Jon Warriner

04/11/2019

Project 2: Circular buffer, UART and interrupts

ECEN 5813, Spring 2019

Part 1

Hardware Block Diagram



**Figure 1: Hardware Block Diagram**

**Figure *1*** shows the hardware block diagram. A PC is connected to the FRDM-KL25Z board via a USB connection. There is an on-board USB to UART connection to the KL25Z MCU. For this project the KL25Z uses the UART peripheral and a GPIO. The selected GPIO is connected to a LED.

**Software Architecture**

**Figure *2*** shows the layered software architecture.



**Figure 2: Software Architecture**

**Main** – Initialize clocks, peripherals and other modules.

**Counting and Display** – Keep a tally of received characters and build the output display data.

**Rx Buffer** – Ring buffer for passing received characters from the UART to the Counting and Display module.

**Tx Buffer** – Ring buffer for passing display data from the Counting and Display module to the UART.

**ISR** – Interrupt Service Routine responsible reading/writing characters between the ring buffers and the UART. The ISR is triggered when a new character is received by the UART and upon completion of a transmitted character.

**UART Driver** – This is the software abstraction of reading from and writing to the UART peripheral. The ISR talks to the UART peripheral through the UART driver. Notice that the Counting and Display box and the UART Driver box are both extended beyond the ISR and Ring Buffers to the right. This is because there is a shortcut from the Counting and Display module to enable the UART transmit interrupt. During initialization a pointer to the UART transmit interrupt enable function is passed to the Counting and Display module. The UART transmit interrupt is level triggered and will continually jump to the ISR when the transmit buffer is ready for a new character. In order to avoid this the ISR disables the UART transmit interrupt when the TX ring buffer is empty. The Counting and Display module uses the pointer to the UART transmit interrupt enable function to trigger the start of a new display update.

**UART** – This is the UART peripheral itself.

Software Modules

UART Driver



**Figure 3: UART Driver Interfaces**

This module creates an interface to the UART peripheral. Only one instance of this module is allowed.

**Input Interfaces**

/\*\*

\* @brief Initialize UART0

\*

\* Initialize UART0 for the application

\*

\* @return void.

\*/

**void** **UART\_init**();

The init interface initializes the UART peripheral. This includes choosing the clock source, configuring pin-muxing, setting the baud rate, configuring and enabling interrupts, and enabling the peripheral.

/\*\*

\* @brief Enable UART0 TX Interrupt

\*

\* @return void

\*/

**void** **UART\_EN\_TX\_INT**();

Enable/unmask the UART transmit interrupt.

/\*\*

\* @brief Disable UART0 TX Interrupt

\*

\* @return void

\*/

**void** **UART\_DIS\_TX\_INT**();

Disable/mask the UART TX interrupt.

/\*\*

\* @brief Put a character in the UART0 TX buffer

\*

\* Assume the empty/full status of the buffer has already been checked

\* and place a character in the TX buffer.

\*

\* @param data character to place in the buffer

\*

\* @return void

\*/

**void** **UART\_TX**(**char** data);

Place a character in the UART transmit buffer. This function does not check if the transmitter is available. It is assumed that has already been checked.

/\*\*

\* @brief Use UART0 to transmit a character when the buffer is empty

\*

\* Wait (blocking) for the UART0 TX buffer to be empty then transmit

\* a character.

\*

\* @param data character to place in the buffer

\*

\* @return void

\*/

**void** **UART\_TX\_block**(**char** data);

Wait for the UART transmit buffer to be empty and then place a character in the buffer. This function blocks execution until the transmit buffer is available.

**Output Interfaces**

/\*\*

\* @brief Is UART0 ready to transmit a character?

\*

\* Check the TDRE bit in S1 register to determine if the TX

\* buffer is empty or has a character that has yet to go out.

\*

\* @return uint8\_t 1 - TX buffer is empty, 0 - TX buffer is full

\*/

uint8\_t **UART\_TX\_rdy**();

Check to see if the UART transmit buffer is ready for a new character.

/\*\*

\* @brief Do we have a new character in the UART0 RX buffer?

\*

\* Check the RDRF bit in S1 register to determine if the RX

\* buffer is empty or has a new character available.

\*

\* @return uint8\_t 1 - RX buffer is full, 0 - RX buffer empty

\*/

uint8\_t **UART\_RX\_full**();

Check to see if there is a new character available in the receive buffer.

/\*\*

\* @brief Get a character from the UART0 RX buffer

\*

\* Assume the empty/full status of the buffer has already been checked

\* and get a character from the RX buffer.

\*

\* @return character read from the buffer

\*/

**char** **UART\_RX**();

Read a character from the UART receive buffer. This function does not check if there is a character in the buffer. It assumes that check has been done externally.

/\*\*

\* @brief Receive a character from UART0 when the buffer is full

\*

\* Wait (blocking) for the UART0 RX buffer to be full then read

\* a character.

