ES Coursework Part 2

February 27, 2017

Contents

1 Introduction

Having completed the first coursework, you should be familiar with the principles of programming for embedded systems, working with the K70 boards, and programming with MQX. In this coursework you will be programming for a bare metal system (i.e., one which is not running an RTOS).

The main objective of the coursework is to produce a configurable audio filter system. Audio data (in this case, signed 8-bit PCM audio at 8000Hz) will be fed into the virtual serial port on the board, filtered using a selectable band-pass filter, and then sent back over the serial port. The audio bitrate is kept low in order to fit within the data rate of the serial port. You should implement four different band pass filters, and allow the user to cycle between them at any time by using the two push-buttons next to the LED array. The LEDs should indicate which filter is currently selected.

Completing this coursework will require writing drivers for several hardware devices, and then using these drivers to implement the audio filter. Most of the information on the hardware devices can be found within the provided documentation. You may need to do some additional research or experimentation in order to obtain an understanding of how the various devices work.

When completing this coursework, you should try to be mindful of code size, energy usage, and performance. Your solution *must* be able to filter the input signal in **real time** or better and should not make use of excessively sized buffers or other temporary storage. When implementing the filter, you may choose to use floating point arithmetic and the floating point unit, or to use fixed point arithmetic and the DSP-like instructions provided by the Cortex-M4F MCU.

It is recommended that you work on each component of your system separately. For example, you may wish to create the UART driver and ensure that it is working before you move on to attempting to use the push-buttons (or vice-versa). You should also make use of multiple source and header files in order to keep your work organized.

The reference documentation for this part of this coursework is located in `/group/teaching/espractical/Part2/Documentation/`.

2 Bare-metal Development

Developing for a system not running any kind of OS or RTOS is known as 'bare-metal' development. This poses some challenges and problems which do not exist when programming for an RTOS such as MQX or a full OS such as Linux. Many features provided in an RTOS (such as hardware drivers and task management) do not exist in a bare metal environment.

The top board of the Freescale Tower features a Freescale K70 System-on-Chip, which contains an ARM Cortex-M4F Microcontroller along with some SRAM and many hardware devices. The Cortex-M4F implements the ARMv7-M Architecture and features a floating point unit and some DSP-like extension instructions. For more information about the K70 SOC, refer to `K70 Reference Manual.pdf` in the documentation directory.

2.1 Interrupts and Exceptions

Interrupts and exceptions are used to communicate synchronous events (such as memory access errors) and asynchronous events (such as messages from devices)

to the core. Interrupts and exceptions typically execute in a slightly different environment to normal user code - for example, it may execute with an increased privilege level (allowing access to OS Kernel memory areas) or using a different stack.

Interrupts are usually passed to a core from a separate (but closely connected) interrupt controller. In the case of the Cortex-M4, the interrupt controller is known as the NVIC - the Nested Vectored Interrupt Controller. This indicates that the controller can handle nested interrupts (i.e., an interrupt can be taken while a lower priority interrupt handler is executing) and that the interrupt vectors (the addresses of interrupt handlers) are provided to the core by the interrupt controller, rather than having the core look them up itself. These two features allow interrupts to be handled extremely efficiently.

The NVIC is configured by reading/writing to specific memory locations. A separate table, the Vector Table, is kept elsewhere in memory. Although the NVIC supports having the location of this table change at runtime, it is recommended that you leave the table at its default location (address 0). Note that this places the vector table in flash memory, which cannot be easily modified at runtime. The first 16 vectors are used for exceptions, and subsequent vectors are for external interrupts. Interrupts can be enabled/disabled individually, and can also be prioritized.

For more information about the NVIC, see the `ARMv7-M Architecture Reference Manual.pdf`, page 750. For more information about the ARMv7-M Exception Model, see the `ARMv7-M Architecture Reference Manual.pdf`, page 631.

2.2 Devices

Devices are typically configured via two channels - by mapping configuration and status registers into regions of memory (memory mapped I/O), or by using special instructions to configure devices. For the Cortex-M4, most devices are configured via memory mapped I/O. Freescale provide a header file containing C macros and struct definitions which can be used to access many of the device registers available on the MCU and it is recommended that you use this rather than accessing the devices directly. For more information on some of the devices you might use in this coursework, see Section 3.

