Principles of Embedded Software

*Project 3 (**125** Pts) - Due Wednesday 12/6*

Logging, SPI, DMA, & Profiling

Overview

Description

In this project assignment, you will write a binary logger application for a microcontroller that uses your prior work with the UART and Circular Buffers. A logger enables one to report data to a PC, this helps to interpret what is happening with the embedded system. This application will be extended to use DMA as a mechanism to off-load transmitting of the data. The execution time of various operations will be documented using a profiler and the systick counter. In the last part of this project, you will leverage this code to communicate with the Nordic Wireless transceiver via the Serial Peripheral Interface (SPI).

Outcomes

After completing this assignment, you will have:

1. Used DMA to perform data memory movements
2. Enhanced your circular buffer code to be more efficient and robust
3. Developed a binary logger to send data to a UART
4. Profiled your library functions to determine execution time
5. Created a SPI driver to interface with a Nordic Wireless transceiver

Guidelines

Projects can be done in teams of 2 students. All coding MUST adhere to the C-programming style guidelines posted on D2L. Failure to do so will result in point deductions. Report questions/material should be turned in the repository in the form of code documentation and readme support in appropriate files in your repository. You may not use a processor expert (or wizard) to develop any of your code.

You must turn in a code dump report. This should be a single file in a **.pdf** format that is submitted to the Project 3 Dropbox D2L. This allows us to check for plagiarism. No other formats will be accepted.

Resources:

- KL25 Sub-Family Technical Reference Manual
- GCC inline keyword - https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Inline.html
- System Tick - Making Embedded Systems (Elecia White) - Pgs 138-140
- SPI - Making Embedded Systems (Elecia White) - Pgs 165-168
- Profiling - Making Embedded Systems (Elecia White) - Pgs 230 -234
- Logging - Making Embedded Systems (Elecia White) - Pgs 230-232
- Direct Memory Access - Making Embedded Systems (Elecia White) - Pgs 184-193
- Diagram Tool - https://www.draw.io/

Version Control and Platform Support

Continuing from the previous project, this project's code must be turned in via a git repository. You may use your bitbucket/github account and create your own repository in your account. Remember, you need to share your repository with the instructor team so we can grade you more effectively. If your group for this project has changed since Project 2, make sure to inform the TAs.

You will need to make a minimum of 5 commits for project 3 and there need to be commits from both group members. You cannot commit all your code at once. Your commit history must include commits for the following design steps in your design:

- Design stubs (C-programming): This commits represents your module outline. This should include all defined prototypes, commented function descriptions, file descriptions, and empty function definitions. No function implementations. You will need a few of these as there are multiple modules to code. Here are some examples
  - UART Stubs
  - Logger Stubs
  - DMA Stubs
  - Circular Buffer Stubs
- Feature Commits: These commits represent your actual feature developments.
- Bug Fixes: These commits should represent any bugs that you found and fixed.

Place a git annotated tag with the name **project-3-rel** where you want the project to be graded. This tag **MUST** be placed before the due date of the project.

- https://git-scm.com/book/en/v2/Git-Basics-Tagging

This project requires you to use an IDE, specifically the Kinetis Design Studio IDE. You need to support three target platforms for certain components in this project. Here is a breakdown on where code should be able to run:

- Logger - BBB, Host, KL25z
- Profiling - BBB, Host, KL25z
- DMA - KL25z
- SPI - KL25z

Memmove and Memzero with DMA (FRDM)

A new piece of hardware will be used to offload some of the processor's work when transferring data. Direct Memory Access (DMA) allows the transfer of large amounts of data with very little CPU interaction. Thus, the CPU can spend time performing other useful operations, while the data transfer occurs. The DMA controller has access to the bus controller and can directly pass data between peripheral devices and memory or just between memory interfaces.

Extend the memzero() and memmove() functions to have some architecture specific code. Write two new functions:

- memmove_dma(...)
- memset_dma(...)

DMA can be setup to transfer 1, 2, or 4 bytes. Depending on the type of transfer, you may be limited to the transfer size. However, you will be analyzing the performance of using the DMA with varying transfer sizes compared to the **memmove()** and **memzero()** functions. Any type of overlap needs to be accounted for with the function that configures the DMA just like the memmove functions.