\*

\* @return character read from the buffer

\*/

**char** **UART\_RX\_block**();

Wait for the UART receive buffer to contain a character and then read it. This function blocks program execution until a character is available.

Ring Buffer



**Figure 4: Ring Buffer Interfaces**

This module creates an interface to a ring buffer. Multiple instances of this module are allowed.

**Data Structure**

/\*\*

\* define the ring buffer structured data type

\*/

**typedef** **struct**

{

**char** \*Buffer;

int32\_t Length;

int32\_t Ini;

int32\_t Outi;

}ring\_t;

**Input Interfaces**

/\*\*

\* @brief Create a new ring buffer of "length" chars

\*

\* Given a length (in chars) return a pointer to a new

\* ring\_t type. Return 0 on failure.

\*

\* @param length Length of buffer in chars

\*

\* @return pointer to ring\_t type or 0 on failure

\*/

ring\_t \***ring\_init**( int32\_t length );

Initialize a new ring buffer. This function allocates a buffer of *length* bytes. It also initializes the input and output indices in the data structure.

/\*\*

\* @brief Insert a new char into the buffer

\*

\* Given a pointer to an existing ring buffer and a piece of

\* data, insert the new data into the buffer.

\*

\* @param ring\_t Pointer to an already initialized ring buffer

\* @param data New data to add to the buffer

\*

\* @return 0 on success, -1 on failure

\*/

int32\_t **insert**( ring\_t \*ring, **char** data );

Add a new character to the buffer if it is not already full.

**Output Interfaces**

/\*\*

\* @brief Extract (remove) the next char from the buffer

\*

\* Given a pointer to an existing ring buffer and a pointer to

\* a data location, extract the oldest piece of data from the

\* buffer.

\*

\* @param ring\_t Pointer to an already initialized ring buffer

\* @param data Pointer to a location to write the extracted data

\*

\* @return 0 on success, -1 on failure

\*/

int32\_t **extract**( ring\_t \*ring, **char** \*data );

Retrieve a character from the buffer if it is not already empty.

/\*\*

\* @brief Return the number of entries in the buffer

\*

\* Given a pointer to an existing ring buffer return the number

\* of entries in that buffer.

\*

\* @param ring\_t Pointer to an already initialized ring buffer

\*

\* @return number of buffer entries, returns -1 on error

\*/

int32\_t **entries**( ring\_t \*ring );

Return the number of entries currently in the buffer.

Counting and Display Module



**Figure 5: Counting and Display Module Interfaces**

This module is responsible for counting received characters and maintaining the totals as well as building the display output. Both sides of this module interface to ring buffers. There is a ring buffer that passes characters from the UART to the RX\_task and a separate ring buffer that the Display\_task uses to pass data to the UART transmitter.

**Data Structure**

/\*\*

\* define the counter and display structured data type

\*/

**typedef** **struct**

{

**void** (\*transmit\_trig)(); //pointer to a function to trigger transmission of //display data

ring\_t \*ibuf; //pointer to input ring buffer. This contains //incoming characters to be counted.

ring\_t \*obuf; //pointer to output ring buffer. This contains //formatted output strings to be transmitted

uint32\_t char\_ctrs[256]; //counters for each of the 256 possible characters

**char** sbuf[80];

uint8\_t trig; //set to 1 to trigger the display update to start or //start over

uint8\_t updating; //flag indicating that the display is currently //updating

uint8\_t i; //index of character count that is currently being //updated

}disp\_t;

**Input Interfaces**

/\*\*

\* @brief Initialize the display module

\*

\* @param d pointer to a display structure

\* @param ibuf pointer to an input ring buffer

\* @param obuf pointer to an output ring buffer

\* @param tx\_func pointer to a function to trigger transmission of the output buffer

\*

\* @return pointer to ring\_t type or 0 on failure

\*/

int32\_t **disp\_init**(disp\_t \*d, ring\_t \*ibuf, ring\_t \*obuf, **void** (\*tx\_func)());

Initialize the data structure for the counting and display module.

* Initialize all counters to 0
* Initialize control variables for trigger (*trig*), index (*i*) and updating flag (*updating*) to 0.
* Initialize pointer to input ring buffer. This buffer is instantiated externally with just a pointer passed to this module.
* Initialize pointer to the output ring buffer. This buffer is instantiated externally with just a pointer passed to this module.
* Initialize pointer to a function that will initiate transmit of the display data. In this implementation this will be a pointer to a function that enables the UART transmit interrupt.

/\*\*

\* @brief Check if there are any new characters in the RX ring buffer

\*

\* If there are new characters in the ring buffer then we need to add them to

\* our tally

\*

\* @param d pointer to a display structure

\*

\* @return void.

