**ECEN 5813 Principles of Embedded Software**

**Project 2 S2019 final report**

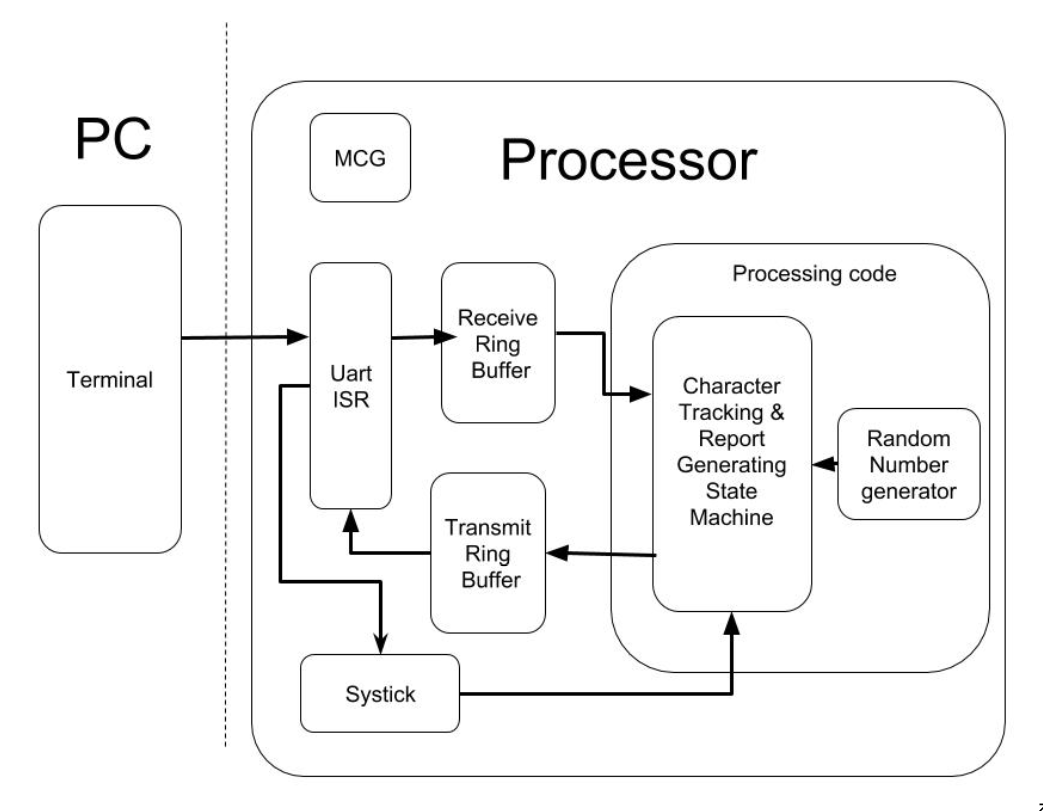
**Circular buffer, UART and interrupts**

