CMPE-250
Assembly and Embedded Programming
Spring 2021
Laboratory Exercise Nine
Serial I/O Driver

This exercise develops a serial I/O driver for the KL05Z board. The objective of this exercise is to implement interrupt-based serial communication with the KL05 UART using circular FIFO queues for receiving and transmitting serial data. An interrupt service routine (ISR) and serial character I/O driver are written and integrated with the circular FIFO queue operations test program from Lab Exercise Seven, and they are run on the KL05Z board.

Prelab Work

1. Based on the specifications below and the notes you have taken in lecture, rewrite the following subroutines from prior lab exercises to work with receive and transmit queues rather than the UART data registers. You must write the subroutines so that no register other than output parameter register(s) and PSR has changed value after return.
   - GetChar: Dequeues a character from the receive queue, and returns it in R0.
   - PutChar: Enqueues the character from R0 to the transmit queue.
2. Following the specifications in this exercise and the notes you have taken in lecture, write the ISR that handles UART0 transmit and receive interrupts: UART0_ISR. You must write the ISR so that no registers have changed value after return. (Note: the Cortex-M0+ automatically preserves R0–R3, R12, LR, PC, and PSR for ISRs.)
3. Write the change required in the KL05 vector table to "install" UART0_ISR.
4. Write the subroutine Init_UART0_IRQ to initialize the KL05 as discussed in class and presented in the class notes for interrupt-based serial I/O with UART0 through port A pins 1 and 2 using this format: one start bit, eight data bits, no parity, and one stop bit at 9600 baud—the same format and speed as previous lab exercises but with interrupts instead of polling. This subroutine should configure UART0, should initialize the UART0 interrupt in the NVIC, and should initialize the receive and transmit queue management record structures (RxQRecord and TxQRecord) for 80-character queue buffers (RxQBuffer and TxQBuffer, respectively) using InitQueue from Lab Exercise Seven. You must write the subroutine so that no registers have changed value after return. (Suggestion: copy Init_UART0_Polling that has been used since Lab Exercise Five, and modify it to use interrupts instead of polling, including changing UART0 initialization to support UART0_ISR and calling InitQueue from Lab Exercise Seven to initialize the receive and transmit queue management record structures.)

Application

This assignment implements and tests the serial I/O driver with an application adapted from Lab Exercise Seven that performs enqueue and dequeue operations on a single circular FIFO queue and prints queue state information using the following commands.

Command	Description
D or d	Dequeue a character from the queue.
E or e	Enqueue a character to the queue. (Prompt to enter a character, and then enqueue it.)
H or h	Help: list the commands.
P or p	Print the queued characters from the queue buffer to the terminal screen, in order from first in to last in.
S or s	Status: print the queue's current *InPointer*, *OutPointer*, and *NumberEnqueued*.

Program Specification

Adapt the queue program from Lab Exercise Seven to test your serial I/O driver from prelab work. The program must perform the following sequence of operations and must meet the requirements that follow. Note that this functionality is identical to that of the program written for Lab Exercise Seven, except for specification 1. (Caution: with mixed byte and word data in multiple queue record structures, you will need to place an `ALIGN` directive before *each* queue record structure allocation since each one will be used for word memory access.)

1. Use Init_UART0_IRQ to initialize the KL05 for interrupt-based serial I/O through receive and transmit queues with UART0 at 9600 baud using a format of one start bit, eight data bits, no parity, and one stop bit—the same format and speed as previous lab exercises but with interrupts instead of polling.
2. Use InitQueue to initialize the program queue record structure (QRecord) for a queue buffer (QBuffer) of 4 characters, (i.e., bytes).
3. Output the character string below to the terminal.
```
Type a queue command (D,E,H,P,S):
```
4. Read a character typed on the terminal keyboard.
5. If the character is a lowercase alphabetic character, save the original character for possible output in step 7, and convert a copy to its uppercase equivalent for the command. For example, if the user typed d, save it and change a copy to `D`. (Having only one case of command characters requires fewer lines of code in the next step.)
6. If the character typed is not a valid command, (i.e., `D`, d, `E`, e, `H`, h, `P`, p, `S`, or `s`), ignore the character and repeat from step 4.
7. Otherwise, if the character is a valid command, print the character from step 4 to the terminal screen.
```
Type a queue command (D,E,H,P,S):p
```

