Application Note: AN00154

Using flash memory for persistent storage

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

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

Required tools and libraries

- xTIMEcomposer Tools - Version 13.2

Required hardware

This application note is designed to run on an XMOS startKIT.

The example code provided with the application has been implemented and tested on the startKIT but there is no dependancy on this board and it can be modified to run on any development board.

Prerequisites

- This document assumes familiarity with 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.
- This document assumes familiarity with flash memory, the xCORE flash library and the XMOS tool XFLASH.
- For descriptions 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 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 dash, backslash, pipe, and slash pattern on the startKIT's LED's. An application will be booted from the boot partition and will use the XMOS xCORE flash 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 flash memory.
- How to use the XFLASH option `--boot-partition-size` to partition flash memory into a boot partition and data partition.
- How to use the XMOS xCORE library to read the data from the data partition.

2 Using 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
  main.xc                       <-- source file for project
Makefile                        <-- make file for project
```

2.2 Makefile support for using the xCORE flash library

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

Options passed to xCORE build tools from makefile:

```
XCC_FLAGS = -g -report -lflash
```

2.3 The data.bin binary file

The data.bin binary file contains 16 bytes of data that will be written to the data partition. The data represents the 4 LED patterns that can be displayed on the startKIT xCORE module board where each LED pattern is made up of 4 bytes. 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:

```
0x00 0x0E 0x03 0x80 0x00 0x06 0x17 0x00 0x00 0x0A 0x16 0x80 0x00 0x0C 0x15 0x80
```

2.4 The application main() function

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

```
int main(void)
{
  char led_byte_buffer[NUM_LED_BYTES];  // Buffer to hold the 16 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 SPI device using the flash library function fl_connectToDevice.
  if(fl_connectToDevice(ports, deviceSpecs, sizeof(deviceSpecs)/sizeof(fl_DeviceSpec)) != 0)
  {
    return 1;
  }

  // Read all 16 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 SPI device.
  fl_disconnect();

  while(1)
  {
    delay_milliseconds(delay);                          // Wait.
    delay += 1;                                         // Increase the delay so the spinning bar gets slower.
    p32 <: getPattern(&led_byte_buffer[led_counter]);   // Drive the next led pattern.
    led_counter += NUM_LED_PATTERNS;                    // 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 flash library function is called to connect to the flash memory device
- A flash library function is called to read data from the data partiton
- A flash 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 flash library

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

```
#include <flashlib.h>
```

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

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

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

The parameter type fl_SPIPorts is a structure defined in the flash 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 startKIT XN file as the variable ports. This example also uses XS1_CLKBLK_1 to drive the clock for flash device access.

```
// Ports for SPI access on startKIT.
fl_SPIPorts ports = {
  PORT_SPI_MISO,
  PORT_SPI_SS,
  PORT_SPI_CLK,
  PORT_SPI_MOSI,
  on tile[0]: XS1_CLKBLK_1
};
```

The parameter type fl_DeviceSpec is a structure defined in the flash 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 SPI devices that are supported by default.
fl_DeviceSpec deviceSpecs[] =
{
  FL_DEVICE_ATMEL_AT25DF041A,
  FL_DEVICE_ST_M25PE10,
  FL_DEVICE_ST_M25PE20,
  FL_DEVICE_ATMEL_AT25FS010,
  FL_DEVICE_WINBOND_W25X40,
  FL_DEVICE_AMIC_A25L016,
  FL_DEVICE_AMIC_A25L40PT,
  FL_DEVICE_AMIC_A25L40PUM,
  FL_DEVICE_AMIC_A25L80P,
  FL_DEVICE_ATMEL_AT25DF021,
  FL_DEVICE_ATMEL_AT25F512,
  FL_DEVICE_ESMT_F25L004A,
  FL_DEVICE_NUMONYX_M25P10,
  FL_DEVICE_NUMONYX_M25P16,
  FL_DEVICE_NUMONYX_M45P10E,
  FL_DEVICE_SPANSION_S25FL204K,
  FL_DEVICE_SST_SST25VF010,
  FL_DEVICE_SST_SST25VF016,
  FL_DEVICE_SST_SST25VF040,
  FL_DEVICE_WINBOND_W25X10,
  FL_DEVICE_WINBOND_W25X20,
  FL_DEVICE_AMIC_A25L40P,
  FL_DEVICE_MACRONIX_MX25L1005C,
  FL_DEVICE_MICRON_M25P40,
  FL_DEVICE_ALTERA_EPCS1,
};
```

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

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

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

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

