Introduction

For our sixth assignment, we will continue working on the FRDM-KL25Z. This assignment will focus on developing a full command interpreter running over UART0, using interrupts and circular buffers. You should be able to base part of your solution on code you developed originally in Assignments 1 and 2, although you will probably need to make modifications.

For this assignment we are going to cast aside virtually all code from the SDK – we will rely only on `MKL25Z4.h` and `startup_mkl25z4.c`. In place of the initialization code typically provided by the SDK, I have provided a simple routine to configure the system clock and the MCG module for use.

A template to start your project is provided as a ZIP file. Instructions for importing this ZIP file into MCUXpresso may be found later in this document.

As before, you will develop a proper GitHub project for this assignment, complete with code, documentation, and README. Details are given below.

Assignment Background

As a way to allow your company's development and QA teams to more fully exercise a new board under development, you have been given the task of setting up an initial command processor running over the serial line. Your manager, a lifelong fan of sugar in the morning, has given this project the code name **BreakfastSerial**, since it will be the first thing done on this board.

You will need to develop the following:

1) **A circular buffer implementation**. You will need two circular buffer objects, one each for the transmit and receive directions. These may be allocated statically (the preferred approach), or they may be allocated dynamically at initialization time[1]. Each circular buffer should have a capacity of 256 bytes.

2) **Test code to exercise your circular buffer**. You may wish to adapt the automated tests you created for Assignment 2. This code should run at startup if the DEBUG define is set in your code, in order to give you confidence that your circular buffers are solid.

[1] You may *not* call malloc after your code has completed initialization.

3) **Code to configure UART0 and send and receive data over it**. Parameters for this assignment are specified below. Your implementation should be fully interrupt based[2]. The UART solution should be built atop your circular buffer implementation.

4) **Glue code that ties your UART communication code into the standard C library functions**. After this glue code is working, a call to `printf()` or `putchar()` on the device should result in characters being sent over UART0 to the PC, and a call to `getchar()` should result in reading a character that the user typed on the PC.

5) **A command processor** that can accept some very simple interactive commands (specified below) and take action on the device.

Interactive Commands

The command processor running on your device should start by printing a prompt ("? "). It should then wait for a command typed by the user, and then respond to that command.

As each character is received by your command processor loop, you will need to echo that character back so that the user can see it printed in his or her terminal window. You do not need to handle the backspace key or any arrow keys.

Each user command should be terminated by a newline.

Command	Arguments	Description
Author	none	Prints a string with your name.
Dump	Start, Len	Prints a hexdump of the memory requested, with up to 16 bytes per line of output. See below for the required output, which is very similar to that used in Assignment 1. Start should always be interpreted in hexadecimal. Len should be interpreted in decimal, unless it begins with the characters 0x, in which case it should be interpreted in hexadecimal. Valid values for Len range from 0 to 640 decimal, inclusive. It will be possible for a user of the program to crash the program, by providing an illegal address for Start. Your code does not need to catch this error.

All commands and arguments should be processed in a case-insensitive manner, including any hex characters that the user enters. Additionally, the command processor should be forgiving about whitespace; the user should be able to have leading or trailing whitespace around the command, and

[2] You may wish to implement a poll-based approach at first as you are developing the code, because it is simpler. You should remove the poll-based approach once you get the interrupt method working.

multiple consecutive whitespace characters should be treated as one for the purposes of separating tokens.

If the command processor receives any command it doesn't understand, it should print the string "Unknown command: (command as received)".

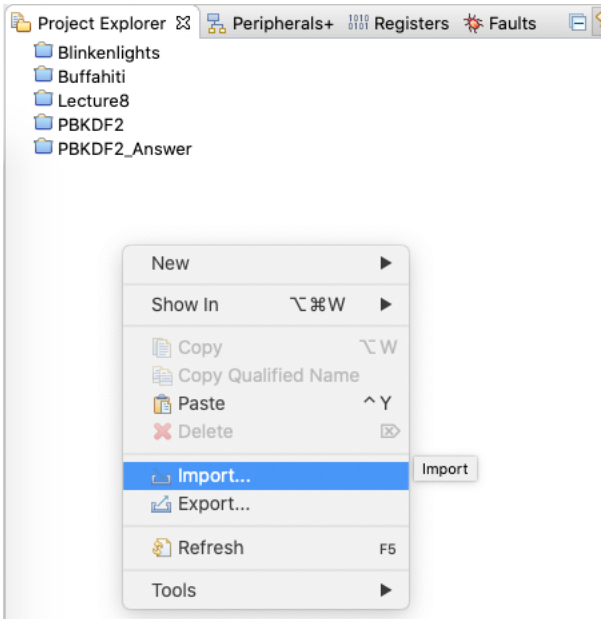
Here is an example of a valid command session:

```
Welcome to BreakfastSerial!
? author
Howdy Pierce
? Author
Howdy Pierce
? DUMP 0 64
0000_0000  00 30 00 20 D5 00 00 00 43 01 00 00 31 27 00 00
0000_0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000_0020  00 00 00 00 00 00 00 00 00 00 00 00 47 01 00 00
0000_0030  00 00 00 00 00 00 00 00 49 01 00 00 4B 01 00 00
? dump a0 0x20
0000_00A0  0F 02 00 00 17 02 00 00 1F 02 00 00 27 02 00 00
0000_00B0  2F 02 00 00 37 02 00 00 3F 02 00 00 47 02 00 00
? print
Unknown command: print
?
```

