Application Note: AN00188

Using QuadSPI flash memory for persistent storage with xCORE-200

This application note demonstrates how to use XFLASH option `--data` to store persistent data within QuadSPI flash memory.

This application note provides an example that uses the boot partition in QuadSPI flash memory to store the application and the data partition in QuadSPI flash memory to store persistent application data. Once booted the application reads data from the data partition using the xCORE quadflash library and uses it to illuminate the LED's in various patterns.

Required tools and libraries

- xTIMEcomposer Tools Suite version 14.0 or later is required.

Required hardware

This application note is designed to run on an XMOS xCORE-200 series device.

The example code provided with the application has been implemented and tested on the xCORE-200 explorerKIT core module board but there is no dependancy on this board and it can be modified to run on any development board which uses an xCORE-200 series device.

Prerequisites

- This document assumes familiarity with the XMOS xCORE-200 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.
- This document assumes familiarity with QuadSPI flash memory, the xCORE quadflash library and the XMOS tool XFLASH.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- The XMOS tools manual contains information regarding the use of xCORE devices[2].

[1] http://www.xmos.com/published/glossary
[2] http://www.xmos.com/published/xtimecomposer-user-guide

1 Introduction

xCORE-200 explorerKIT contains everything you need to start developing applications on the powerful xCORE-200 multicore microcontroller products from XMOS. It's easy to use and provides lots of advanced features on a small, low cost platform.

The xCORE-200 explorerKIT features our XE216-512 xCORE-200 multicore microcontroller. This device has sixteen 32bit logical cores that deliver up to 2000MIPS completely deterministically. The combination of 100/1000 Mbps Ethernet, high speed USB and 53 high performance GPIO make the xCORE-200 explorerKIT an ideal platform for functions ranging from robotics and motion control to networking and digital audio.

1.1 Block diagram

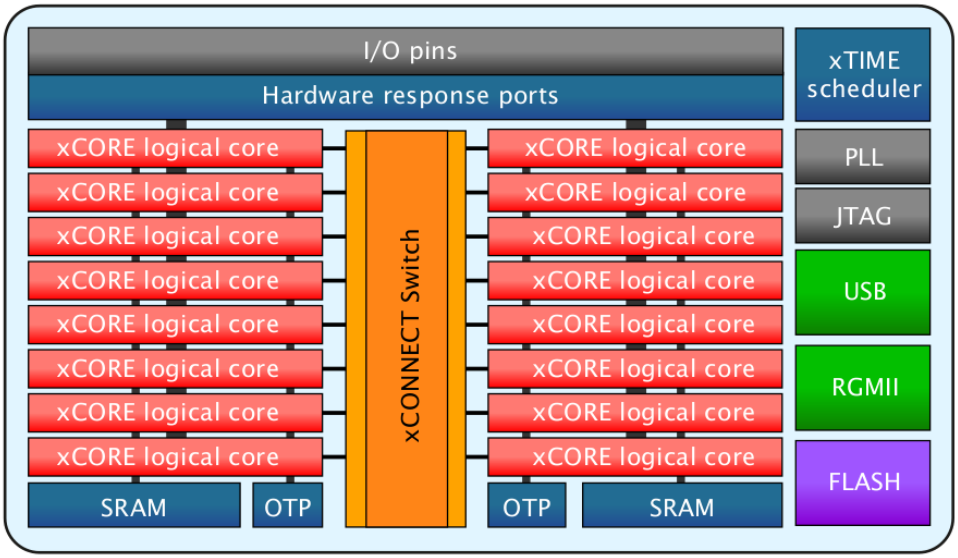

Figure 1: Block diagram of XE216-512 device on xCORE-200 explorerKIT

1.2 QuadSPI flash memory

The xCORE-200 explorerKIT features a QuadSPI flash device providing a 4-bit multiplexed I/O serial interface to boost performance while maintaining the compact form factor of standard serial flash devices. This allows the xCORE-200 device to load data from the data partition of flash memory significantly faster than with traditional SPI flash memory devices.

The QuadSPI flash memory is logically split between a boot and data partition.

The boot partition consists of a flash loader followed by a factory image and zero or more optional upgrade images. Each image starts with a descriptor that contains a unique version number and a header that contains a table of code/data segments for each tile used by the program and a CRC.

