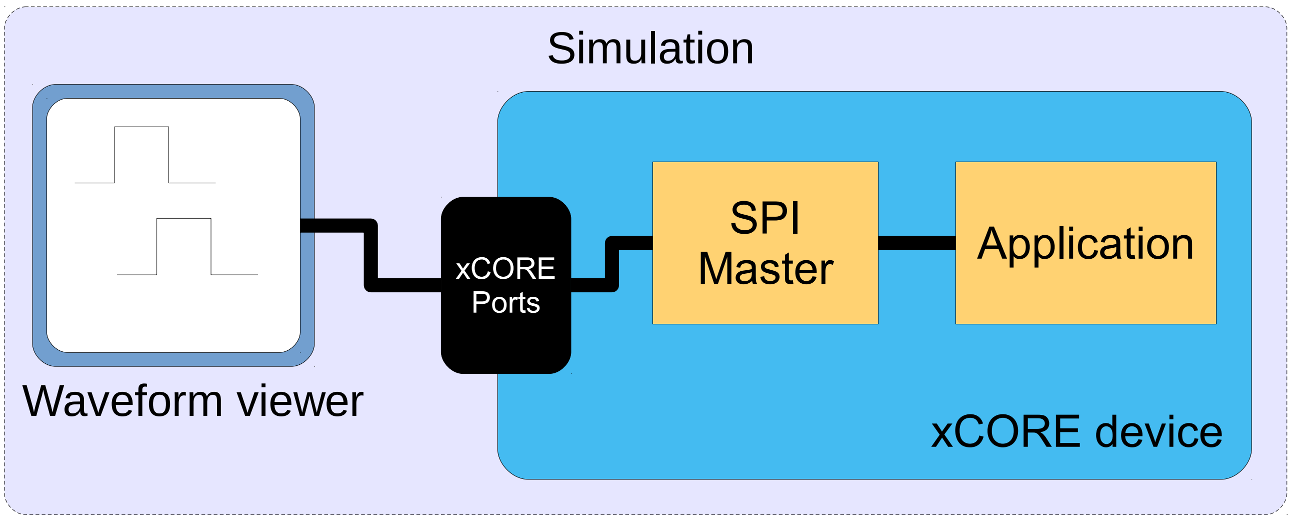


Figure 1: Block diagram of SPI master application example

2 SPI master example

The example in this application note uses the XMOS SPI library to perform some bus transactions as SPI master. The binary is then run in simulation so the user can see the waveform output in the VCD tracing perspective in the xTIMEcomposer.

The application consists of two tasks:

- A task that drives the SPI bus
- An application task that connects to the SPI task

These tasks communicate via the use of xC interfaces.

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

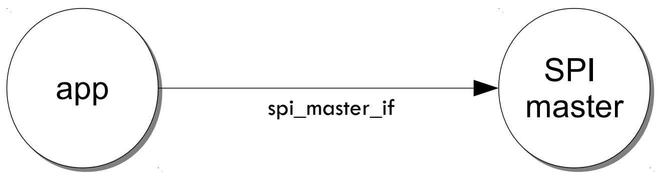


Figure 2: Task diagram of SPI master example

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>
```

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:

```
out buffered port:32   p_sclk  = on tile[0]: XS1_PORT_1I;
out port               p_ss[1] = on tile[0]: {XS1_PORT_1J};
in buffered port:32    p_miso  = on tile[0]: XS1_PORT_1K;
out buffered port:32   p_mosi  = on tile[0]: XS1_PORT_1L;
```

Note that the slave select declaration (p_ss) is an array of ports. This is what the SPI library expects since there may be many slave select lines for multiple devices.

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.

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_master_if i_spi[1];
  par {
    on tile[0]: app(i_spi[0]);
    on tile[0]: spi_master(i_spi, 1,
                           p_sclk, p_mosi, p_miso, p_ss, 1,
                           null);
  }
  return 0;
}
```

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

- The par functionality describes running two separate tasks in parallel
- The `spi_master` task drives the SPI bus and takes the ports it will use as arguments.
- The app task communicates to the `spi_master` task via the shared interface argument `i_spi`. This is an array since the SPI master task could connect to many other tasks in parallel.