General Requirements

- Configure DMA for ONLY the FRDM Board. (BBB is not supported in this task)
- Design your functions to work on 1, 2, & 4 byte transfer sizes
- Prototypes for the DMA operation should be similar to your previous memmove and memzero functions
- DMA should trigger an interrupt when the transfer has completed

Profiling (BBB & FRDM)

For this part, you will be gathering statistics on some of your code's execution time and on some additional functions provided by the standard libraries. On the FRDM, use the default 20 MHz core clock (instructions will be running as fast as 50 nsecs per instruction). Configure a counter with a fast count rate so you can have microseconds worth of resolution in the timing. Because the timing can vary from one execution of the code to another, the code should be executed several times to get an average execution time.

Get a start time of the operation before you call a function or trigger DMA to start a transfer. When a function completes or when DMA finishes, an end time must be obtained. Enable the DMA to generate an interrupt when the operation has completed so that you can calculate the runtime for the DMA-centric operations. For example, see the pseudo-code below:

```
// Does not have to be a function, can just read a global or the
// counter register
start_time = gettime();
ret = memmove(src, dst, length);
end_time = gettime();
```

Profile your code to find the time it takes to move:

1. 10 Bytes
2. 100 Bytes
3. 1000 Bytes
4. 5000 Bytes

Byte transfers can be created using a very large array of memory from the stack or memory from the heap. **However, if you use the heap or the stack, make sure you can allocate enough space by adjusting your compiler settings at build time [hint].**

Obtain the execution times of the following operations and add them to your report for all of the size lengths above.

1. memmove
    a. Standard Library Version (BBB and FRDM)
    b. Your Non-DMA version (BBB and FRDM)
    c. Your Non-DMA optimized version using -O3 (BBB and FRDM)
    d. Your DMA Version (FRDM)
2. memzero/memset
    a. Standard Library Version (BBB and FRDM)
    b. Your Non-DMA version (BBB and FRDM)
    c. Your Non-DMA optimized version using -O3 (BBB and FRDM)
    d. Your DMA Version (FRDM)

General Requirements

- Overlap needs to be handled correctly in your DMA transfers
- Provide a table of the execution times of the DMA, non-DMA transfers, and standard library transfer times. To get time on the BBB use the built-in time function to get time on the BBB. See "sys/time.h" and the function gettimeofday().
    - Be sure to use the right units in your table. Is it seconds you are measuring or clock cycles. This is important when comparing the different architectures.
    - There is a library that you can use to track execution statistics already called time.h. In time.h there is a structure that you can put data into as well. Utilize this library. Documentation can be found here: https://www.tutorialspoint.com/c_standard_library/time_h.htm
- Include your results in a README.txt/md and **comment** about the different measured values. Which code is the most efficient? Does the efficiency change based on the transfer size?

Binary Logger

Printing general character sets to the screen is easy, things start to get more difficult when you want to start interpreting data values as a string of printable characters. However, this is not typically done with an embedded system, because of the processing time it takes to convert integer or fractional data to ASCII characters. It is generally better to use a scripting language to analyze raw binary output and produce a human readable form. For this part of the project, you will create a binary logger, using log IDs and a payload block to log important data to a serial output. The logger system should be a background task for your embedded system. However, to combine everything, interrupts, UART, buffers, and logger into one deliverable is a challenge. Take your circular buffer and UART code from previous assignments and this project, and combine them to create a binary logging interface. This interface should have a reasonably-sized *logger buffer* to hold the log data. Functionality should be documented appropriately in the write-up.

The log interface can be used for live debugging with logs or for general I/O. However, logging can cause a lot of processing overhead. Extend your compile-time switches to add the LOG functionality so this feature may be turned on or off at compile time. That is, provide some type of preprocessor directive to

Enable/Disable logging functionality. Call this directive DEBUG, VERBOSE, or DPRINT.

Blocking Log

First create simple log functions to will wrap your UART transmit functions. These log function should take a string of characters and a length parameter. Also write an integer log that uses itoa() to print numbers. These first functions can use the UART as a blocking version (no interrupts). The logger should use the appropriate output functions for the target. For the FRDM, the UART is used. For the BBB and Host, simply use the printf() function.