The data partition is empty by default and can be used to store application data using the XFLASH option `--data`.

An application designer may wish to use the data partition when the application data is too large to store within application memory and/or when the application data does not change.

In this application note example, a series of bytes is stored within the data partition that drive a different sequence of patterns on the explorerKIT's LED's. An application will be booted from the boot partition and will use the XMOS xCORE quadflash library to read the data from the data partition.

This application note demonstrates

- How to use the XFLASH option `--data` to store persistent application data in the data partition of Quad SPI flash memory.
- How to use the XFLASH option `--boot-partition-size` to partition QuadSPI flash memory into a boot partition and data partition.
- How to use the XMOS xCORE quadflash library to read the data from the data partition.

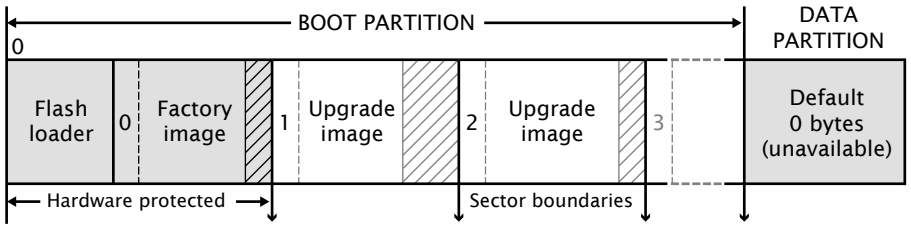

Figure 2: Flash format diagram

2 Using QuadSPI flash memory for persistent storage application note

2.1 Source code structure for this application note

This application note example is stored in the following directory structure:

```
src                                  <-- top level project source directory
  data.bin                           <-- binary file containing LED patterns
  qspi_persistent_storage.xc         <-- source file for project
Makefile                             <-- make file for project
```

2.2 Makefile support for using the xCORE quadflash library

It is a requirement to tell the xCORE toolchain that the application uses the xCORE quadflash library. This is achieved by passing the linker option `-lquadflash` with the associated build flags.

Options passed to xCORE build tools from makefile:

```
XCC_FLAGS = -g -report -lquadflash
```

2.3 The data.bin binary file

The data.bin binary file contains 12 bytes of data that will be written to the data partition. The data represents the LED patterns that can be displayed on the explorerKIT xCORE module board. When written to flash memory using the XFLASH option `--data` the sequence of bytes will be written exactly as is within the data.bin binary file.

The sequence of bytes stored within the data.bin file are:

```
0x01 0x02 0x04 0x08 0x03 0x05 0x09 0x06 0x0A 0x0C 0x0F 0x00
```

2.4 The application main() function

The led_driver() function for this example is contained within the file `qspi_persistent_storage.xc` and is as follows,

```
int led_driver (void)
{
  char led_byte_buffer[NUM_LED_BYTES];  // Buffer to hold the 12 bytes read from the data partition.
  int delay = 50;                       // Initial delay 50 ms.
  int led_counter = 0;                  // A counter to count through the led patterns.

  // Connect to the QuadSPI device using the quadflash library function fl_connectToDevice.
  if(fl_connectToDevice(ports, deviceSpecs, sizeof(deviceSpecs)/sizeof(fl_QuadDeviceSpec)) != 0)
  {
    return 1;
  }

  // Read all 12 bytes from offset 0 in the data partition and store them in
  // the buffer led_byte_buffer using the flash library function fl_readData.
  if(fl_readData(0, NUM_LED_BYTES, led_byte_buffer) != 0)
  {
    return 1;
  }

  // Disconnect from the QuadSPI device.
  fl_disconnect();

  while(1)
  {
    delay_milliseconds(delay);                          // Wait.
    delay += 1;                                         // Increase the delay so the LED pattern gets slower.
    led_port <: led_byte_buffer[led_counter];           // Drive the next led pattern.
    led_counter++;                                      // Pick the next pattern.
    if (led_counter == NUM_LED_BYTES)                   // If we are at the last pattern,
    {
      led_counter = 0;                                  // then wrap around.
    }
  }

  return 0;
}
```

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

- A quadflash library function is called to connect to the flash memory device
- A quadflash library function is called to read data from the data partiton
- A quadflash library function is called to disconnect from the flash memory device
- An infinite while loop is entered to continually rotate the pattern displayed on the LED's