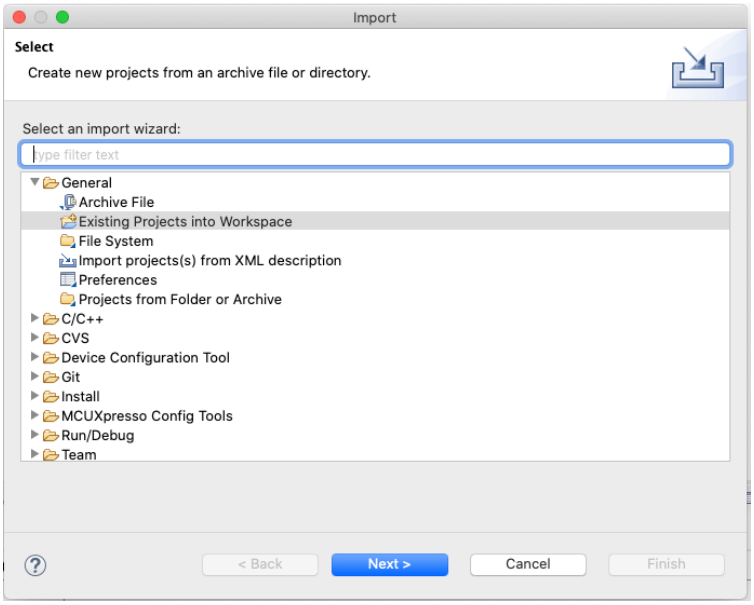
Importing the Source into MCUXpresso

You should be able to follow these steps to import the provided ZIP file into MCUXpresso:

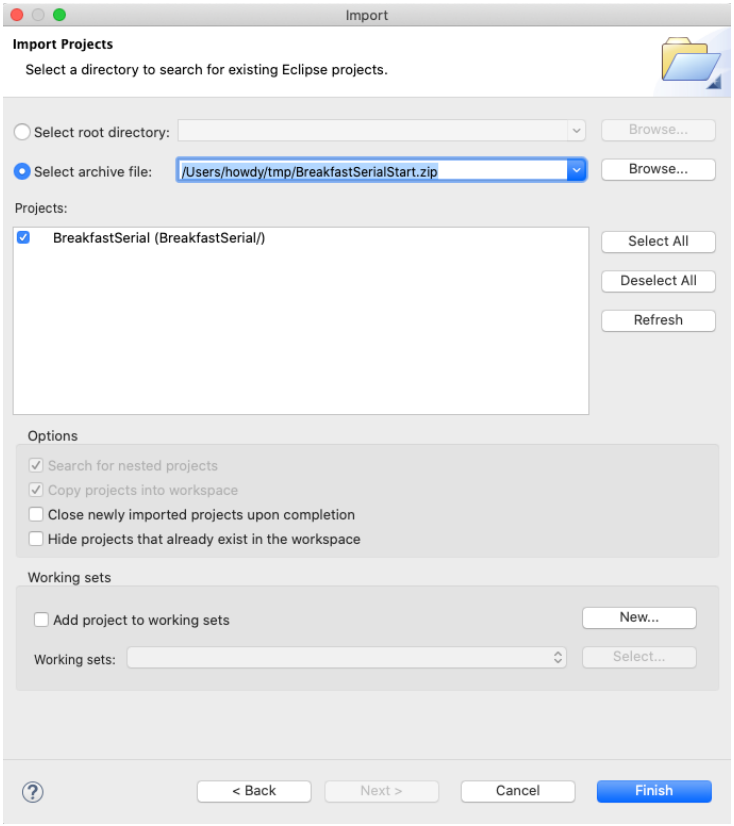
- In the Project Explorer area, right click and select "Import…"



- Select General > Existing Projects into Workspace, then click Next



- Click “Select archive file:”, then click the Browse button, and select the ZIP file



- Click Finish

Implementation Details

For the circular buffer, you may wish to reuse both your Assignment 2 implementation and your Assignment 2 test code, although you will need to enhance this solution in several ways: You will need to change the capacity; you will need to support multiple FIFO instances; and you will need to protect critical sections, since we will be using the circular buffers from within interrupt handlers.