2.4 The app() function

The app function uses its interface connection to the SPI master task to perform SPI transactions. It performs two transactions (each transaction will assert the slave select line, transfer some data and then de-assert the slave select line). The functions in the SPI master interface can be found in the SPI library user guide.

```
void app(client spi_master_if spi)
{
    uint8_t val;
    printstrln("Sending SPI traffic");
    delay_microseconds(30);
    spi.begin_transaction(0, 100, SPI_MODE_0);
    val = spi.transfer8(0xab);
    val = spi.transfer32(0xcc);
    val = spi.transfer8(0xfe);
    spi.end_transaction(100);

    delay_microseconds(40);
    spi.begin_transaction(0, 100, SPI_MODE_0);
    val = spi.transfer8(0x22);
    spi.end_transaction(100);

    printstrln("Done.");
    _exit(0);
}
```

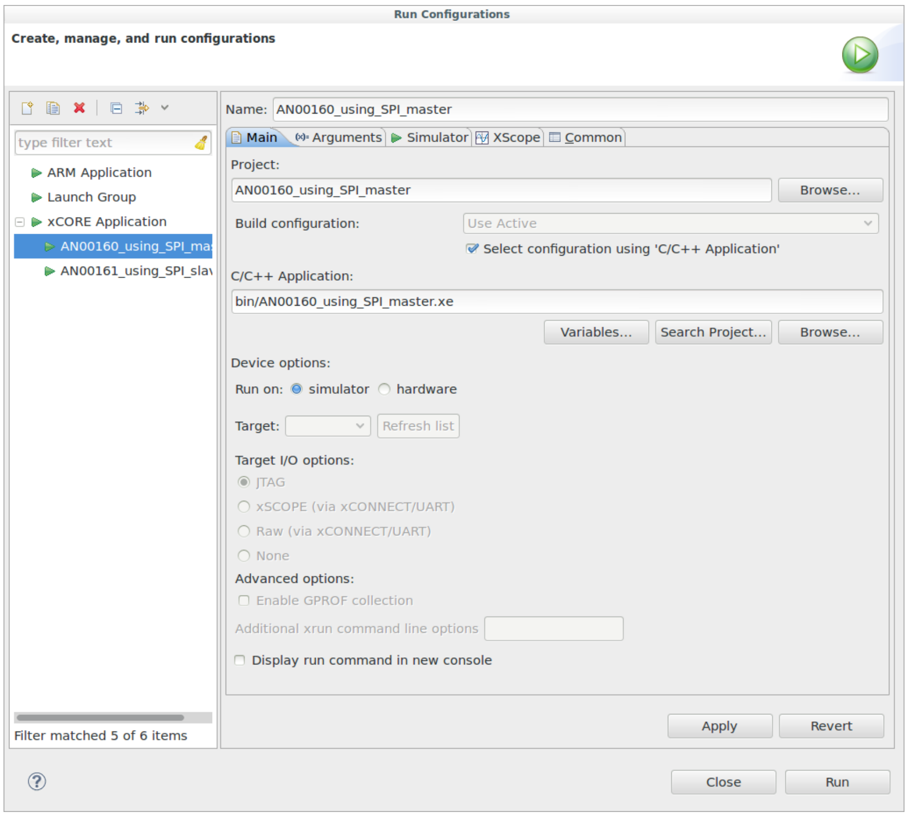
Note that when `begin_transaction` is called the device is selected by the first argument. In this case it is 0, so the zero-th element of the p_ss array will be used for the slave select line. This is the method that is used to communiate with multiple SPI slave devices. The speed and mode of the SPI protocol is also set at in the `begin_transaction` call.

2.5 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 `AN00160_using_SPI_master.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:**:



Name: AN00160_using_SPI_master

Project: AN00160_using_SPI_master

Build configuration: Use Active

[✓] Select configuration using 'C/C++ Application'

C/C++ Application: bin/AN00160_using_SPI_master.xe

Device options:

Run on: [✓ simulator] hardware

Target:

Target I/O options:

