Principles of Embedded Software

*Project 2 (**125** + **5**(Extra credit)) - Due Wednesday 10/25*

Data Structures, Unit Tests & Microcontrollers

Overview

Description

In this project assignment, you will write a data processing application for a microcontroller. This application will utilize a circular buffer data structure, configure a microcontroller and the use of a few peripherals including: clock systems, General Purpose Input/Output (GPIO) and a Universal Asynchronous Receive Transmit (UART) module. Additionally, you will develop unit test code using the Cmocka unit test framework.

Outcomes

After completing this assignment, you will be able to:

1. Design a circular buffer data structure
2. Write firmware for a UART peripheral to transmit and receive data
3. Use interrupts to handle I/O requests to process an input byte sequence

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 will need to turn in a code dump report. This should be a single file in a **.pdf** format that will turn into the Project 2 Dropbox D2L, as you did with Project1. This allows us to check for plagiarism. No other formats will be accepted.

Resources:

- KL25 Sub-Family Technical Reference Manual
- Circular Buffer Example (https://en.wikipedia.org/wiki/Circular_buffer)
    - Making Embedded Systems (Elecia White) - Pgs 177 -183
- cmocka Unit Test Framework - https://cmocka.org/
- cmocka Unit Test Framework API - https://api.cmocka.org/
- Diagram Tool - https://www.draw.io/

Version Control and Platform Support

Continuing with the previous project, the code for this project is a modification and enhancement of the

previously written code and must be turned in via your git repository. You can 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 1, make sure to inform the TAs.

You will need to make a minimum of 5 commits for Project 2, and there need to be commits from both members. You may not 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.

You will need to place a git annotated tag with the name **project-2-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 will require you to use an IDE, specifically the Kinetis Design Studio IDE. For certain components in this project, you will need to support three target platforms. Here is a breakdown of where code should be able to run:

- Data Processing - KL25z & Host
- Circular Buffer - All Platforms
- Unit Tests - Host Platform
- DMA - KL25z
- UART - KL25z

Circular Buffer

A *circular buffer* is a data structure that contains two parts. A buffer where data is placed and a controlling structure to monitor the buffer area. To added or removed data from the circular buffer, interface functions need to be defined to interact with the control structure and the buffer memory. These interface functions should be buffer independent. Meaning, buffer pointers should be passed into each function so that all operations utilizing a buffer are encapsulated within a function.

Your circular buffer code must placed in aptly named files like circbuf.c and circbuf.h. The header file must contain all public interfaces to the circular buffer with documentation of inputs, outputs and description. The circular buffer should be defined to buffer **unsigned byte** items. The size of the buffer should be given at allocation and both the buffer structure and buffer itself must be allocated on the

heap. Lastly, the circular buffer must be architecture independent (host, BBB, or FRDM). All code **must** use standard data type sizes (int8_t, uint32_t, etc)

The C-programming functions listed in this section must be written on your own. Many of these functions will be used in the next project. No online code may be used. All code must be written on your own. Any plagiarized code will result in an honor code violation and a 0% for both group members.

Data Requirements

Define both an enumeration and a structure for this module. This structure and the enumeration should be type defined as the type **CB_t** and **CB_status**, as shown below:

```
typedef struct {
    …
} CB_t;

typedef enum {
    …
} CB_status;
```

Your structure must be allocated dynamically and track the following members:

- Buffer Pointer
  - Allocated memory for the buffer elements
  - Buffer area must be created dynamically
- Head
  - Pointer to Head or newest item
  - This should match the item type
- Tail
  - Pointer to Tail or oldest item
  - This should match the item type
- Length/Size
  - Number of items allocated to the buffer
  - This should be set when buffer is allocated
- Count
  - Current item count in the buffer
  - Must be updated with every add/remove

The enumeration needs to support a handful of status conditions. These conditions should be returned at the appropriate points in the buffer operations. These include the following enumerators (but are not limited to these):

- Buffer Full
- Buffer Empty
- Success / No Error
- Null Error

Function Requirements

The following functions should be written for your circular buffer module:

<CB enum type> CB_buffer_add_item(<buffer to add to>, <data to add>);

1. *The function will take 2 arguments:*
    a. *Pointer to the circular buffer to which the data item is to be added*
    b. *The data to be added to the circular buffer*
2. *The function returns an enumeration that specifies the success, failure etc. of the function call*
3. *Must take into consideration the corner cases possible*

<CB enum type> CB_buffer_remove_item(<buffer to remove from>, <variable to store data removed>);

1. *The function will take 2 arguments:*
    a. *Pointer to the circular buffer from which the data item is to be removed*
    b. *Variable to store and return the removed item from the buffer*
2. *The function returns an enumeration that specifies the success, failure etc. of the function call*
3. *Must take into consideration the corner cases possible*

<CB enum type> CB_is_full(<buffer to check>);