Linfeng Li

Wei Ting Chien

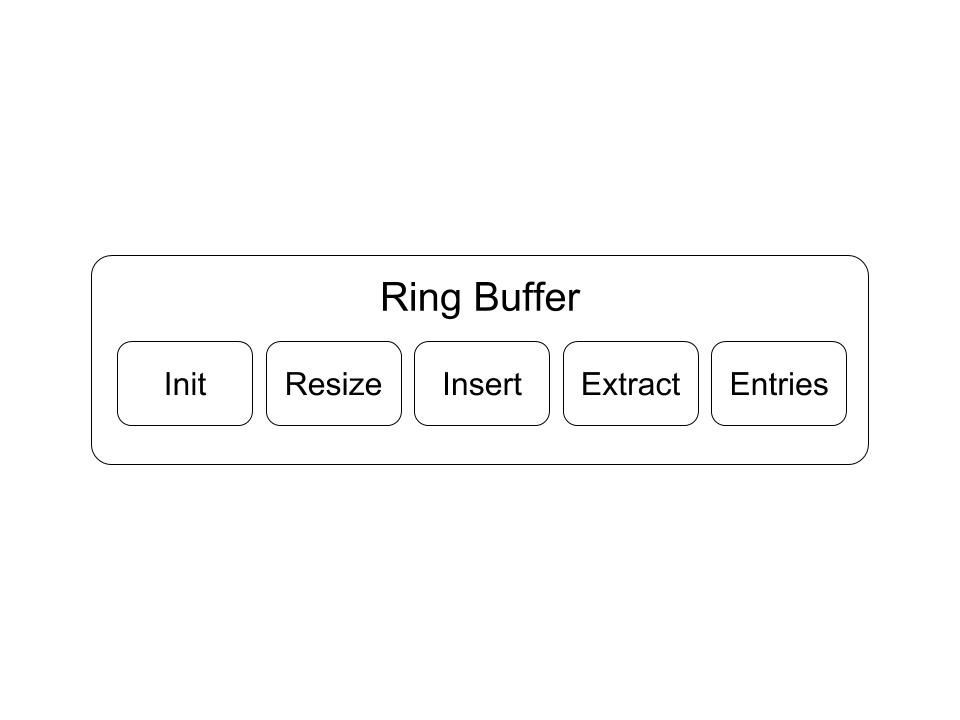
4/9/2019

**Part 1 : Block diagram and architecture**



The software diagram can be divided into two parts, PC and Processor. User will input the data stream with terminal software on PC side, and the terminal software will transfer those data to the processor by Uart in real time. While processor receive data, it will run “Ring Buffer” code to save data. And then those data in ring buffer will go through the processing code to generate a report. Moreover, a state machine is used in processing code in order to track ring buffer’s condition. To avoid any disturbance in report, “Input delay” features defines the delay time interval and the algorithm. Furthermore, clock sources from MCG is needed for all the peripherals and functions in processor.

|  |  |
| --- | --- |
| Components | Functionalities |
| Terminal | Communicate with Processor. |
| UART ISR | Communicate with terminal.  Receive data and store in the receive ring buffer.  Read the data in the transmit ring buffer and sent them out via UART. |
| Transmit Ring buffer | Store the data of the report waiting to be read by the transmit interrupt. |
| Receive Ring buffer | Buffer the incoming characters. Read by the main loop. |
| State machine | Monitor the receive ring buffer to keep track of the incoming characters.  Write the transmit ring buffer and generate report. |
| Systick | Increment the tick counter and serve as the reference of the maximum input delay feature.  Tick counter reset by UART receive interrupt. |
| MCG(Multipurpose Clock Generator) | Distribute the clock to core and peripherals. |
| Random number generator | Use the algorithm of linear congruential generator to generate random number along with the state machine. The random number is part of each report. |

 In ring buffer, there are 5 components, Init, Resize, Insert, Extract, and Entries.

|  |  |
| --- | --- |
| Components | Functionalities |
| Init | Initialization of the ring buffer with certain length. |
| Resize | Resize an exisiting buffer with certain length.  Data existed in the buffer would be maintained.  Only latest data would be maintained if new size cannot hold all existing data. |
| Insert | Insert one element into the buffer.  Buffer Full situation detection . |
| Extract | Extract one element from the buffer.  Buffer empty situation detection. |
| Entries | return the number of elements in the buffer that have been wrote but yet to read. |

## Questions

**Part2 : Circular buffers**

1. **Is your implementation thread safe? Why or why not?**

No. Both the insert and extract function access and modify the same set of global resource, including the input, output pointer, and almost full signal. If any thread is preempted by other thread, and both threads would try to read and write the same variable. If the preemption happens right within either insert or extract function and the preempting thread calls either insert or extract function, the preempting thread would end up with running with partial updated shared resources, which could cause problem.

1. **What potential issues exist if the same buffer is used by both interrupt and non-interrupt code? How can these issues be addressed?**

With some code analysis, it turns out that the given the same buffer referred extract function in the main loop could be interrupted by the insert function in the ISR. The opposite way is not viable, which means the insert function in the main loop cannot be interrupted by the insert function in the ISR.

To address this problem, before putting the new report into the transmit buffer, the report generation state machine would check if the transmit interrupt enable bit is disabled. the transmit interrupt enable bit would be disabled inside the transmit ISR when ISR cannot read out any characters from the transmit buffer. Upon a disabled transmit interrupt enable bit, the previous report has been transmitted completely and the new one now is safe to put into the buffer.