```
// Read all 16 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 flash 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 four LED patterns is 16 and is defined by `NUM_LED_BYTES`.

```
// There are 16 bytes stored in the data partition.
#define NUM_LED_BYTES 16
```

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 16 bytes read from the data partition.
```

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

2.8 Finishing with the xCORE flash library

The flash library provides the function `fl_disconnect()` that is used to close the connection to the flash device. When this function is called it releases the ports and clock resources used by the flash 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 SPI 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 startKIT core module board with the pattern change reducing in speed.

```
while(1)
{
  delay_milliseconds(delay);                          // Wait.
  delay += 1;                                         // Increase the delay so the spinning bar gets slower.
  p32 <: getPattern(&led_byte_buffer[led_counter]);   // Drive the next led pattern.
  led_counter += NUM_LED_PATTERNS;                    // Pick the next pattern.
  if (led_counter == NUM_LED_BYTES)                   // If we are at the last pattern,
  {
    led_counter = 0;                                  // then wrap around.
  }
}
```

There are four patterns in total with each pattern made up of 4 bytes. The utility function `getPattern(char * led_byte_buffer)` is used to build the 4 byte pattern that is displayed on the LED's from the `led_byte_buffer`. This function returns an integer value where the byte value at position 0 of the `led_byte_buffer` is shifted into the MSB of the integer value and the byte value at

position 3 of the led_byte_buffer is the LSB.

```
// Utility function for building the led pattern from the led_byte_buffer.
int getPattern(char * led_byte_buffer)
{
  // LED pattern is made up of 4 bytes shifted into an integer value.
  int led_pattern = led_byte_buffer[0] << 24 |
                    led_byte_buffer[1] << 16 |
                    led_byte_buffer[2] << 8 |
                    led_byte_buffer[3];

  return led_pattern;
}
```

The integer value returned from getPattern is then output to the port p32.