1. *The function will take in the circular buffer to check if it is full*
2. *The function returns the status of the circular buffer or an error code*

<CB enum type> CB_is_empty(<buffer to check>);

1. *The function will take in the circular buffer to check if it is empty*
2. *The function returns the status of the circular buffer or an error codes*

<CB enum type> CB_peek(<buffer to peek into>, <position to peek>);

1. *The function will take 2 arguments:*
    a. *Pointer to the circular buffer to peek into*
    b. *The position from the head of the buffer to peek into*
2. *The function returns the status of the circular buffer or an error code*
3. *Must take into consideration the corner cases possible*

<CB enum type> CB_init(<pointer of buffer type>, <length of buffer>);

1. *The function will take 2 arguments:*
    a. *Pointer to the circular buffer*
    b. *The length/size to be reserved for the buffer*
2. *The function returns the status of the circular buffer or an error codes*

<CB enum type> CB_destroy(<pointer of buffer type>);

1. *The functions takes in the buffer to be destroyed*
2. *The function returns the status of the circular buffer or an error codes*

Each of these functions (as necessary) must take in a circular buffer pointer type (as necessary) and all actions for a circular buffer must be encapsulated in a circular buffer function and have a circular buffer enum for a return type. For example:

```
CB_e CB_AddItem(CB * buf, …);
```

Unit Tests with Cmocka

Unit tests are a type of test that is run on a module/function. We are going to write unit tests for much of the code developed in project 1, project 2 and the homework. However, these will not run on the target embedded system. Instead, these unit tests are used to test our higher-level application code in a simulated environment. The ***cmocka*** unit test framework is a convenient way to test our code . You will do both positive and negative testing.

A new build target needs to be created to build and run the unit tests. This target can be called **unittests**. This should build and run all the unit tests.

Write the following unit tests or make sure you have the following coverage for your code in your memory, conversion, and circular buffer functions. It is up to you to decide how to create the unit tests. A suggested approach is to split them up into multiple unit tests for each module.

memory.c

- memmove tests
    - Invalid Pointers - Should return fail if pointers are NULL
    - No Overlap - Should return a pass for a move
    - SRC in DST region Overlap - Should succeed at this
    - DST in SRC region Overlap - Should succeed at this
- memset
    - Invalid Pointers - Should return fail if pointers are NULL
    - Check Set - Should accurately set region for length Value
- memzero
    - Invalid Pointers - Should return fail if pointers are NULL
    - Check Set - Should accurately set region to zeroes
- reverse
    - Invalid Pointers - Should return fail if pointers are NULL
    - Check Odd reverse - Should check that reverse succeeded for odd length
    - Check Even reverse - Should check that reverse succeeded for even length
    - Check characters - Should be able to reverse any character set (256 byte array of 0-255)

conversion.c

- big_to_litte
    - Invalid Pointer
    - Valid Conversion - Test that a big-to-little conversion worked
- little_to_big
    - Invalid Pointer
    - Valid Conversion - Test that a little-to-big conversion worked

CircularBuffer.c

- Allocate-Free - Checks that a dynamic buffer can be created on the heap
- Invalid Pointer - Check that buffer pointer is valid
- Non-initialized Buffer - Check that buffer is initialized
- Add-Remove - Check that add and then a remove returns the same item for full length of buffer
- Buffer Full - Check buffer reports true for full

- Buffer Empty - Check buffer reports true for full
- Wrap Add - Test that your buffer can wrap around the edge boundary and add to the front
- Wrap Remove - Test that your buffer tail point can wrap around the edge boundary when a remove occurs at the boundary
- Overfill - Test that your buffer fails when too many items are added
- Over Empty - Test that your buffer fails to remove an item when empty

UART

You will be configuring the UART interface of the Freedom Freescale board so that you have a method of transmitting to and receiving characters from your board, without the use of the printf and the debug console. To do this, use the same USB connector as labeled for the debug interface. This connector has an internal UART to USB converter in the onboard OpenSDA Emulator (Kinetis K-Series K20DX... processor). After downloading code to the Freedom Freescale board, you will be using a terminal emulator (like Putty or Realterm) to interact with the board and your code.

The UART configuration is very flexible, but you should use appropriate settings. Below are some suggestions for the UART:

- **LSb** first (on by default)
- 1 Start/1 Stop Bits
- 8-bit data transfers
- No Parity (if you want to enable parity, you need to write software that can handle transmission failures and retry).
- The fastest BAUD you can achieve
    - 115200/38400/57200 Baud.

Selecting high baud rates will be better for speed but BAUD rate is dependent on the UART clock and BAUD prescaler registers. Sometimes getting a strange combination of BAUD and UART clock will not give you an exact BAUD clock rate, but rather produce some timing errors. This could cause issues with misreading characters. Look-up the standard BAUD rates. You may need to change your fundamental clock rate to make it easier to find a good BAUD rate.