2.5 Using the xCORE quadflash library

In order to use the quadflash library on the xCORE, the header file `quadflashlib.h` must be included.

```
#include <quadflashlib.h>
```

2.6 Using the xCORE quadflash library to connect to the flash device

The quadflash library needs to connect with the flash device and provides the function `fl_connectToDevice(fl_QSPIPorts &SPI, const fl_QuadDeviceSpec spec[], unsigned n)` to achieve this. When this function is called the quadflash library connects with the flash memory device, validates some configuration and determines the layout of the flash memory. The quadflash library therefore knows where the boot partition ends and where the data partition begins.

```
// Connect to the QuadSPI device using the quadflash library function fl_connectToDevice.
if(fl_connectToDevice(ports, deviceSpecs, sizeof(deviceSpecs)/sizeof(fl_QuadDeviceSpec)) != 0)
{
  return 1;
}
```

The parameter type fl_QSPIPorts is a structure defined in the quadflash library and contains the ports and clocks required to access the flash device. In this example the structure is defined and initialised with the ports convieniently labelled in the explorerKIT XN file as the variable ports. This example also uses XS1_CLKBLK_1 to drive the clock for flash device access.

```
// Ports for QuadSPI access on explorerKIT.
fl_QSPIPorts ports = {
  PORT_SQI_CS,
  PORT_SQI_SCLK,
  PORT_SQI_SIO,
  on tile[0]: XS1_CLKBLK_1
};
```

The parameter type fl_QuadDeviceSpec is a structure defined in the quadflash library and contains the properties of the flash device. In this example the structure is defined and initialised as an array of flash device properties that are currently supported and enabled by XMOS as the variable deviceSpecs.

```
// List of QuadSPI devices that are supported by default.
fl_QuadDeviceSpec deviceSpecs[] =
{
  FL_QUADDEVICE_SPANSION_S25FL116K,
  FL_QUADDEVICE_SPANSION_S25FL132K,
  FL_QUADDEVICE_SPANSION_S25FL164K,
  FL_QUADDEVICE_ISSI_IS25LQ080B,
  FL_QUADDEVICE_ISSI_IS25LQ016B,
  FL_QUADDEVICE_ISSI_IS25LQ032B,
};
```

The parameter n is used to specify the length of the fl_QuadDeviceSpec array.

If the quadflash library successfully connects to the flash device, the function fl_connectToDevice will return 0.

2.7 Using the xCORE quadflash library to read from the data partition

To read from the data partition of the flash device, the xCORE quadflash library provides the function fl_readData(unsigned offset, unsigned size, unsigned char dst[]).

```
// Read all 12 bytes from offset 0 in the data partition and store them in
// the buffer led_byte_buffer using the flash library function fl_readData.
if(fl_readData(0, NUM_LED_BYTES, led_byte_buffer) != 0)
{
  return 1;
}
```

The quadflash library knows the address of where the data partition begins. The parameter offset represents the offset from the start of the data partition in bytes of the data that is to be read. In this example the data is at the start of the data partition so the value 0 is given.

The parameter size is the number of bytes that is to be read from the data partition. In this example the number of bytes making up the LED patterns is 12 and is defined by NUM_LED_BYTES.

```
// There are 12 bytes stored in the data partition.
#define NUM_LED_BYTES 12
```

The parameter dst[] is an array of type unsigned char to hold the bytes that are read from the data partition. In this example the buffer that holds the data read from the data partition is defined by the

variable led_byte_buffer.

```
char led_byte_buffer[NUM_LED_BYTES];  // Buffer to hold the 12 bytes read from the data partition.
```

If the quadflash library successfully reads from the data partition, the function fl_readData will return 0.

2.8 Finishing with the xCORE quadflash library

The quadflash library provides the function fl_disconnect() that is used to close the connection to the QuadSPI flash device. When this function is called it releases the ports and clock resources used by the quadflash library back to the application. There will be no further access to the flash device once this function has been called. In this example the function fl_disconnect is called after the function fl_readData successfully returns.

```
// Disconnect from the QuadSPI device.
fl_disconnect();
```