```
// Port where the leds reside on startKIT.
port p32 = XS1_PORT_32A;
```

APPENDIX A - Example Hardware Setup

This application note example is designed to run on the xCORE startKIT module board which has 256KB (0x40000 bytes) of flash memory and 9 LED's positioned as a 3x3 square. The xCORE startKIT module board should be connected to a host machine via the USB cable supplied with the board to allow program download.

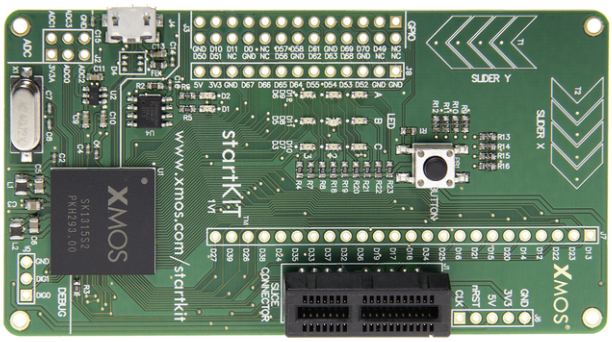

Figure 1: XMOS startKIT core module board setup

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 flash memory of the xCORE startKIT module board.

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

In this example the flash memory will be partitioned into 32KB (0x8000 bytes) as the boot partition and the remaining 224KB (0x38000 bytes) 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 startKIT 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 --target=STARTKIT --factory bin/app_using_data_partition.xe --boot-partition-size 0x8000 --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 startKIT 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 0x8000 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 startKIT module board, the xCORE 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 startKIT module board should start rotating an led pattern in the sequence pipe, slash, dash and backslash, decreasing in speed as it does so.

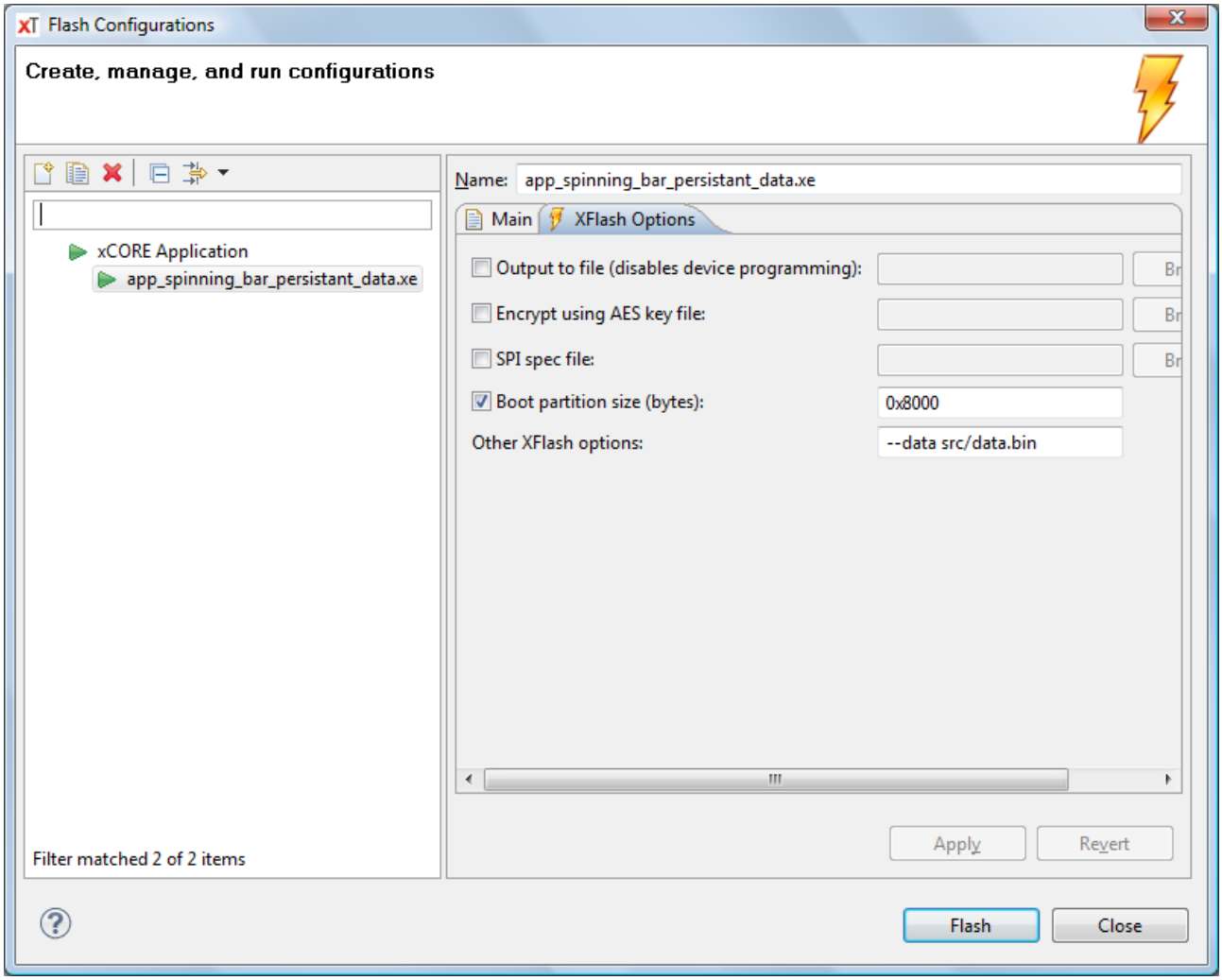

Figure 2: xTIMEcomposer studio flash configuration editor

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

XMOS startKIT Hardware Guide

http://www.xmos.com/published/startkit-hardware-manual

APPENDIX D - Full source code listing

D.1 Source code for project app_factory_image_button_down

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

/*
 * the patterns for each LED on startKIT are:
 *   0x80000 0x40000 0x20000
 *   0x01000 0x00800 0x00400
 *   0x00200 0x00100 0x00080
 *
 * As the leds go to 3V3, 0x00000 drives all 9 leds on,
 * and 0xE1F80 drives all nine leds off.
 * The 16 bytes stored in the data partition
 * drive a dash, backslash, pipe, and slash.
 * 0x000E0380
 * 0x00061700
 * 0x000A1680
 * 0x000C1580
 */

// Ports for SPI access on startKIT.
fl_SPIPorts ports = {
  PORT_SPI_MISO,
  PORT_SPI_SS,