All code for this module should be put in uart.c/*.h files. Your code **must be written in C** and not C++.

uart.c/.h

The settings for the BAUD value should not be hardcoded. **Create a Preprocessor Macro to auto calculate the baud settings given a BAUD**. Setting the correct values for the BAUD will depend on a number of factors including the peripheral clock, the intended BAUD rate and any prescalers you choose.

The UART functions to be written are outlined below:

<type> UART_configure ();

1. Configures UART to the given settings.

2. No hardcoded configurations may be used. All settings need to use predefined Bit Masks and macro functions to help determine calculated values

<type> UART_send (<data-to-send>);

1. This function will send a single byte down a specific UART device
2. The function will take 1 argument:
    a. Pointer to the data item to send
3. This function should block on transmitting data

<type> UART_send_n (<data-to-send>, <length-of-data>);

1. Function takes 2 arguments
    a. Pointer to a contiguous block of data that needs to be transmitted
    b. Number of items to transmit
2. This function should block on transmitting data

<type> UART_receive(<received-data>);

1. This function should return a received byte on the UART using an input parameter pointer.
2. This function should block until a character has been received.

<type> UART_receive_n (<received-data>, <length-of-data-to-receive>);

1. This function should receive a number of bytes on the UART.
2. Function takes 2 arguments
    a. Pointer to a memory location to place data that is being received
    b. Number of items to receive
3. This function should block until a number of characters have been received.

void UART0_IRQHandler();

1. This function is the IRQ handler for the UART.
2. You will need to handle two types of interrupts in this function
    a. Receive Interrupts
    b. Transmit interrupt
3. Each interrupt should clear their associated flag when completed but only if they were set
4. This routine should be as short as possible

Data Processing

This task relates to processing a string of characters coming in from the UART. To do this, take the circular buffer code and integrate it with your UART interface. Your code should track the number of characters that come into the interface as well as the number of different types of characters; such as alphabetic, numeric, etc. Output a small table summarizing statistics on the data you processed. It is up to you to decided what triggers the printing of the table; perhaps, when some number of characters has been processed, or perhaps if a particular character is processed. Functionality needs to be documented appropriately in your software via comments, header documentation and README support.

main.c

Your main function will be very simple. You will just need to call a function that is defined in the project2.c source file. However, you need to use a compile time switch to wrap this function to call. This way we can have a simple main() function that can switch which project deliverable we wish to call by specifying the **-DPROJECT2** compile time switch.

```
#ifdef PROJECT2
     project2();
#endif
```

project2.c/project2.h

The project2() function needs to process an input sequence of characters. As you receive these, you should be regularly updating your statistics on the types of characters you are receiving. You must do this while you are continuously receiving input data (the input data stream is not any fixed size or length). **You may not do this processing in the interrupt handler.** This processing must be done in the project2 function. The character counts you need to track include:

- Number of alphabetic characters
- Number of numeric characters
- Number of punctuation characters
- Number of Miscellaneous characters

This part should only be attempted when UART interface and the circular buffer functionality is correct. (In a more complex setup, you would have to make sure your UART is interrupt safe. This really means the introduction of critical sections and atomic operations. For this project it is sufficient to make your buffer data types volatile. We may discuss critical sections in a future lecture.)

We suggest that you integrate the modules in steps. Do not try and build the whole system in one step. First integrate interrupts with the circular buffer. In parallel, write the code to process input strings using statically allocated C-strings, and in addition, writing a statistic table using your uart functions and **itoa()** functions to print information. To test your processing code on your host machine you can utilize input functions like **gets** or **scanf** to get input sequences from the user.

The RX interrupt should either signal project2() to start analysis or Full RX Buffer should start the analysis once it has filled. However, project2() can start analysis earlier than the full received data, but it should not report any information until characters have been received. The RX and TX interrupt code should as short as possible.

When statistics are ready to print, you can use a dump_statistics() function to call itoa() and uart functions to send output to the terminal. Use TX interrupts to avoid blocking on sending data when writing data out. To do this, you can trigger a UART Interrupt point. Once data has been put into the transmit circular buffer, there should be a flag to trigger TX interrupts.

Receiving data will require the use of RX UART interrupts. However, you will need to store each received UART character into some larger data structure for processing like an RX circular buffer, a

second one (See previous parts of this project). We will use the RX side of the UART to type in a sequence of ascii characters and process in our main loop.

The output executable that gets built needs to be called **project2.elf.** This can be executed using dot-slash notation:

```
$ ./project2.elf
```

Documentation

Make sure your functions and files are documented. Recall the Doxygen system from Project 1.

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 the pieces of your system connect together. Indicate what the modules are.

Extra Credit

Remote Install and Debug from the Command line (5 Pts)

Add support into your build system to build the project sources and install them straight from the command line without the IDE. This can be in the form of linux bash scripts that are version controlled and the GNU Debugger (gdb).

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.