The functions that need to written are:

- log_data - Takes a pointer to a sequence of bytes and length of bytes to log
- log_string - Takes a C-string and logs that to the terminal
- log_integer - Takes an integer and logs that to the terminal (use itoa())
- log_flush - Blocks until the current logger buffer is empty

These logs need to work across platforms so you need to create a wrapper macro that can translate these functions to the different platforms. An example of how these should look is shown below:

```
LOG_RAW_DATA(<type> * data, <type> len)
LOG_RAW_STRING(<type> string)
LOG_RAW_INT(<type> int)
LOG_FLUSH()
```

These logs will be a stepping stone as you develop the binary logger.

Non-Blocking Logs

Next expand your logger system to track very specific events in your embedded system. However, these logs will run in the background using interrupts in a non-blocking fashion. This system will use a log queue. The log queue requires a custom-designed data structure. This data will be transferred over your binary logger system. Each log structure will need to contain the following items:

- Log ID - Indicator on what the log is
- Timestamp (32-bit RTC value)
- Log Length - Number of bytes of Payload
- Payload - Any associated data that is sent with the log (can vary in size). Dependent on the log ID.
- Checksum - Validate the received packet with a checksum method. You can choose adding, XOR, 1's counter, or some other method for generating the checksum.

These event logs should use the following macro:

```
LOG_ITEM(<logtype> * data, <circbufftype> * buffer)
```

Define a custom logger enum to track the events you are logging. These events will have a unique IDs for each event type. IDs that you need to track are:

- LOGGER_INITIALZED - No Payload
- GPIO_INITIALZED - No Payload
- SYSTEM_INITIALIZED - No Payload

- SYSTEM_HALTED - No Payload
- INFO - Sends important information with regards to an info string
- WARNING - Sends important information with regards to an info string
- ERROR - Sends important information with regards to an info string
- PROFILING_STARTED - Profiling analysis has started
- PROFILING_RESULT - Logs a function identifier and a count for how long it took
- PROFILING_COMPLETED - Profiling analysis has completed
- DATA_RECEIVED - Logs that an item was received on the UART, transmits it back
- DATA_ANALYSIS_STARTED - No Payload
- DATA_ALPHA_COUNT - Logs number of alphabetic characters
- DATA_NUMERIC_COUNT - Logs number of alphabetic characters
- DATA_PUNCTUATION_COUNT - Logs number of alphabetic characters
- DATA_MISC_COUNT - Logs number of alphabetic characters
- DATA_ANALYSIS_COMPLETED - No Payload

You will need to add at least one logger function to help with the logging process.

- log_item - Logs a log item to the logger queue. Takes a log structure pointer and a length of the log.

Incorporate the system-level logs listed above into your program at the proper points. For example, the circular buffer used to log should be setup first, then the logger. Once the UART logger is configured, you should log the LOGGER_INITIALZED. Once all initialization is done, you should log the SYSTEM_INITIALIZED.

The profiling logs need to be incorporated into your profiling code. Results for each profiled function should be printed. Character processing for input data from project 2 will need to be converted to use these logging mechanisms.

One of the fields you need to add is the timestamp. Generate a timestamp from the Real Time Clock (RTC) module. In order for your timestamps to make sense, you will need to calibrate the start time of the RTC module. Without some type of external source for this, you will not be very exact and can drift with time. Instead, choose a logical start time and set that as a starting point for the RTC. (See Extra credit)

The RTC module needs to be configured with an input clock source. This can be done a number of ways. There are 3 options for input clock sources:

- System Oscillator when it is configured for 32 KHz
- RTC_CLKIN pin
- The LPO

The RTC_CLKIN operation requires you to output a clock source on a pin and route it as an input into the RTC_CLKIN pin.

The timestamp can be read from the RTC_SECS Register. As with other peripherals, for this peripheral to work, it needs to be enabled. RTC interrupts also need to be enabled. On the BeagleBone Black, you can use the time library to grab the current epoch time. With this, we will log a heartbeat log message to the console. To your log enums, add a new enum called **HEARTBEAT**.

Finally, add a feature to your logger to enable the FRDM to receive commands from the terminal that turns logging on/off. (This is not the same as adding a compile-time switch to add logging functionality. This is a runtime, interactive feature.)