2.9 Driving the LED patterns with the data read from the data partition

Once the data has been read from the data partition, the main purpose of the application begins. In this example the data read from the data partition is used to drive a sequence of patterns to the LED's on the explorerKIT core module board with the pattern change reducing in speed.

```
while(1)
{
  delay_milliseconds(delay);                          // Wait.
  delay += 1;                                         // Increase the delay so the LED pattern gets slower.
  led_port <: led_byte_buffer[led_counter];           // Drive the next led pattern.
  led_counter++;                                      // Pick the next pattern.
  if (led_counter == NUM_LED_BYTES)                   // If we are at the last pattern,
  {
    led_counter = 0;                                  // then wrap around.
  }
}
```

Each byte stored within the led_byte_buffer is sequntially driven out the led port led_port changing the combinations of LED's that are illuminated.

```
// Port where the leds reside on explorerKIT.
on tile[0] : port led_port = XS1_PORT_4F;
```

APPENDIX A - Example Hardware Setup

This application example is designed to run on the xCORE-200 explorerKIT core module board. The xCORE-200 explorerKIT core module board should be connected to both power and have the development adapters connected to a host machine to allow program download. This can be seen in the following image.

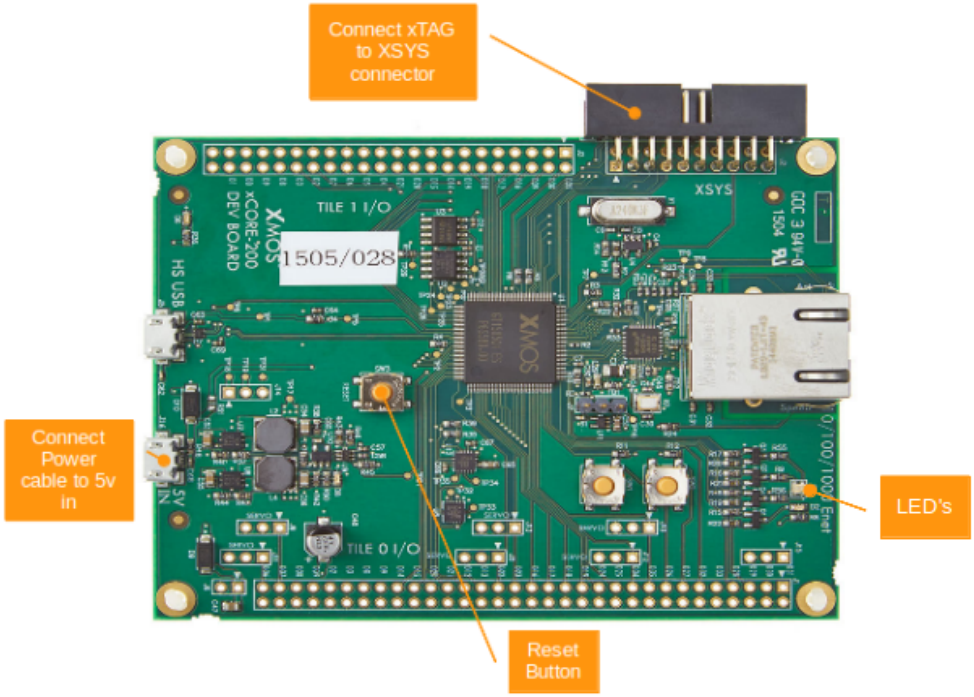

Figure 3: XMOS xCORE-200 explorerKIT

The hardware should be configured as displayed above for this example:

- The XTAG debug adapter should be connected to the XSYS connector and the XTAG USB cable should be connected to the host machine
- The xCORE-200 explorerKIT should have the power cable connected
- The RESET button can be used to repeatedly boot the xCORE-200 device from QuadSPI

APPENDIX B - Launching the example application

Once the example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer Studio the application and data.bin file can be written to the QuadSPI flash memory of the xCORE-200 explorerKIT module board.

Once built there will be a bin directory within the project which contains the binary for the xCORE-200 tile. The xCORE-200 binary has a XMOS standard .xe extension.

In this example the flash memory will be partitioned into 64KB (0x10000 bytes) as the boot partition and the remaining QuadSPI flash is used as the data partition. By keeping the boot partition small the amount of space available for application data is maximised and the boot time is reduced as the flash loader only has a small area of flash memory to search when looking for valid images.