8. Next, output a carriage return character (0x0D) and a line feed character (0x0A) to move the cursor to the next line of the screen, and perform the command, as follows.
   - `D` or `d`: Use `Dequeue` to attempt to dequeue a character from the queue. If the queue was not empty, print the dequeued character followed by a colon, as shown below where "A" was dequeued.
     ```
     Type a queue command (D,E,H,P,S):d
     A:
     ```
     Otherwise, print "Failure:" if the queue was empty, as shown below
     ```
     Type a queue command (D,E,H,P,S):d
     Failure:
     ```
     In either case continue with step 9.
   - `E` or `e`: Prompt the user to enter a character to enqueue, as shown below.
     ```
     Type a queue command (D,E,H,P,S):e
     Character to enqueue:
     ```
     Read a character typed on the terminal keyboard, and then print it to the screen followed by a carriage return character (0x0D) and a line feed character (0x0A) to move the cursor to the next line of the screen, as shown below for a typed "A."
     ```
     Type a queue command (D,E,H,P,S):e
     Character to enqueue:A

     ```
     Next, use `Enqueue` to attempt to enqueue the typed character to the queue. If the queue was not full, print "Success:" as shown below.
     ```
     Type a queue command (D,E,H,P,S):e
     Character to enqueue:A
     Success:
     ```
     Otherwise, print "Failure:" if the queue was full, as shown below.
     ```
     Type a queue command (D,E,H,P,S):e
     Character to enqueue:A
     Failure:
     ```
     In either case continue with step 9.
   - `H` or `h`: Print a help string that lists the various valid commands. An example is shown below.
     ```
     Type a queue command (D,E,H,P,S):h
     D (dequeue), E (enqueue), H (help), P (print), S (status)
     ```
     Next, output a carriage return character (0x0D) and a line feed character (0x0A) to move the cursor to the next line of the screen, and repeat from step 3.
   - `P` or `p`: First print ">" as a beginning delimiter, and then print all the actively queued characters in the queue. The characters should be printed in order from the first queued to the last queued—the same order as they would be removed from the queue, but *without removing them from the queue*. If the queue is empty, no characters should print. Then print "<" as an ending delimiter, followed by a carriage return character (0x0D) and a line feed character (0x0A) to move the cursor to the next line of the screen, and repeat from step 3. The example output below shows the result for only one character "A" in the queue.
     ```
     Type a queue command (D,E,H,P,S):p
     >A<

     ```
   - `S` or `s`: Print "Status:" and then continue with step 9.

9. Report the status of the queue:
   a. Print "`In=0x`";
   b. Print the hexadecimal representation of the value of the queue record's *InPointer*;
   c. Print "`  Out=0x`";
   d. Print the hexadecimal representation of the value of the queue record's *OutPointer*;
   e. Print "`  Num=`";
   f. Print the value of the queue record's *NumberEnqueued*; and
   g. Print a carriage return character (0x0D) and a line feed character (0x0A) to move the cursor to the next line of the screen.

   Note that you should adjust the spacing in the output for readability and consistency among the commands that produce this output, (i.e., `D`, `E`, and `S`), as shown below.

```
Type a queue command (D,E,H,P,S):S
 Status:  In=0x1FFFE100  Out=0x1FFFE100  Num=0
Type a queue command (D,E,H,P,S):E
Character to enqueue:A
Success:  In=0x1FFFE101  Out=0x1FFFE100  Num=1
Type a queue command (D,E,H,P,S):D
A:        In=0x1FFFE101  Out=0x1FFFE101  Num=0
Type a queue command (D,E,H,P,S):D
Failure:  In=0x1FFFE101  Out=0x1FFFE101  Num=0
```

10. Repeat from step 3.

In addition, the program must meet these requirements.