[✓ JTAG]
xSCOPE (via xCONNECT/UART)
Raw (via xCONNECT/UART)
None

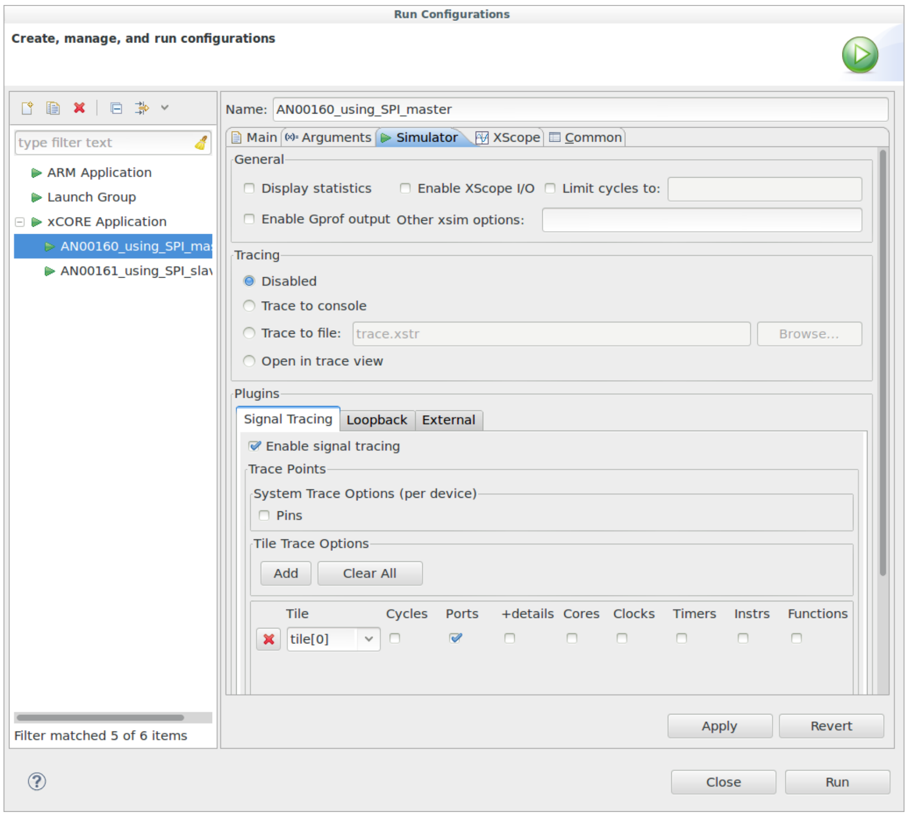
Advanced options:

[ ] Enable GPROF collection

Additional xrun command line options

[ ] Display run command in new console

In the **Simulator** tab of the run configuration. The **Enable signal tracing** check box has been enabled and a **Tile Trace Option** has been added to trace the ports on tile[0]:



Run Configurations

Create, manage, and run configurations

type filter text

- ARM Application
- Launch Group
- xCORE Application
  - AN00160_using_SPI_ma
  - AN00161_using_SPI_slav

Filter matched 5 of 6 items

Name: AN00160_using_SPI_master

Main | Arguments | Simulator | XScope | Common

General
- ☐ Display statistics
- ☐ Enable XScope I/O
- ☐ Limit cycles to:
- ☐ Enable Gprof output
- Other xsim options:

Tracing
- ● Disabled
- ○ Trace to console
- ○ Trace to file: trace.xstr Browse...
- ○ Open in trace view

Plugins

Signal Tracing | Loopback | External

- ✓ Enable signal tracing

Trace Points

System Trace Options (per device)
- ☐ Pins

Tile Trace Options

Add | Clear All

Tile	Cycles	Ports	+details	Cores	Clocks	Timers	Instrs	Functions
tile[0]	☐	✓	☐	☐	☐	☐	☐	☐

Apply | Revert

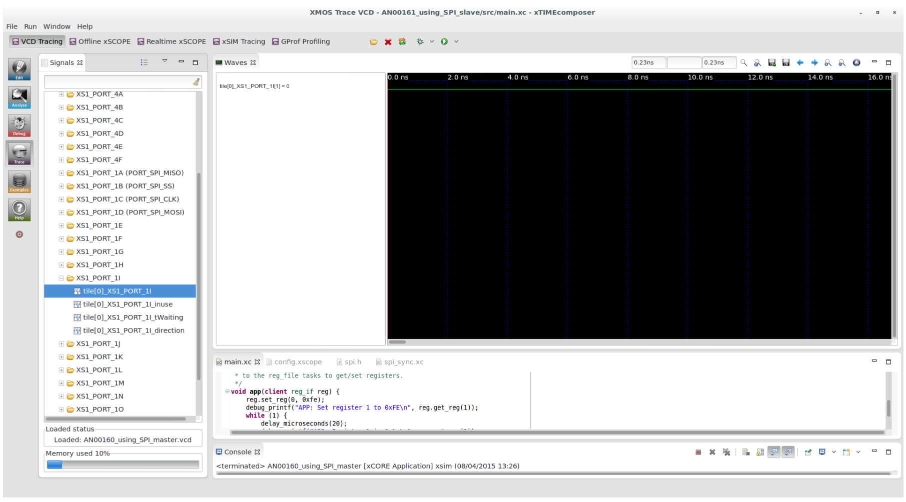
Close | Run

2.6 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:

```
Sending SPI traffic
Done.
```