General Requirements

- Write functions that can transmit characters for data logged through the UART.
- You must design/add-in a few implementation and header files for this:
  - logger.c/.h: Holds wrapper functions for the UART module to print data to the terminal
  - logger_queue.c/.h: Holds wrapper functions for the modified circular buffer module
- [BBB] Utilize the time library on the BeagleBone Black to get the most accurate timestamps (time.h)
- [FRDM] RTC Interrupt Heartbeat Log needs to be send at every RTC interrupt
- [BBB and FRDM] Add a new Log item called **HEARTBEAT** to your logger system

Circular Buffer Enhancements

Create a new circular buffer to work on log items. Each element should be a log item. This should allow the UART to handle transmitting the log automatically without trying to block to send. However, we need to make some other modifications to this so that the buffer is fast and secure. Call this new circular buffer Logger_q.

You will need to make is to make circular buffer operations atomic. Do this by making use of critical sections. For this you will need to enable and disable global interrupts with the **__enable_irq()** and **__disable_irq()** CMSIS functions. Define a two new macro functions that correctly enable/disable interrupts:

```
#define START_CRITICAL()
#define END_CRITICAL()
```

You do not need to support these on the BeagleBone Black. Only on the FRDM do you need to support this (for now).

Next you are going to define some buffer functions as inline. Modify the following functions to be inline functions:

- Checking for Buffer Full
- Checking for Buffer Empty

You will need to change your function prototype for the buffer functions to utilize a mixture of inline, attributes and static keywords. An example is below:

__attribute__((always_inline)) static inline **<normal-function-prototype-here>(<Params>);**

If you use static and inline, you need to define your code in a header file so that the code is compiled directly into the caller. Without the static keyword, the source code will be compiled as usual. The inline keyword is ignored when optimizations are not enabled. Adding the attribute forces the compiler to inline code.

One thing you will need to take into account is that the inline keyword is not supported on C89 and the __attribute__ keyword is not supported on all GCC compilers. To deal with this situation, use the following definitions provided by the CMSIS software:

__STATIC_INLINE

__INLINE

General Requirements

- [FRDM] Define the critical section macros for the KL25z to make the operations atomic. These can be disabled for the BBB.
- [BBB and FRDM] Change the buffer functions to be inlined
- [BBB and FRDM] A Logger Circular Buffer needs to be made to add/remove log items.

SPI Driver

For this part of the lab, you will configure the Serial Peripheral interface (SPI) for the Freedom Freescale KL25Z processor. This interface will be used to interact with the Nordic nRF24L01+ module that was mentioned in the course syllabus class materials list. The goal of this part is to create a driver that can read and write to the Nordic module, as well as send some simple commands. You will not need to get wireless data transmission working.

This section will require you to design your code in multiple layers similar to the UART project. These layers will involve the following:

- Firmware-level interface
  - Create a SPI library that will configure the peripheral, allow sending and receiving data.
  - Create a GPIO library that allows you to configure the ports as well as the chip selects and chip enable pins for the Nordic device.
- Driver LIbrary
  - Create a high-level library that uses the lower-level SPI library and GPIO to interact with the Nordic Chipset.

The GPIO library (gpio.c/gpio.h) will need to include the following functions:

- GPIO_nrf_init()
  - This should initialize the GPIO pins associated with the NRF and SPI devices

Your SPI library can be written to be a blocking interface for now since it is easier to debug this way. The SPI library (spi.c/spi.h) will need to include the following functions:

- `void SPI_init();`
  - Initializes the SPI controller
- `void SPI_read_byte(uint8_t byte);`
  - Reads a single byte from the SPI bus
- `void SPI_write_byte(uint8_t byte);`
  - Sends a single byte on the SPI bus

- `void SPI_send_packet(uint8_* p, size_t length);`
    - Sends numerous SPI Bytes given a pointer to a byte array and a length of how many bytes to send.
- `void SPI_flush();`
    - Blocks until SPI transmit buffer has completed transmitting

The nordic library (nordic.c/nordic.h) will need to include the following function definitions in order to interact with some of the registers on the device. nrf_read_register and nrf_write_register should be used inside all of the specific read/write functions.