It is not necessary to protect the receive buffer, since the insert function in the ISR could safely interrupt the extract function in the main loop.

Moreover, the elements to handle the ring buffer are all declared with volatile attributes so that compiler would only generate assembly code in which the direct memory loading instruction would always be used to get the value instead of from a register that is previously holding the value.

1. **How could you test these issues?**

We could draw the data flow diagram of both of the insert and extract function and try to interrupt one function with another one at every possible places with every possible status. If the interrupt won’t cause any of the partial update problem, we could conclude that the interrupt could be unlimitedly used in that particular manner, and vice versa.

The Linux and POSIX library also proved to be help. We can put the functions to be tested into two threads. The preempting thread is set with higher priority than the to-be-preempted thread does. The preempting thread is created with a fix duration of sleep in the beginning. The thread that would be preempted is created also with a certain duration of sleep in the beginning. The duration of the sleep set to the to-be-preempted thread is less than that set to the preempting thread and varies, so that the function in the to-be-preempted thread could be interrupted at different places and thus emulate the real situation where the asynchronized interrupt could possibly disturb the functions in the main loop. Every time the results are kept track and could be used to analyze the effect caused by the interrupts and the global shared resources.

**Part4 : UART device driver**

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?**

CPU will hang and wait for the user input in the blocking implementation. The CPU would be stuck at a while loop to indefinitely wait for the Receive Data Register Full Flag turning low, which means a new character is ready to be read out.

On the other hand, in non-blocking implementation, CPU can go through another procedure, i.e. remain data inside ring buffer without being read. The GPIO toggle will also keep going in non-blocking implementation even if there is no data received. The main loop would keep checking if there is any new data been put into the receive buffer in which the ISR would store the incoming character. If there is no data in that ring buffer, the entries call would just return with a zero. The main loop could continue to execute other task.

1. **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:**

* In the UART\_BLOCKING\_RX\_GETCHAR function, poll the UART0\_S1\_RDRF flag and proceed only when the flag turns high. Upon a new incoming character, UART0\_S1\_RDRF turns low.
* In the UART\_BLOCKING\_RX\_GETCHAR function, follow the previous step, the UART Data Register would be read and returned.
* The returned character would be use as the argument in the UART\_BLOCKING\_PUTCHAR to transmit the character back.
* In the UART\_BLOCKING\_PUTCHAR, poll the UART0\_S1\_TDRE flag and proceed only when the flag turn to high(subroutine: UART\_TX\_RD). Upon a successfully transmitted character, UART0\_S1\_TDRE turns high. This step means to put in a new character to be transmitted only after the previous one is done.
* In the UART\_BLOCKING\_PUTCHAR, follow the previous step, the UART Data Register would be write with the argument from the caller, which stores the character to be transmitted.

**Non-blocking:**

* Upon a new incoming character, UART0\_S1\_RDRF turns high and trigger the Receiver Interrupt.
* Context switch from the main loop, and enter the UART0 ISR. In the ISR, the UART0\_S1\_RDRF flag is checked. If the flag turns high, the UART data register is read and the read data is inserted into the ring buffer. Then, context switch back to the main loop.
* Proceed until the main loop calls the entries function. Upon a non-zero return, the main loop would check if the transmit interrupt is enabled. If it is not yet enabled, the application would enable the transmit interrupt.
* Upon a high UART0\_S1\_TDRE flag, the transmit Interrupt is triggered if enabled.
* Context switch from the main loop, and enter the UART0 ISR. The UART0\_S1\_TDRE flag is checked. If the flag is high, the ring buffer would be read. If a new character is successfully read out from the ring buffer, the UART data register is write with the extracted data. If the read with only a buffer empty return, which means no character pending to be echoed, the transmit interrupt would be disabled. Then, context switch back to the main loop.

1. **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?**

Blocking is easier to code to because we do not need to consider the effect of different interrupts or functions. Every character receiving and transmitting is synchronized with the main loop. However, the blocking application code will make the program stop in one place until a new character is received or transmitted. The interfaces are just GETCHAR with a return and PUTCHAR with an input argument. This is very simple to implement.