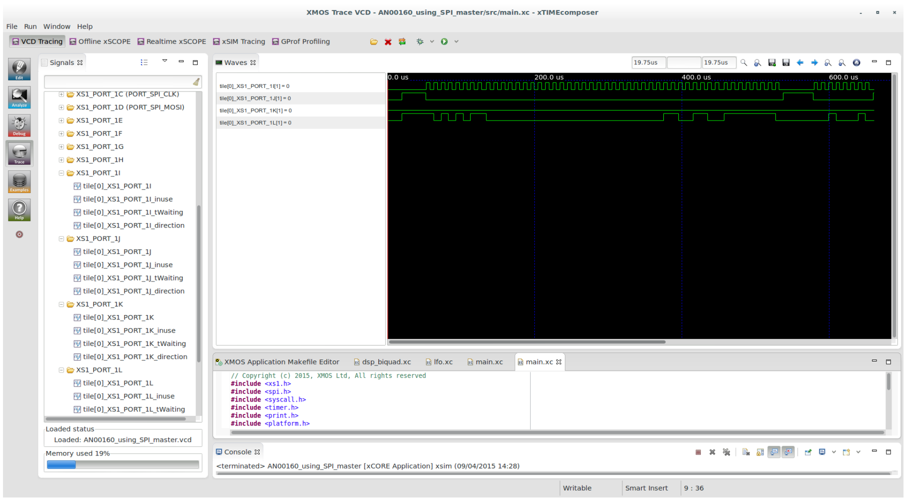
After this it will immediately swith the the VCD tracing perspective (since signal tracing was enabled in the run configuration). In this perspective you can drag ports from within the tree in the **Signals** pane on the left hand side to the **Waves** pane on the right:


XMOS Trace VCD - AN00161_using_SPI_slave/src/main.xc - xTIMEcomposer
Signals
Waves
tile[0]_XS1_PORT_1I
tile[0]_XS1_PORT_1I_inuse
tile[0]_XS1_PORT_1I_tWaiting
tile[0]_XS1_PORT_1I_direction
Loaded: AN00160_using_SPI_master.vcd
Memory used 10%
main.xc
config.xscope
spi.h
spi_sync.xc
Console

```
 * 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", reg.get_reg(1));
    while (1) {
        delay_microseconds(20);
```

```
<terminated> AN00160_using_SPI_master [xCORE Application] xsim (08/04/2015 13:26)
```

By dragging in the four ports used by the application (1I, 1J, 1K and 1L) you can see that the application has driven the correct SPI signal. Note that you need to expand each port and drag in just the port value, the other traces (such as `tile[0]_XS1_PORT_1J_inuse`) just show the internal port state and are not so interesting here.



```
// Copyright (c) 2015, XMOS Ltd, All rights reserved
#include <xs1.h>
#include <spi.h>
#include <syscall.h>
#include <timer.h>
#include <print.h>
#include <platform.h>
```

```
<terminated> AN00160_using_SPI_master [xCORE Application] xsim (09/04/2015 14:28)
```

2.7 Using the asynchronous interface

There is an alternative `main()` function in the program to try:

```
/* Uncomment the main below (and comment out the one above) to try the
   async version.

int main(void) {
  interface spi_master_async_if i_spi_async[1];
  par {
    on tile[0]: async_app(i_spi_async[0]);
    on tile[0]: spi_master_async(i_spi_async, 1,
                              p_sclk, p_mosi, p_miso, p_ss, 1,
                              clk0, clk1);
  }
  return 0;
}

*/
```

By uncommenting this main (and commenting out the original main) the application will use the `spi_master_async` task instead of the synchronous `spi_master` task. This task still drives the SPI bus but runs on a separate logical core and will drive the bus in parallel to your application. This way your application can overlap processing with communication.

The async_app() function performs the same function as the app() function using the asynchronous interface. The big difference is that pointers have to be passed to the SPI task that point to the buffers to send from/receive into. These need to be *movable* pointers (see the xC programing guide for more information) that can be passed to another task:

```
void async_app(client spi_master_async_if spi)
{
    uint8_t indata[10];
    uint8_t outdata[10];
    uint8_t * movable inbuf = indata;
    uint8_t * movable outbuf = outdata;
```