B.1 Writing to flash memory from the command line

From the command line the xflash tool is used to download the code to the flash device on the xCORE-200 explorerKIT module board. The XFLASH option --factory is used to specify the application factory image. The XFLASH option --boot-partition-size is used to specify the amount of flash memory to reserve for the boot partition. The remainder of the flash memory is used for the data partition. The XFLASH option --data is used to write the data binary file data.bin to the data partition. The complete XFLASH command line is as follows:

```
> xflash --factory bin/qspi-persistent_stroage.xe --boot-partition-size 0x80000 --data src/data.bin
```

B.2 Writing to flash memory from xTIMEcomposer Studio

From xTIMEcomposer Studio there is the Flash As mechanism to edit the flash configuration and download the code to the flash device on the xCORE-200 explorerKIT core module board. Within the flash configuration editor the XFLASH options can be explicitly set. In this example, the Boot Partition Size checkbox is ticked and a corresponding value of 0x80000 is inserted. The --data option along with the path to the data.bin file is also added alongside Other XFLASH options.

Once XFLASH has sucessfully programmed the flash memory device on the xCORE-200 explorerKIT module board, the xCORE-200 device is reset. The application will boot from the boot partition and once booted the application will read the application data from the data partition. In this instance the LED's on the explorerKIT core module board should start rotating a pattern of LED's that are illuminated on and off, decreasing in speed as it does so.