  PORT_SPI_CLK,
  PORT_SPI_MOSI,
  on tile[0]: XS1_CLKBLK_1
};

// Port where the leds reside on startKIT.
port p32 = XS1_PORT_32A;

// List of SPI devices that are supported by default.
fl_DeviceSpec deviceSpecs[] =
{
  FL_DEVICE_ATMEL_AT25DF041A,
  FL_DEVICE_ST_M25PE10,
  FL_DEVICE_ST_M25PE20,
  FL_DEVICE_ATMEL_AT25FS010,
  FL_DEVICE_WINBOND_W25X40,
  FL_DEVICE_AMIC_A25L016,
  FL_DEVICE_AMIC_A25L40PT,
  FL_DEVICE_AMIC_A25L40PUM,
  FL_DEVICE_AMIC_A25L80P,
  FL_DEVICE_ATMEL_AT25DF021,
  FL_DEVICE_ATMEL_AT25F512,
  FL_DEVICE_ESMT_F25L004A,
  FL_DEVICE_NUMONYX_M25P10,
  FL_DEVICE_NUMONYX_M25P16,
  FL_DEVICE_NUMONYX_M45P10E,
  FL_DEVICE_SPANSION_S25FL204K,
  FL_DEVICE_SST_SST25VF010,
  FL_DEVICE_SST_SST25VF016,
  FL_DEVICE_SST_SST25VF040,
  FL_DEVICE_WINBOND_W25X10,
  FL_DEVICE_WINBOND_W25X20,
  FL_DEVICE_AMIC_A25L40P,
  FL_DEVICE_MACRONIX_MX25L1005C,
  FL_DEVICE_MICRON_M25P40,
  FL_DEVICE_ALTERA_EPCS1,
};

// There are 16 bytes stored in the data partition.
#define NUM_LED_BYTES 16

// There are 4 LED patterns made up of 4 bytes each.
#define NUM_LED_PATTERNS (NUM_LED_BYTES / 4)
```

```
// Utility function for building the led pattern from the led_byte_buffer.
int getPattern(char * led_byte_buffer)
{
  // LED pattern is made up of 4 bytes shifted into an integer value.
  int led_pattern = led_byte_buffer[0] << 24 |
                    led_byte_buffer[1] << 16 |
                    led_byte_buffer[2] << 8 |
                    led_byte_buffer[3];

  return led_pattern;
}

int main(void)
{
  char led_byte_buffer[NUM_LED_BYTES];  // Buffer to hold the 16 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 SPI device using the flash library function fl_connectToDevice.
  if(fl_connectToDevice(ports, deviceSpecs, sizeof(deviceSpecs)/sizeof(fl_DeviceSpec)) != 0)
  {
    return 1;
  }

  // Read all 16 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 SPI device.
  fl_disconnect();

  while(1)
  {
    delay_milliseconds(delay);                          // Wait.
    delay += 1;                                         // Increase the delay so the spinning bar gets slower.
    p32 <: getPattern(&led_byte_buffer[led_counter]);   // Drive the next led pattern.
    led_counter += NUM_LED_PATTERNS;                    // 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;
}
```