2.3 The Core at Reset

When the core is powered up or reset in any way, after performing some internal initialization it performs a Reset Exception. This is essentially the point at which our code begins to execute - in the Reset Exception Handler. The core loads an initial stack pointer from address 0, loads the address of the Reset Handler from address 4, and then jumps to the Reset Handler. Since we're programming in C, the C library we are using provides some code to do initial setup of the core, so the reset handler should be `__thumb_startup`. `__thumb_startup`

will call `init_hardware()` (where the core can be configured, devices initialised etc), and then eventually call `main()`.

In order to present a consistent and safe environment to code running on the core, many of the features provided are initially disabled. For example, the FPU is disabled, interrupts are disabled etc. If you wish to use any of these features, you must enable them using the relevant mechanisms.

3 Hardware Components

3.1 Clock Management

In order to save energy, many systems support a technique known as clock gating. This allows the core to disable individual devices, or groups of devices, by disabling the clock signal which is driving those devices. The K70 device starts up with all non-core clocks disabled, so before using any device, the associated clock must be enabled.

In addition, it may be desirable to increase or decrease the frequencies of the various clock signals in use, in order to improve performance or save power, or to allow timing critical components to function at all. The K70 contains several Phase-Locked-Loop and Frequency-Locked-Loop devices which can be used to generate clock signals at a variety of frequencies. However, for this coursework it is recommended that you use the external oscillator from the Ethernet board which is locked at 50MHz.

The clocks are controlled using two devices - the System Integration Module, and the Multipurpose Clock Generator. For more information on these two devices, see the K70 Reference Manual, pages 319 and 633 respectively.

3.2 Pin MUXing

Although modern microcontrollers have access to a large number of internal devices and device controllers, they are frequently extremely constrained when it comes to the number of external wires, or 'pins', available. Certain types of device such as external RAM and buses frequently use large numbers of pins, as they require parallel address and data wires in order to provide acceptable performance.

Although it is possible for multiple active devices to share pins, this is often undesirable as it complicates the use of these devices (as the pin mux configuration must be reset for each access to a particular device). For this reason, pins are assigned to devices by the system designer in such a way that most applications are able to avoid pin sharing.

For this coursework you will only be using a small number of devices which use external pins, so you will not need to manage pin sharing. However, you will still need to ensure that the devices you are using have the relevant pins MUXed correctly.

Pin MUXing is controlled using the PORT devices. These must have their clocks enabled prior to use. There are 6 PORT devices, each one responsible for a different group of pins. For more information on the PORT device, see the K70 Reference Manual, page 299. For information on which pins are assigned to each device, see the table starting on page 275.

3.3 GPIO

The GPIO (General Purpose Input/Output) device is used to 'read' and 'write' to physical pins. This can be used to implement communications protocols in software or to control external devices. For this coursework, you should use the GPIO device to light the LEDs in the capacitive touch panels in order to indicate which of the audio filters is currently selected.

There are 6 GPIO devices, one per port. They share clocks with their associated PORT device. For more information on the GPIO devices, see the K70 reference manual, page 2147.

3.4 UART

The UART (or Univeral Asynchronous Receiver/Transmitter) is a serial communications device. The UART is 'Universal', in that it can be configured with a wide variety of data format and transmission speeds, and is 'asynchronous', meaning that the sender and reciever do not share any kind of clock signal. In a standard configuration of 8 data bits, 1 stop bit, and no parity (known as 8N1), the effective data rate (in KB/s) is 1/10th of the selected Baud rate. This means that for 8-bit, 8000Hz audio (62.5Kb/s or 7.8KB/s) a rate of at least 72000 baud is required - the closest standard baud rate above this is 115200.

Using UART devices is fairly straightforward when communicating with serial ports. Typically it is only necessary to enable the UART device, ensure that the correct baud rates are selected on the transmit and receive sides (i.e., on the UART device and on the serial port it is connected to), and then write data to the UART's data register to transmit, and read data from the UART's data register - when data is available - to receive. Modern UART devices feature FIFO buffers in order to allow for more time for the core to do other things between servicing the UART. Although many serial interfaces implement complex control protocols, in this case we are using only the TX and RX lines.

For more information on the UART device, see the K70 reference manual, page 1885.

3.5 PIT

The PIT (Periodic Interrupt Timer) is a device used to deliver interrupts to the core at a given, programmable interval. This device can be used to, for example, poll for changes in another device. The PIT timer continues even while the core is in low power modes, making it useful for low power timers.