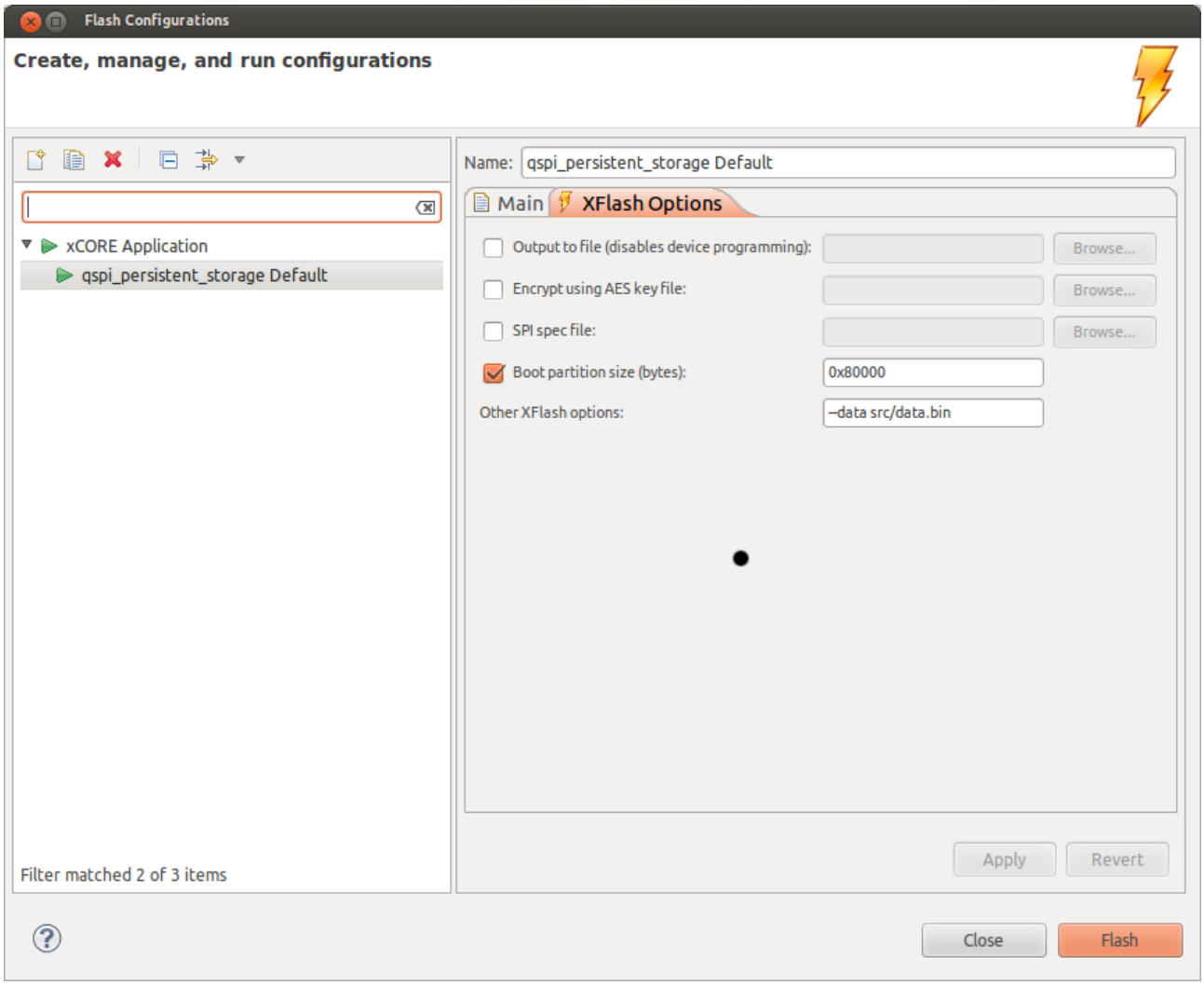

Figure 4: xTIMEcomposer studio flash configuration editor

APPENDIX C - 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

APPENDIX D - Full source code listing

D.1 Source code for qspi_persistent_storage.xc

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

/*
 * the LEDs on explorerKIT are:
 *
 * Single    : 0x01
 * RGB Blue  : 0x02
 * RGB Green : 0x04
 * RGB Red   : 0x08
 *
 * The 12 bytes stored in the data partition
 * drive various combinations of the 4 LED's'.
 * 0x01 - Single
 * 0x02 - RGB Blue
 * 0x04 - RGB Green
 * 0x08 - RGB Red
 * 0x03 - Single | RGB Blue
 * 0x05 - Single | RGB Green
 * 0x09 - Single | RGB Red
 * 0x06 - RGB Blue | RGB Green
 * 0x0A - RGB Blue | RGB Red
 * 0x0C - RGB Green | RGB Red
 * 0x0F - Single | RGB Blue | RGB Green | RGB Red
 * 0x00 - Off
 */

// Ports for QuadSPI access on explorerKIT.
fl_QSPIPorts ports = {
  PORT_SQI_CS,
  PORT_SQI_SCLK,
  PORT_SQI_SIO,
  on tile[0]: XS1_CLKBLK_1
};

// Port where the leds reside on explorerKIT.
on tile[0] : port led_port = XS1_PORT_4F;

// List of QuadSPI devices that are supported by default.
fl_QuadDeviceSpec deviceSpecs[] =
{
  FL_QUADDEVICE_SPANSION_S25FL116K,
  FL_QUADDEVICE_SPANSION_S25FL132K,
  FL_QUADDEVICE_SPANSION_S25FL164K,
  FL_QUADDEVICE_ISSI_IS25LQ080B,
  FL_QUADDEVICE_ISSI_IS25LQ016B,
  FL_QUADDEVICE_ISSI_IS25LQ032B,
};

// There are 12 bytes stored in the data partition.
#define NUM_LED_BYTES 12

int led_driver (void)
{
  char led_byte_buffer[NUM_LED_BYTES];  // Buffer to hold the 12 bytes read from the data partition.
  int delay = 50;                       // Initial delay 50 ms.
  int led_counter = 0;                  // A counter to count through the led patterns.

  // Connect to the QuadSPI device using the quadflash library function fl_connectToDevice.
  if(fl_connectToDevice(ports, deviceSpecs, sizeof(deviceSpecs)/sizeof(fl_QuadDeviceSpec)) != 0)
  {
    return 1;
  }

  // Read all 12 bytes from offset 0 in the data partition and store them in
  // the buffer led_byte_buffer using the flash library function fl_readData.
```

```
  if(fl_readData(0, NUM_LED_BYTES, led_byte_buffer) != 0)
  {
    return 1;
  }

  // Disconnect from the QuadSPI device.
  fl_disconnect();

  while(1)
  {
    delay_milliseconds(delay);                          // Wait.
    delay += 1;                                         // Increase the delay so the LED pattern gets slower.
    led_port <: led_byte_buffer[led_counter];           // Drive the next led pattern.
    led_counter++;                                      // Pick the next pattern.
    if (led_counter == NUM_LED_BYTES)                   // If we are at the last pattern,
    {
      led_counter = 0;                                  // then wrap around.
    }
  }

  return 0;
}

int main(void)
{
  par
  {
    on tile[0]:
    {
      led_driver();
    }

    on tile[1]:
    {
      while(1);
    }
  }

  return 0;
}
```