\*/

**void** **RX\_task**(disp\_t \*d);

Check for new characters in the input ring buffer. If there are any there, read them all and add to the tally for each specific character. Also, if there are newly counted characters, set the *trig* flag to tell the display task to start a fresh update.

**Output Interfaces**

/\*\*

\* @brief Build the results display to send out the serial port

\*

\* We've got new characters and we need to update the count(s) on the display.

\*

\* @param d pointer to a display structure

\*

\* @return void.

\*/

**void** **Display\_task**(disp\_t \*d);

Build the display data and put it in a ring buffer for transmission. This function builds the display for one counted character at a time and transmits that display before building the next. The display format for a printable character is:

char – count

Where, char is the character and count is the number of times that character has been received.

The display format for non-printable characters is:

0xcc – count

Where, 0xcc is the hexadecimal value of the character and count is the number of times that character has been received.

Characters with a count of 0 are not added to the display.

Once the display has been fully updated it is not updated again until new characters are received.

If the trigger (*trig*) flag is set by the RX\_task the display update is started from character number 0. If the display is in the middle of an update when a new character is received it finishes the current character display then starts over from character 0. This ensures that the displayed counts for all characters is accurate and allows this function to remain relatively idle when no characters are being received.

Part 2

Questions

1. Is your implementation thread safe? Why or why not?
   * Whether or not the ring buffer implementation is thread safe depends on how it is being used. If a single buffer is being added to in one thread and extracted from in a second task then the implementation is safe as long as one thread does not interrupt the other.
2. What potential issues exist if the same buffer is used by both interrupt and non-interrupt code? How can these issues be addressed?
   * If interrupt code and non-interrupt code are both adding to or extracting from a single buffer then the buffer could accidentally be over or under flowed. Let’s say non-interrupt code is in the process of adding to the buffer. The buffer has been checked to have one more slot for data and the new data is about to be written. Now, we get an interrupt just before the data is written. The interrupt also wants to add a piece of data to the buffer. Since the non-interrupt code never finished adding its data the interrupt also sees that there is one remaining free slot in the buffer. The interrupt goes ahead and adds its data to the buffer. Now, we return to the non-interrupt code which still thinks its ok go add data to the buffer. This data will be written and overflow the buffer. If both interrupt and non-interrupt code are extracting from the buffer the opposite issue could occur and underflow the buffer. The initial thought to prevent this is to only allow the non-interrupt code to extract from the buffer and only allow interrupt code to insert into the buffer or vice versa. Assuming that the buffer is extracted from in a timely manner this reduces the risk of overflow or underflow. However, let’s say we have non-interrupt code that is about to extract from a full buffer. We’re somewhere in the extract code but haven’t grabbed the data or updated the in/out indices yet. Now we get an interrupt that wants to insert a new piece of data. That is going to fail because the non-interrupt code hadn’t finished its extraction. The best way to address this issue is to treat the inserting and extracting of the ring buffer as critical code and disable interrupts before and re-enable interrupts after these operations.
3. How could you test these issues?
   * These issues could be tested by intentionally causing an interrupt while updating the buffer. I was not able to make this happen in my testing by only using the UART interrupts. To force the test you’d need to use something like a timer interrupt or some interrupt source that you could control to happen when you know the buffer is being updated.

Part 3

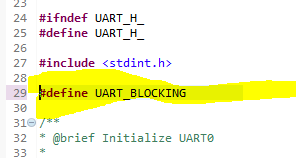
Unit test source code and makefile are in the Uinttest directory.

**make test** - Build and run “ringtest” which is a completely predefined set of tests to meet the basic requirements of this part of the project.

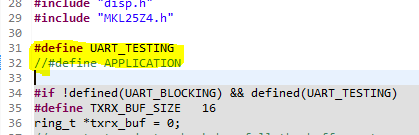
**make test\_long** – Build and run “long\_test” which is a long running test of random insertions and extractions from the buffer to meet the extra credit requirement for this part. The test is currently configured to run 1 million iterations. After the standard CUnit output I added some of my own statistics to show how many insertions and extractions were successful and how many were correctly identified as overflow or underflow of the buffer.

Part 4

In order to build the “blocking” version for this part go to uart.h and make sure the #define for UART\_BLOCKING is uncommented. Also, in Project2.c, make sure the #define for UART\_TESTING is uncommented and the #define for APPLICATION is commented.

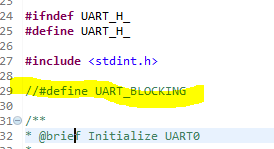


**Figure 6: uart.h configuration to build the “blocking” test for this part**

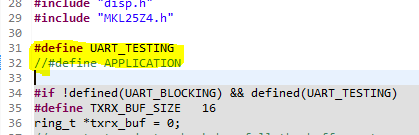


**Figure 7: Project2.c configuration to build the “blocking” test for this part**

In order to build the “non-blocking” version for this part go to uart.h and make sure the #define for UART\_BLOCKING is commented. Also, in Project2.c, make sure the #define for UART\_TESTING is uncommented and the #define for APPLICATION is commented.