For more information on the PIT, see the K70 Reference Manual, page 1353.

4 Audio Filtering

For this coursework, you will be implementing a variety of band pass filters. Since this is not a digital signal processing course, you do not need to understand exactly how these filters work or are created. Instead, we recommend that you use a tool such as that found at `http://www.micromodeler.com/dsp/` in order to find out how these filters can be implemented. Importantly, you must implement a streaming filter i.e. a filter which receives and processes one sample at a time. An order 4 filter is sufficient.

A bandpass filter takes a signal as its input, and produces a signal where components of the signal outside of the given band are attenuated (i.e. made 'quieter'). A perfect filter would completely remove from the input signal any component which is outside of the pass band. However, this is not typically achievable - for example, examine the frequency response curve in Figure 1. Although the desired pass band of the filter is 1000Hz-1500Hz, note that the response at 800Hz (i.e., 0.1) is not zero. The frequencies at which the filter attenuates the input signal by -3dB (i.e., to approximately 70%) are known as the 'corner frequencies'. You should use a tool (such as that described above) in order to generate filters with the properties described in Table 1.

Filter	Low Frequency	High Frequency
1	500Hz	1000Hz
2	1500Hz	1750Hz
3	2000Hz	2500Hz
4	3000Hz	3750Hz

Table 1: Corner frequencies to be used in the band-pass filters

5 Provided Resources

In order to help you get started with the coursework, and to help you test your solution, we have provided an example bare metal program (which can also serve as a template), as well as a number of example audio files to be filtered, and scripts to send these files to the K70 board and record the response.

5.1 The Example

As with the first coursework, we have provided a small example project which can be used as a template. This project consists of the following files:

1. `bareboard_flash.lcf` - A linker script, which describes how the program should be laid out in memory
2. `led.[ch]` - A simple set of functions for accessing the LEDs on the board.

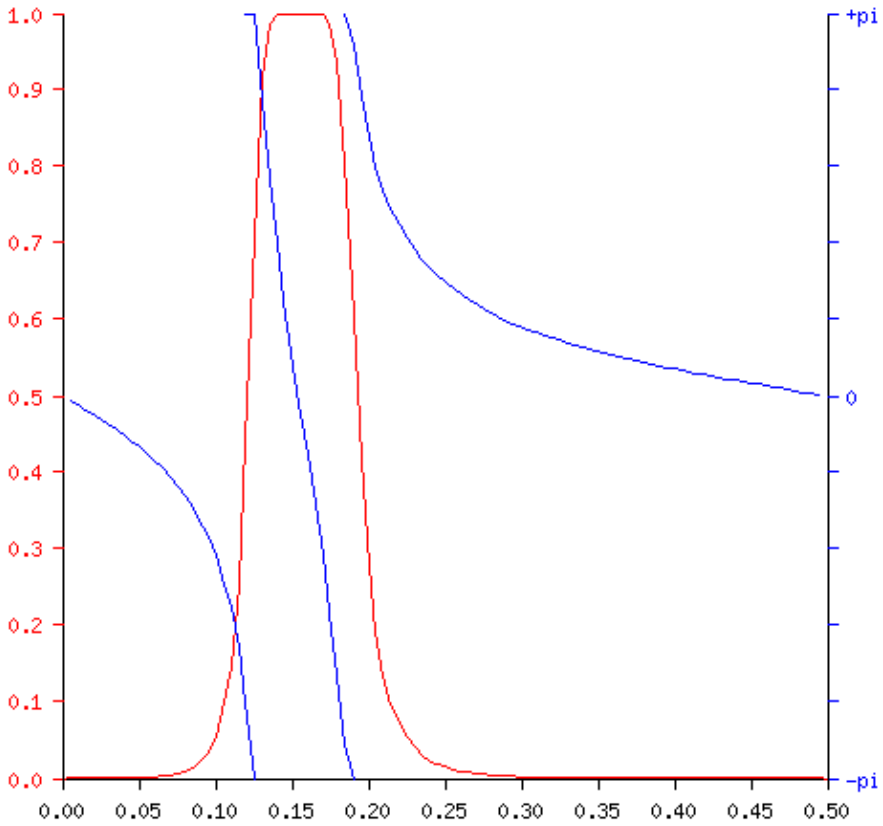

Figure 1: The response curves for a simple band pass filter. The filter passes 1000Hz-1500Hz of an 8000Hz signal and is of order 4. The X Axis represents the frequency of the input signal. The Y Axis on the left and red curve represent the filter response (i.e., a response of 1.0 indicates that the amplitude of the output signal is the same as that of the input signal, and a response of 0.5 indicates that the amplitude of the output would be halved), and the Y Axis on the right and blue curve represent a phase offset applied to the output signal.