- `uint8_t nrf_read_register(uint8_t register);`
    - Read the register and return the value
- `void nrf_write_register(uint8_t register, uint8_t value);`
    - Write to the given register with the data.
- `uint8_t nrf_read_status();`
    - Reads the STATUS register
- `void nrf_write_config(uint8_t config);`
    - Write to CONFIG register
- `uint8_t nrf_read_config();`
    - Read the CONFIG register
- `uint8_t nrf_read_rf_setup();`
    - Reads RF_SETUP register
- `void nrf_write_rf_setup(uint8_t config);`
    - Writes to the RF_SETUP register
- `uint8_t nrf_read_rf_ch();`
    - Reads RF_CH register
- `void nrf_write_rf_ch(uint8_t channel);`
    - Writes to the RF_CH register
- `void nrf_read_TX_ADDR(uint8_t * address);`
    - Reads the 5 bytes of the TX_ADDR register
- `void nrf_write_TX_ADDR(uint8_t * tx_addr);`
    - Writes the 5 byte TX_ADDR register
- `uint8_t nrf_read_fifo_status();`
    - Reads FIFO_STATUS register
- `void nrf_flush_tx_fifo();`
    - Sends the command FLUSH_TX
- `void nrf_flush_rx_fifo();`
    - Sends the command FLUSH_RX

When interacting with the Nordic chip you will have to enable the chip select connection. This should be enabled at the start of any packet transaction, and disabled at the end. Create the following functions to be as short as possible with the use of inlines or macros. These will need to interact with the GPIO library.

- `nrf_chip_enable()` -> Inline or macro function
- `nrf_chip_disable()` -> Inline or macro function
- `nrf_transmit_enable()` -> Inline or macro function
- `nrf_transmit_disable()` -> Inline or macro function

The Nordic library should also define many of the constants that are needed for the NRF device. This should include all of the registers in the register map, all of the commands, and any configuration values for bitwise manipulation for the associated registers you are to read/write. This means all of the bitwise data for the following registers:

- STATUS
- CONFIG
- RF_SETUP
- RF_CH
- FIFO_STATUS

The NRF data sheet will provide all of the interface information. Some example registers and bit masks are provided:

```
#define NRF_STATUS_REG     (0x00)
#define NRF_TXADDR_REG     (0x10)
#define NRF_POWER_UP       (1)
#define NRF_POWER_DOWN     (0)
#define NRF_POWER_UP_MASK  (0x02)
```

There are many open source examples of this code that you can look at as an example. However, most of the online examples loop/poll or bit-bang data to the Nordic device and don't actually use the onboard SPI peripheral interface. At first pass of this design, it is suggested that you design your interface with SPI to block until read/write has completed. This will help simplify the testing of your code for this device.

One final note is that you must be careful with your SPI configuration.

1. The SPI clock frequencies you select needs to be no larger than 8MHz.
2. The KL25z SPI will need to be set into Master mode and the NRF will act as a slave device.
3. Your SPI module should be configured for 3-wire mode (+2 gpio wires). (That is, use a gpio pin for the chip select instead of the built-in CS pin.)
4. The device should be configured for MSB first.
5. Clock should Idle Low and be active on the rising edge.
6. Your clock Chip selects should wrap around the full length of the nordic commands.
7. The SPI NOP command for this device is 0xFF

General Requirements

- You cannot use a library for the nRF24L01 module. You must write everything.
- The SPI configuration must be in 3-wire mode
- You must create implementation and header files for:
    - GPIO.c/.h: Holds all of your pin control and config functions
    - spi.c/.h: Holds all of your SPI initialization and setup code
    - nordic.c/.h: with all registers, constants, and other structure interface definitions in a file called. Should call your SPI libraries
- You must show that you can read and write data to/from the device in the report. This can be done by doing a write and then a read to confirm both work.
    - CONFIG register (read/write): Set this value and read to validate
    - STATUS register (read)

- TX_ADDR (read/write): Set this value and read to validate
- RF_SETUP register (read/write): Set this value and read to validate
- RF_chregister (read/write): Set this value and read to validate
- FIFO_STATUS register (Read)

main.c

Your main function will be very simple. You will just need to call a function that is defined in the project3.c source file. As with Project 2, wrap your call to project3() in your main() function using a **-DPROJECT3** compile time switch.

```
#ifdef PROJECT3
     project3();
#endif
```

project3.c/project3.h

Your project3() function needs to do numerous things:

- Run the profiling code
- Run a SPI Demo sequence of commands to show you can read and write to the nRF device

Feel free to organize this how you like. It is suggest that you separate the Profiling and the SPI Nordic demo operations into different code.