On the other hand, the non-blocking implementation can let the processor do other tasks and only check for the specific flag that indicates whether the transmitter is available or a new character is ready to be read out. A new transmission is started by enabling the transmit interrupt, which could automatically trigger the UART interrupt. In this case, the receiving and transmitting is asynchronized regarding the main loop, which means those could happened everywhere inside the main loop. In this case, function calls which would access and modify global shared resources need special care to be implemented. The partial update could cause problem if the functions are interrupted while updating the global shared resources. Some synchronization mechanism need to be use if global resources could cause potential problem.

Generally, the non-blocking’s interface maybe is harder to implement, because those interrupt flag checking and enabling need to be paid attention to make sure the checking and enabling order could correctly sense the proper UART working status and enable the transmission without disturbing the main loop.

**Part5 : Application**

1. **What is the CPU doing after the last character has been received and while the report is being printed?**

To cope with the onslaught, the application code has a maximum input delay and maximum number of tracked characters. Time to transmit from the terminal to the FDRN board is increasing linearly with the amount of the input characters. The time to generate the report and transmit back actually could be consider as increasing logarithmically with the amount of the input characters, because the characters making up the report would increase rapidly with just a few character in the beginning, and then increase with lower and lower speed as the input characters stream in. Therefore, we decide to count the characters remained unreported since the previous report and generate the new on when it reach a certain pre-set maximum number of tracked characters. If the overhead to process this amount of characters and generate the new report plus the total time consumed by transmitting the new report is less than the total character time to send this amount of character from terminal, we could conclude that the system could handle any period of incessant onslaught. So when the unreported new character reach the pre-set maximum value, the state machine would generate a new report and put the new report into the transmit ring buffer given the completion of the transmit of the previous report.

The input delay is devised to handle the situation where the onslaught stops with the amount of the unreported new character less the set maximum value. When the onslaught stops, the state machine would wait for 50 ms and then generate a new report, which means putting the new report into the transmit ring buffer. This wait is a non-blocking one. The state machine would check whether the tick is over a pre-determined value. If it does, the state machine would switch to a new state and starting generate the new report. If it doesn’t, the state machine would return and run the pseudo-random number generator once.

While the report is being printed, there are several situations. If an onslaught happens, the received characters are stored into the receive ring buffer and read by the application state machine. If instead just several characters have been received, the state machine should wait for the 50 ms input delay. Since the ring buffer insertion cannot be interrupted by the extraction, The state machine would always check whether the previous report has already been completely transmitted by checking whether the UART transmit interrupt is enabled. If UART transmitting interrupt is disabled, which means that the ISR has already read all of the data of the previous report, sent via UART, disabled the UART transmit interrupt. Thus, a new report could be generated and stored in the transmit ring buffer.

1. **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?**

With our onslaught handling algorithm, only overhead of sensing the UART receiving signal, entering the interrupt, checking interrupt flags, putting the character in the receiver buffer into internal ring buffer, switching back to the main loop, and reading this character out of the ring buffer could limit the rate of the processor to process the input characters. If there is some spare time after subtracting the overhead from the total character time, we could always set an appropriate maximum number of tracked characters to maintain enough margin to generate the new report. If the spare time is very small, the interval between each report during the onslaught would be longer. If the spare time is zero, the maximum number of tracked characters theoretically would be infinite and the new report cannot be generated until the onslaught stops. If the spare time is negative, the characters remained in the ring buffer waiting to be read would pile up. Depending on the size of the buffer and the duration of the onslaught, the ring buffer could possibly be filled up. If the overhead of receiving interrupt ISR only is large enough compared to the total character time, we would likely loss some of the incoming characters.

1. **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?**

We have separate buffers for the transmitter and receiver. The speed of reading a character from the receiving ring buffer plus other receiving interrupt ISR overhead is much faster than the character time 115200bps baud rate. So the receiving ring buffer just needs to be larger enough to hold the incoming characters while the CPU is generating the new report. In my case, 50 bytes is enough. For the transmitting ring buffer, it is necessary to be larger enough to hold the entire report, since the insert call of the ring buffer is unable to be interrupted but the extract call in the ISR. In my case, I set the size of the transmitting buffer with 1000 bytes in order to handle the extreme situation when all of the 256 types of character need to be reported.