- The program is to use three circular FIFO queues: the program queue (4-byte QBuffer), the receive queue (80-byte RxQBuffer), and the transmit queue (80-byte TxQBuffer). RxQBuffer and TxQBuffer are reserved for interrupt-driven character I/O. Otherwise, this program uses the same code as written for Lab Exercise Seven with new Init_UART0_IRQ, UART0_ISR, GetChar, and PutChar. The following is a summary of the parameter specifications for each of the new and modified subroutines. (Note: unlike subroutines, ISRs do not have parameters.)
  - GetChar
    - Input parameter:
      - (none)
    - Output parameter:
      - R0: character dequeued from receive queue (unsigned byte ASCII code)
  - Init_UART0_IRQ
    - Input parameter:
      - (none)
    - Output parameter:
      - (none)
  - PutChar
    - Input parameter:
      - R0: character to enqueuer to transmit queue (unsigned byte ASCII code)
    - Output parameter:
      - (none)
- It must use GetChar and PutChar subroutines from prelab work that interface with UART0_ISR via the receive and transmit queues, respectively, for character I/O.
- It must use subroutines from Lab Exercise Six for string I/O.
- It must use subroutines from Lab Exercise Seven for queue operations (Dequeue, Enqueue, and InitQueue) and for hexadecimal and decimal number output (PutNumHex and PutNumUB, respectively).
- It must have properly commented subroutines, including a comment header block that includes these components.
  - Description of functionality
  - List of subroutines called
  - Lists of registers
    - Input parameters
    - Output parameters
    - Modified registers (after return, not preserved by subroutine)

Lab Procedure

1. Create a new directory and Keil MDK-ARM project for this exercise.
2. Write a properly commented and properly formatted KL05 assembly language program according to the preceding specification.
3. Build the program with Keil MDK-ARM to create a listing file and a linker map file.
4. Test your program. Make sure to test these cases, which must be demonstrated to the instructor.
   - Uppercase and lowercase commands
   - Commands with an empty queue
   - Commands with a partially full queue
   - Commands with a full queue
   - Commands with circular buffer operation
5. Examine the KL05 listing and memory map produced for your program, and list the exact memory range, (i.e., the starting address and the address of the last byte), for these components:
   a) Executable code, (i.e., total `MyCode AREA`),
   b) UART0 ISR code,
   c) Constants in ROM, (e.g., prompt string), and
   d) RAM used for the following:
      i. Program queue (buffer and record structure),
      ii. Receive queue (buffer and record structure), and
      iii. Transmit queue (buffer and record structure).
6. Demonstrate steps 4 and 5 for your lab instructor. Following demonstration, capture the terminal screen output to include and discuss in your documentation.
7. Submit your source (.s) file, listing (.lst) file, map (.map) file, and documentation to the myCourses assignments for this exercise..

Baseline Metrics

The metrics in the table below can be used to compare your solution to a baseline solution. They are provided solely for your information and personal curiosity about the efficiency of your solution. There is an extra credit opportunity, (see Extra Credit Opportunity section), for minimizing the number of instructions executed by UART0_ISR. Otherwise, there are no grading criteria associated with how your code compares with these baseline metrics.

Language	Routine	Instructions/LOC	Code Size (bytes)
Assembly	main program*	115	306
C	main	65	380
Assembly	GetChar	7	16
C	GetChar	7	28
Assembly	PutChar	10	22
C	PutChar	7	34
Assembly	Init_UART0_IRQ	71	146
C	Init_UART0_IRQ	23	180
Assembly	UART0_ISR	27	58
C	UART0_ISR	9	72
Assembly	**Total RO Size**	—	2280
C	**Total RO Size**	—	2404

*Not including instructions from provided program template.

Extra Credit Opportunity

UART0_ISR (5 points). Design UART0_ISR so that it executes a minimal number of instructions to service the UART0 IRQ. Points will be awarded for a correctly working ISR based on the minimum number of instructions (counting BL Enqueue and BL Dequeue instructions but not including the instructions in Enqueue and Dequeue) executed for the following conditions based on checking if TIE (then if TDRE) followed by if RDRF, as developed in lecture notes.

UART IRQ Condition	5 Points	3 Points	1 Point	No Points
RDRF only	≤ 16	17	18	≥ 19
RDRF and TDRE	≤ 24	25	26	≥ 27
TDRE only	≤ 21	22	23	≥ 24

Documentation

Prepare a document (in Microsoft Word or PDF format) containing the following.

- The screen capture from procedure step 6 along with an explanation of how the output shown in the screen capture verifies that the UART0 ISR operation is correct.
- The memory ranges from procedure step 5.

Submission

myCourses Assignments
Separate assignment for each item below • Documentation (including cover sheet) • Source code (.s) • Listing (.lst) • Map (.map)

As specified in "Laboratory and Report Guidelines," late submissions are penalized per day late, where "day late" is calculated by rounding submitted time to the nearest greater or equal integer number of days late, (i.e., ceiling function).

Note: To receive grading credit for this lab exercise you must earn points for *both* the demonstration *and* the documentation.