The output executable that gets built needs to be called **project3.elf.** This needs to be executed using dot-slash notation:

$ ./project3.elf

Documentation

Make sure your functions and files are documented. Recall the Doxygen system from Project 2.

Each function should have the following:

- @brief - short description
- Long description
- @param - if applicable
- @return

Each file needs to have the following documentation:

- @file - name of file
- @brief - short description
- long description
- @author
- @date

Additionally, you will have to document your software architecture. Create a diagram of your choice and commit it to your repository so we can see how your system connects together. Indicate what the modules are.

Extra Credit

Connect UART to DMA (5 Pts)

Create a new version of your UART system. For this system integrate your UART code to your DMA controlled circular buffer. There will be some timing and management issues when doing this. You will have to determine how the UART routines and ISR interact with the DMA ISR via the circular buffers.

Accurate Time Stamps (1pts)

Normally something will sync to a time or you start by compiling in a build time as a compile time define. This then allows the image to be created with a logical start time. There are a few ways to obtain this, but here is an example with the Linux date command. You would just need to pass this in as a compile time define.

```
$ date +%s
```

By compiling this into your build and setting the RTC time to this value as soon as you start up, you should have accurate timestamps. There will be some amount of error, due to build time, install, and startup, but we will ignore this.

Logger Script (5 pts)

Create a Python script to decode your binary log data and print more meaningful messages to the screen.

Wireless Communication (20pts)

Once configuration of the Nordic module is working and you can read and write registers try to send data wirelessly from one device to the other (BeagleBone to KL25z). A command not discussed yet with the Nordic transceiver is its actual wireless data transmission functionality. The chip has an on-chip transmit/Receive FIFO that is used to actually send wireless data. The goal here is to be able to send data from your BeagleBone to your MKL25z processor wirelessly. This will require you to interact with the Transceiver TX/RX state machine described in their data sheet.

The BeagleBone does not require low-level Firmware however, you need to figure out how to use the SPI device drivers to get it to interact with the Nordic chip. Because of this, using SPI on the BeagleBone is a lot easier than on the MKL25z processor as you will not have to debug any firmware. SPI driver already exists on the device. There are some specific steps you will need to do to enable the SPI interface and disable the HDMI interface.

- BeagleBone SPI Enable HDMI Disable Help
  - http://embedded-basics.blogspot.com/2014/10/enabling-spi0-on-beaglebone-black.html
  - http://electron14.com/?p=404

Your main task with the BeagleBone will be to reuse your Nordic library to wrap the BeagleBone SPI Driver to Read/Write register libraries for the Nordic chip. This new code should have the same functionality as the Freedom Freescale board did but you will fundamentally need to write wrapper code for the SPI driver libraries to maintain the same SPI library code.

Once you have made the modifications for the BeagleBone, both devices need to be configured for wireless transmission with a unique RX/TX address (this is one of the registers in the data sheet). You will need to configure a unique address for the two devices so that they can transmit.

Nordic SPI Library Requirements

- You cannot use processor expert
- You should configure one device to act solely as the Transmitter and the other device to act solely has the receiver.
- Use your Command Message interface to transmit your commands between the two devices.

Project Demo Procedures

Your project will be graded during a demo session with the instructor team. The following items will be asked of your group for the project during this demo.

1. Students will be asked to run certain commands and observe the output. These commands can be tested on any of the platforms supported
2. Documentation on your source files and functions will need to be shown (just open them to view during the demo).
3. Successful running of your compiled files from project2() function without any errors.
4. The instructor team will ask you to add a new feature request during the demo.

Students should plan the demo practical to take 20-30 minutes per group. Students in groups will be asked to work on tasks in parallel. Campus students will be asked to meet in person with the instructor team. Distance students will use Screen share and a Zoom session to demo.