Once the pointers have been initialized they can be passed and later retrieved from the SPI master task:

```
printstrln("Sending SPI traffic (async)");
delay_microseconds(30);

// Fill the out buffer with data
outbuf[0] = 0xab;
outbuf[1] = 0xcc;
outbuf[2] = 0; outbuf[3] = 0; outbuf[4] = 0;
outbuf[5] = 0xfe;
spi.begin_transaction(0, 100, SPI_MODE_0);

// This call passes the buffers over to the SPI task, after
// this the application cannot access the buffers until
// the retrieve_transfer_buffers_8 function is called.
spi.init_transfer_array_8(move(inbuf),
                          move(outbuf),
                          6);

// Your application can do calculation here whilst the spi task
// transfers the buffer.

// A select will wait for an event. In this case the event we are waiting
// for is the transfer_complete() notification event from the SPI task.
select {
    case spi.transfer_complete():
       // Once the transfer is complete, we can retrieve the
       // buffers back into the inbuf and outbuf pointer variables
       spi.retrieve_transfer_buffers_8(inbuf, outbuf);
       break;
}

spi.end_transaction(100);
```

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 <syscall.h>
#include <timer.h>
#include <print.h>
#include <platform.h>

/* These ports are used for the SPI master */
out buffered port:32   p_sclk  = on tile[0]: XS1_PORT_1I;
out port               p_ss[1] = on tile[0]: {XS1_PORT_1J};
in buffered port:32    p_miso  = on tile[0]: XS1_PORT_1K;
out buffered port:32   p_mosi  = on tile[0]: XS1_PORT_1L;

clock clk0 = on tile[0]: XS1_CLKBLK_1;
clock clk1 = on tile[0]: XS1_CLKBLK_2;

/* This application function sends some traffic as SPI master using
 * the synchronous interface. Since this is run in simulation
 * there is no slave, so the incoming data (stored in the 'val'
 * variable) will just be zero.
 */
void app(client spi_master_if spi)
{
    uint8_t val;
    printstrln("Sending SPI traffic");
    delay_microseconds(30);
    spi.begin_transaction(0, 100, SPI_MODE_0);
    val = spi.transfer8(0xab);
    val = spi.transfer32(0xcc);
    val = spi.transfer8(0xfe);
    spi.end_transaction(100);

    delay_microseconds(40);
    spi.begin_transaction(0, 100, SPI_MODE_0);
    val = spi.transfer8(0x22);
    spi.end_transaction(100);

    printstrln("Done.");
    _exit(0);
}

/* This application function sends some traffic as SPI master using
 * the asynchronous interface. Since this is run in simulation
 * there is no slave, so the incoming data (stored in the 'val'
 * variable) will just be zero.
 */
void async_app(client spi_master_async_if spi)
{
    uint8_t indata[10];
    uint8_t outdata[10];
    uint8_t * movable inbuf = indata;
    uint8_t * movable outbuf = outdata;

    printstrln("Sending SPI traffic (async)");
    delay_microseconds(30);
    // Fill the out buffer with data
    outbuf[0] = 0xab;
    outbuf[1] = 0xcc;
    outbuf[2] = 0; outbuf[3] = 0; outbuf[4] = 0;
    outbuf[5] = 0xfe;
    spi.begin_transaction(0, 100, SPI_MODE_0);

    // This call passes the buffers over to the SPI task, after
    // this the application cannot access the buffers until
    // the retrieve_transfer_buffers_8 function is called.
    spi.init_transfer_array_8(move(inbuf),
                              move(outbuf),
                              6);

    // Your application can do calculation here whilst the spi task
    // transfers the buffer.

    // A select will wait for an event. In this case the event we are waiting
    // for is the transfer_complete() notification event from the SPI task.
    select {
        case spi.transfer_complete():
           // Once the transfer is complete, we can retrieve the
           // buffers back into the inbuf and outbuf pointer variables
           spi.retrieve_transfer_buffers_8(inbuf, outbuf);
           break;
    }

    spi.end_transaction(100);


    delay_microseconds(40);
    spi.begin_transaction(0, 100, SPI_MODE_0);
    outbuf[0] = 0x22;
    spi.init_transfer_array_8(move(inbuf),
                              move(outbuf),
                              1);
    select {
        case spi.transfer_complete():
           spi.retrieve_transfer_buffers_8(inbuf, outbuf);
           break;
    }
    spi.end_transaction(100);

    printstrln("Done.");
    _exit(0);
}

int main(void) {
  interface spi_master_if i_spi[1];
  par {
    on tile[0]: app(i_spi[0]);
    on tile[0]: spi_master(i_spi, 1,
                           p_sclk, p_mosi, p_miso, p_ss, 1,
                           null);
  }
  return 0;
}
```

```
/* Uncomment the main below (and comment out the one above) to try the
   async version.

int main(void) {
  interface spi_master_async_if i_spi_async[1];
  par {
    on tile[0]: async_app(i_spi_async[0]);
    on tile[0]: spi_master_async(i_spi_async, 1,
                                 p_sclk, p_mosi, p_miso, p_ss, 1,
                                 clk0, clk1);
  }
  return 0;
}

*/
```