ECSE 444: Microprocessors
Lab 6: Quad SPI Flash

Abstract

In this lab you will learn how to use the B-L475E-IOT01A's on-board 64 Mbit flash storage, erasing, writing, and reading this memory using the quad SPI (QSPI) interface. This will be demonstrated with an extension of your work in Lab 4.

Deliverables for demonstration

- C implementation of erasing, writing, and reading using QSPI
- C implementation of DAC using timer, DMA, and QSPI
- Audible confirmation of sine signals, with at least six different frequencies, changing about every half second

Grading

- C implementation of QSPI interface using board support package (BSP)
    - 20% Erasing
    - 20% Writing
    - 20% Reading
- C implementation of DAC using timer and DMA
    - 20% DAC triggers QSPI
- At least six different frequencies, changed automatically about every half second
    - 20%

Changelog

- 24-Aug-2020 Initial revision.

Overview

In this lab, we'll learn about the flash storage integrated with our development board. Flash is more complex than SRAM: before new values can be written to flash, old values need to be erased. Because this flash is accessed over the serial peripheral interface (SPI), interacting with it is more complex than the integrated peripherals we've worked with to date. Fortunately, STM has written a board support package (BSP), or driver, that simplifies the otherwise complex process of addressing the memory. We'll configure the pins of our process as before, but the BSP will take care of the rest of the configuration for us. We'll extend the final result of Lab 4, playing a sequence of tones: we'll save more tones, and more samples for each tone, making saving it all in SRAM impractical. We'll therefore have to erase a portion of the flash memory, and then write the samples for our tones to it. We'll read these tones out of flash so that a new tone can be played about every half second, in a process driven entirely by interrupts.

Resources

B-L475E-IOT01A User Manual
HAL Driver User Manual
MX25R6435F Datasheet
STM32L475VG Datasheet
STM32L47xxx Reference Manual
Quad-SPI interface on STM32 MCUs Application Note

Configuration

Initialization

Start a new project in STM32CubeMX, and initialize it as in earlier labs. For this lab, we need:

- One DAC output channel, and associated timer; and,
- Quad SPI to the 64 Mbit MX25R6435F flash chip.

QSPI operations can take some time to complete. To assist with debugging, it is worthwhile considering using LEDs as progress and/or error indicators. The green LED on PB14 red LED on PE3 may be useful for this purpose. Note that the LED on PE3 lights when the GPIO output is low.

QSPI Flash

Flash memory, a non-volatile storage technology, operates fundamentally differently from the on- and off-chip memory we're used to programming with, RAM. RAM can be read and written in any order, at any address, at any time (provided that it is powered). Reading and writing takes about the same amount of time, and is generally fast.

Flash, on the other hand, must be manipulated with greater care. Before flash can be written, it must first be erased, block by (e.g., 64KB) block. This sets all bits in the block to 1. The block can then be programmed, during which any 0s are set. Erasing and writing are power intensive and slow; reading is generally faster, though still not as fast as RAM.

Our board provides a quad serial peripheral interface (QSPI) to an on-board 64 Mbit flash chip. QSPI implements a synchronous serial connection to a peripheral using four data lines, a clock signal, and chip select signal. SPI peripherals are interacted with using *commands*. Commands have a large number of fields that specify the desired behavior: Should the device return data, or save new data? What address should be used? Once set, the command and parameters are sent to the device. A large number of parameters are available for configuring a variety of commands. For a lot more details, see Quad-SPI interface on STM32 MCUs Application Note.

Frankly, it's a bit overwhelming.

Fortunately, STMicroelectronics has provided a board support package (BSP), which defines functions that simplify a number of basic operations, including setting device-specific parameters such as memory size.

In this case, all we need to do to configure the device is enable the appropriate pins. From the peripheral list on the left hand side of MX, choose *Connectivity*, and then *QUADSPI*. In *Mode* and under *Single Bank*, select *Quad SPI Line*. This will enable six pins. Two are set correctly: clock (`QUADSPI_CLK`) and chip select (`QUADSPI_NCS`), which are mapped to PE10 and PE11. The IO pins, however, are not. Refer to the schematic in B-L475E-IOT01A User Manual to identify the appropriate pins for the four IO signals. Once you've remapped the signals, generate your code.

Implementation

The first step is to copy the BSP files into your project:

- `stm32l475e_iot01_qspi.c`
- `stm32l475e_iot01_qspi.h`
- `mx25r6435f.h`

and include `stm32l475e_iot01_qspi.h` in `main.c`.

Then, add a call to `BSP_QSPI_Init()` after `USER CODE BEGIN 2`. MX will generate a HAL handle `hqspi` for QSPI functions. This has been configured according to the parameters in MX, *which we did not change from their defaults.* `BSP_QSPI_Init()` configures a different handle (defined in `stm32l475e_iot01_qspi.c`) with parameters corresponding to the particular device on the board (defined in `mx25r6435f.h`). When you call other BSP_QSPI functions, it uses this second, appropriately configured handle.

Erase, Write, Read

At this point, it is worthwhile experimenting with the BSP_QSPI functions. Again, flash must be erased before it can be written. Three erase functions are available to you, which will erase a single sector (4k bytes), block (64KB), or the entire chip. It is worth noting that:

- Erasing the entire chip is time consuming, and not recommended (it is unnecessary in this lab).
- The sector erase function is non-blocking, meaning that the processor will not wait for the erase to complete before continuing to execute from `main()`.
- As such, block erase is probably the most convenient for the purposes of this lab.

I recommend trying to erase a block, write data to it, and read the data back out to confirm that the write operation was successful. I also recommend using the following structure to make BSP_QSPI function calls:

```
if (BSP_QSPI_Read((uint8_t *) tone, toneAddr, toneSamples) != QSPI_OK)
      Error_Handler();
```

This ensures that your `Error_Handler()` function (which is empty upon generation) is called if anything goes wrong with the function. My `Error_Handler()` for this lab is:

```
HAL_GPIO_WritePin(LEDError_GPIO_Port, LEDError_Pin, GPIO_PIN_RESET);
__BKPT();
```

This code turns on a red LED, and then halts the debugger with a breakpoint instruction. I have added similar code to a number of interrupt handlers in `stm32l4xx_it.c`, e.g., `HardFault_Hander()`, which is called under a variety of circumstances.

Generate, Store, Read on Repeat

Now write a program that stores at least six different tones in flash, and then reads and plays them on your speaker. The requirements of your program are as follows. Your program must

- Store at least six different tones in flash memory using QSPI
- Store approximately half a second worth of samples for each tone
- Play through each tone exactly once before moving on to the next
- Move from one tone to the next without user intervention
- Output your tones at 44.1kHz

As specified, it should not be possible to store all tones in on-chip SRAM (which is limited to 96KB). *That's the point*.

Notes

- Code re-use is good programming practice. Draw on your experience in Lab 4.
- Test your tones at least once before you save them to flash to confirm they are behaving correctly; this is a slightly different generation use case in Lab 6 than in Lab 4.
- The DAC has two interrupt handler callback functions when using DMA, one for when half the array is tran`sferred, and another for when the array is fully transferred; unlike DFSDM DMA, DAC DMA triggers these callbacks in both normal and circular modes.
- If things don't sound right, double check your DMA settings. Does your playback rate match your DMA rate?
- Using DMA means that if you debug with breakpoints, program execution may be different from when it simply runs (because the relative timing of events changes). Don't forget about execution tracing using the ITM! (See Lab 1.)