**Figure 8: uart.h configuration to build the “non-blocking” test for this part**



**Figure 9: Project2.c configuration to build the “non-blocking” test for this part**

## Questions

1. For each implementation, what is the CPU doing when there are no characters waiting to be echoed? What is the behavior of the GPIO toggle in the non-blocking implementation?
   * In the case of the blocking version the CPU is just sitting in a loop, watching for the UART RX buffer to be full. This implementation “blocks” any other code from running in the main loop. The only other code that could potentially run is an interrupt that may be enabled in the system. What happens in the non-blocking version depends on the implementation. If the transmit interrupt is always enabled it will continually be serviced indicating that that the UART TX buffer is ready for a new character. The flags that trigger the TX interrupt are only cleared by writing a new character into the buffer. In this situation the TX interrupt may hog most if not all of the processing time and prevent the background loop from running. When testing with this configuration the GPIO would typically only toggle after a received character. The background loop was able to run what the transmitter was busy echoing the character. Because of this I changed my implementation to disable the TX interrupt when there are no characters to transmit. This allows the main loop to run most of the time. The ISR only runs when there is a character to receive or transmit. In this case the GPIO toggles very rapidly. It stays on the same state slightly longer when the ISR is running.
2. For each implementation trace the sequence of events that occur by listing, in order, the functions called from the point that a character sent to the FRDM board has been received until the point where the echoed character has been sent.
   * Blocking
     1. UART\_RX\_block() – wait in this function until a character is available then read the character into a temporary variable.
     2. UART\_TX\_block(temp) – put the value that was stored in the temporary variable into the TX buffer. Wait in this function for the TX buffer to empty.
     3. Go back to step 1 and repeat forever.
   * Non-blocking
     1. Received character triggers the ISR, UART0\_DriverIRQHandler(void)
     2. UART\_RX\_full() – checks to see if the ISR was caused by a received character. It was so go on to step three. If there was not received character we’d go to step 4.
     3. insert(txrx\_buf, UART\_RX()) – get the character from the RX buffer and put it into a ring buffer.
     4. UART\_TX\_rdy() && (entries(txrx\_buf) != 0) – check if the transmitter is ready for data and if we have any characters to transmit. This could mean that we go a TX interrupt or we just got a character. In all of my testing the received character went out before the next receive interrupt. The buffer never got fuller than one character.
     5. Return to the main loop until the next interrupt.
3. Comment on the interface for sending and receiving characters presented to the main() application code for blocking vs. non-blocking variation. Which variation is easier to code to?
   * The blocking version was easier to code because it didn’t require any extra configuration to enable interrupts at the CPU or peripheral level.

Part 5

## Questions

1. What is the CPU doing after the last character has been received and while the report is being printed?
   * Receipt of new characters and transmission of the report is entirely handled by interrupts. While the report is being printed the main loop continues to run. Any code that is in the main loop is able to run. Additionally, a UART RX interrupt could be serviced to receive new characters.
2. Baud rate aside, what limits the rate at which the application can process incoming characters? What happens when characters come in more quickly than they can be processed?
   * The first limitation to how quickly the application can process incoming characters is the latency in grabbing a character and putting it in the ring buffer. If the ISR can’t be serviced fast enough the UART RX buffer could overflow. The second limitation is how quickly the application can recognized that there are new characters in the ring buffer, extract them and add them to the individual character tallies. This limitation could be a tradeoff between how efficiently the code is written, what other tasks are being processed and the length of the RX ring buffer.
   * In my application I was not able to make it miss characters. I tried blasting it with several thousand characters and it counted them all. Theoretically you could overflow either the UART RX buffer or the ring buffer. My application is not able to update the display for every individual character that comes in. Instead, it just makes sure that the display is correct by the time the last character is received.
3. How does the size of the circular buffer affect report output behavior (especially during an onslaught)? What is an appropriate buffer size to use for this application? Why?
   * My implementation has two circular buffers. The first buffer shuttles characters from the UART RX buffer to the counting and display task. This buffer needs to be big enough to handle any latency between receiving and processing characters. In my testing this buffer never filled up with more than three characters when blasting the port with several thousand characters at a time. I chose a buffer size of 16 to provide plenty of headroom. The second buffer is used to move the report data from the counting and display task to the UART for transmission. My counting and display builds the report for one counted character at a time. Assuming that the count for a character could fill an unsigned 32-bit integer, the string length of the report for one character would be a maximum of 20 characters including some formatting and the null. Since my application only reports one character at a time I made this buffer 32 characters long. This method of only reporting one character at a time also allows me to easily restart the report if new characters are received while it is being generated.