3. `main.c` - A program which sets up the board, then uses the provided LED library to flash the red and blue LEDs.

4. `Makefile` - A GNU Makefile for building the example. If you add additional C source files to your project, you will need to insert their names into the relevant part of this file.

5. `MK70F12.h` - A header file containing macros allowing you to access all of the devices provided by the K70. You don't need to read all of this file, but you should look at the macros used elsewhere in the template to see what they actually do.

6. `vectors.[ch]` - Files containing the Exception Vector table definition and some configuration options.

You will still need to source the `setup.sh` file in (`/group/teaching/espractical/OpenOCD/setup.sh`) as before prior to building the example program (or any program based on it).

5.2 Test Audio Files

So that you are able to test your filters, we have provided a number of audio files containing tones at various frequencies, as well as frequency 'sweeps' which containing tones starting at low frequencies and rising to higher frequencies over time. These are encoded as signed 8-bit, 8KHz Wav files. We have also provided scripts for converting from Wav files to raw audio files suitable for transmission onto the board, and vice-versa, as well as sending data to the board and recording the response, as described below.

The test audio files are available in `/group/teaching/espractical/Part2/AudioSamples`.

5.3 Scripts

The `setup.sh`, `flash.sh`, `debug.sh` and `console.sh` scripts are all still available and should be used for setting up a build environment, flashing programs to the board, debugging programs on the board and viewing text UART output. Note that once you are sending/receiving audio files over the UART you will no longer be able to print debug messages over it!

Three additional scripts are also provided. The first, `pipe.sh`, sends a file over the UART connection to the board, and records the response. You should use this script to test your audio filter program. Note that this script assumes a baud rate of 115200 on the K70's UART.

The second, `wav2raw.sh`, converts a .wav audio file into a signed 8-bit PCM, 8KHz RAW audio file suitable for sending to the board. The third script, `raw2wav.sh`, converts back from a signed 8-bit PCM, 8KHz RAW to a similarly encoded .wav file.

All of the scripts are available in `/group/teaching/espractical/Part2/Scripts`.

5.4 Documentation

There are two main sources of documentation for this coursework (alongside this handout). The one you are likely to refer to the most is the K70 Reference Manual. This document describes the K70 device in detail, including some information on the Cortex-M4F core it contains and the devices and peripherals attached to it. Although the manual is quite long (and intimidating!) at 2259 pages, you will likely only need to refer to the sections mentioned in this handout.

The second document you might refer to is the ARMv7-M Architecture Reference Manual. This document is produced by ARM Holdings and primarily contains a low-level description of the ARMv7-M Architecture (such as the various instructions, how interrupts and exceptions are handled etc.).

One final document provided is the K70 Tower Board Reference Manual. This describes the Freescale Tower System board on which the K70 device is mounted. This document contains information such as which GPIO pins the LEDs are attached to, which TSI sensor channels are enabled etc.

All of the documentation is available in `/group/teaching/espractical/Part2/Documentation`.

6 Submission

The coursework should be submitted using the submit system, by Friday, March 24th, 6PM. You should include a readme file indicating how your solution is structured.

```
submit es 2 es_part_2/
```

7 Marking

The submissions will be marked on the following criteria:

1. **Button library** (20 marks): the design and implementation of the methods which are used in the system to initialize and get information about button presses;
2. **UART library** (20 marks): similar to the button library, this one regards the UART code;
3. **Filter library** (20 marks): the way the generated filter code is integrated into the system;
4. **Overall integration** (20 marks): the way the modules are put together. Ideally, the main function will read like English, containing only object construction and invocations to high-level methods that manage different parts of the system e.g. displaying the current filter with LEDs;
5. **Floating Point Unit** (5 marks): the FPU has been correctly initialized;

6. **Interrupts** (5 marks): interrupts have been used;

7. **Code quality** (5 marks): quality of code and comments.

8. **README quality** (5 marks): quality of the README file.

The reason you are given the marking scheme is to help you guide your efforts, rather than encouraging you to perform an economics exercise of computing ROI. Aim for a 100.