Likewise, you may wish to reuse the `hexdump()` function you wrote in Assignment 1 for the command processing code. You will also need to modify it slightly because the addresses printed in the leftmost column should be absolute addresses, not relative to the start of the region being dumped, and they must be 8 characters long since we have a 32-bit address space. See the example output above.

The sscanf function might be useful to parse out the arguments for the “dump <start> <len>” command.

In order to make the code a bit different from that given as an example in Dean Ch8, we will use the following serial parameters:

Baud rate	38400
Data size	8
Parity	None
Stop Bits	2

You should also be aware that the system clock initialization code I have provided sets different parameters than what Dean’s solution is based upon.

All four of the above serial parameters should be defined in a single place (probably at the top of UART.c or similar) using well-commented #defines.

Assuming you are linking with Redlib (and you will be if you imported my project), you can define two functions in order to tie your UART code in with the standard system I/O functions such as printf(), getchar(), and so forth:

```
int __sys_write(int handle, char *buf, int size);
```

> Writes the specified bytes to either stdout (handle = 1) or stderr (handle = 2). For our purposes you may ignore the handle argument, since both are tied to the serial output. Returns -1 on error or 0 on success.

```
int __sys_readc(void);
```

> Reads one character from the serial connection and returns it. Returns -1 if no data is available to be read.

In previous assignments, you have created DEBUG and RUN build targets with slightly different behavior. Because BreakfastSerial is intended only for internal use during development, for this assignment you only need the DEBUG build target.

Submission Details

Your primary submission is due on Tuesday, November 9, at 10:30 am. At that time, you should submit the following to Canvas:

- A private GitHub repo URL which will be accessed by SAs to review your code and documentation. For this assignment, this repo should have an MCUXpresso project containing multiple .c and .h files. **Please check the MCUXpresso tree directly into GitHub**—do not zip it or extract individual files from it.
- Two screenshots: The first showing the terminal parameters you use to connect, and the second showing your interactive terminal session, similar to the excerpt above. You should run the following five commands:
    - author (all lower case)
    - Author (initial capital)
    - DUMP 0 64 (note, decimal second argument)
    - dump a0 0x20 (hex second argument)
    - print (show handling of unknown command)

  (If you do the extra credit, also show the info and help output.)
- A PDF containing all C code. As before, the PDF is used specifically for plagiarism checks.

Grading

Points will be awarded as follows:

Points	Item
1	Did you submit your GitHub URL in your Canvas submission?
1	Did you submit both screenshots to Canvas as requested?
28	Overall elegance: Is your code professional, well documented, easy to follow, efficient, and elegant? Is each module (UART, hexdump, command processor) located in its own file, with an appropriate .h file to expose the interface?
30	Does the UART code allow text to be output to the terminal window reliably, using 38400 baud rate, 8 data bits, no parity, and 2 stop bits? Are these values easily changed by altering well-documented #defines in your code? Is this code interrupt-based and tied into printf correctly?
30	Does your command processor work as specified? Are prompts printed correctly (with no missing or extra CRs or LFs)? Are characters echoed back to the user correctly? Can the commands be entered in either case? Does the “dump” command work correctly?
10	Is it trivially easy to add a new command to your command processor?
10	Extra credit as described below

Good luck and happy coding!

Extra Credit

[5 points] Extend your command processing loop so that it correctly handles the user hitting the backspace key.

[5 points] Extend the command processor to add the following commands:

Command	Arguments	Description
Info	none	Prints a string with build information that is dynamically generated at build time from your machine. Output should be as follows: Version 1.0 built on Calvin at 2020-10-23_05:32:12 Commit cf58e68fa52fb6e7258998aed343af3477746079 …where the version number (1.0), the machine name (Calvin), the build date/time (2020-10-23_05:32:12), and the commit ID ("cf58e6…") are all dynamically generated at build time.
Help	none	Prints a help message with info about all of the supported commands.

Information about dynamically generating build information may be found here[3]. The following shell commands may be useful for generating that output:

- `git log --pretty=format:"%H" HEAD -1` will output the ID for the most recent commit.
- `hostname` (at least on Mac/Linux) will output the machine's name
- The `date` command can be used to output the date/time stamp (with appropriate arguments), or you can use a Python one-liner.

[3] https://embeddedartistry.com/blog/2019/02/18/using-custom-build-steps-with-eclipse-auto-generated-makefiles/