|  |
| --- |
| -------------------------------------------------------------------- |
| Q1: Implement the following inline functions: |
| inline int CircularBufferIsFull(CircularBuffer\* que) |
| inline int CircularBufferIsEmpty(CircularBuffer\* que)  A:  // **Implementation 1** - using one empty slot.  // That is buffer full case is when there is only one  // free entry left between writePtr and readPtr.  // Advantage – simple; Disadvantage – modulus operation to chk full, one slot wasted.  struct CircularBuffer  {    char buffer[BUF\_SIZE];    unsigned int readPtr;  // points to where the next read can start from    unsigned int writePtr; // points to where the next write can start from  }  inline int CircularBufferIsFull(CircularBuffer\* que)  {    int queue\_full = 0;    if ((que->writePtr+1)%BUF\_SIZE == que->readPtr)    {      queue\_full = 1;    }    return queue\_full;  }  inline int CircularBufferIsEmpty(CircularBuffer\* que)  {    int queue\_empty = 0;    if ((que->readPtr == que->writePtr)    {      queue\_empty = 1;    }    return queue\_empty;  }  // **Implementation 2** - using one extra bit in each pointer  // that is toggled every time pointer crosses 0 value  // or wraps around.  #define BUF\_SIZE 128    typedef union \_buf\_ptr    {      unsigned int ptr\_wrap;      struct      {        ptr      : 7;        wrap\_bit : 1;      }s;    }buf\_ptr;  struct CircularBuffer  {    char buffer[BUF\_SIZE];    buf\_ptr read\_ptr;    buf\_ptr write\_ptr; // points to where the next write can start from  }  inline int CircularBufferIsFull(CircularBuffer\* que)  {    int queue\_full = 0;    if ((que->write\_ptr != que->read\_ptr) && (que->write\_ptr.s.ptr == que->read\_ptr.s.ptr))    {      queue\_full = 1;    }    return queue\_full;  }  inline int CircularBufferIsEmpty(CircularBuffer\* que)  {    int queue\_empty = 0 ;    if ((que->read\_ptr == que->write\_ptr)    {      queue\_empty = 1;    }    return queue\_empty;  } |
|  |
|  |
|  |
|  |
| -------------------------------------------------------------------- |
| Q2: How to compute "Interrupt Latency"?  A : Interrupt Latency is defined as the number of cpu clock cycles it takes from the time Interrupt is asserted to the time the first instruction of ISR is executed. In the case of ARM Cortex M3 the Interrupt Latency is 12 clock cycles, provided:   * The memory system has zero wait state   Ie., memory is as fast or faster than the processor, thus requiring the processor to not wait any additional cycles for a fetch.   * The system level design of the chip does not add delay in the interrupt signal connections between the interrupt sources and the processor * The Interupt service is not blocked by another current running exception/interrupt service:   Cortex M3 has NVIC (Nested Vector Interrupt Controller) that allows a lower priority Interrupt to be pre-empted by a higher priority one. The above mentioned 12 cycles latency is achieved only if processor is not currently processing an interrupt of equal or higher priority.   * The current executing instruction is not doing an unaligned transfer/bitband transfer (which can take 1 extra transfer cycle)   Unaligned transfer means a 32 bit transfer to a 2 byte aligned address. And bit band transfer is transfer from/to the bit band memory region where each bit can be individually addressed, to allow for atomic operations.   * During these 12 cycles after interrupt is triggered, Cortex M3 processor also saves /pushes some of the registers to stack (PC, LR, r0 – r3, r12), thus saving the ISR from having to perform these time consuming context saving operations. |
|  |
|  |
|  |
| -------------------------------------------------------------------- |
| Q3: Serial pc(USBTX, USBRX); // tx, rx |
| What API (function) do you use to change the baud rate?  unsigned int baud\_rate=9600;  pc.baud(baud\_rate) |
|  |
|  |
|  |
|  |
| -------------------------------------------------------------------- |
| Q4: How many bit(s) does TDO have? Is TDO an input or output type? (Ref: JTAG Pins)  TDO is a JTAG output. It is a serial data output from shift register, hence it is only one bit wide. |
|  |
|  |
|  |
|  |
|  |
|  |
| -------------------------------------------------------------------- |
| Q5: |
| What is the address of ledArr[0]? = **0x100001f4** |
| What is the address of ch? = **0x10007ff4**  **Subtract ch and ledArr – then you get stack plus heap. ledArr is global and ch is local.** |
|  |
| The program was compiled using O0 optimization flag.  (gdb) file homework4.elf  Reading symbols from homework4.elf...done.  (gdb) load homework4.elf  Loading section .text, size 0x83d0 lma 0x0  Loading section .ARM.exidx, size 0x8 lma 0x83d0  Loading section .data, size 0xb8 lma 0x83d8  Start address 0x9f4, load size 33936  Transfer rate: 4 KB/sec, 1696 bytes/write.  (gdb) c  Continuing.  Breakpoint 1, main () at main.cpp:30  30 pc.baud(9600);  **(gdb) p &ch**  **$3 = (int \*) 0x10007ff4**  **(gdb) p &ledArr[0]**  **$4 = (mbed::DigitalOut \*) 0x100001f4 <ledArr>**  (gdb) |
| /\* |
| \* Enter a hex number [0-9a-fA-F]; Decode it in 4-bit binary format and display them on 4 on board leds. |
| \*/ |
|  |
| #include "mbed.h" |
|  |
| Serial pc(USBTX, USBRX); // tx, rx |
| DigitalOut ledArr[4] = {DigitalOut(LED1), DigitalOut(LED2), DigitalOut(LED3), DigitalOut(LED4)}; |
| void DisplayLed(int ch) |
| { |
| int i=0; |
|  |
| if (ch>='0' && ch<='9') |
| ch-='0'; |
| else if (ch>='A' && ch<='F') { |
| ch-='A'; |
| ch+=10; |
| } else if (ch>='a' && ch<='f') { |
| ch-='a'; |
| ch+=10; |
| } else |
| ch=0; |
|  |
| for (i=0; i<4; i++) { |
| if(ch& (1<<i)) |
| ledArr[i] = 1; |
| else |
| ledArr[i] = 0; |
| } |
| } |
|  |
| int main(void) { |
| int ch; |
| pc.baud(9600); |
| pc.printf("\r\nHello World!"); |
| while(1) { |
| pc.printf("\r\nEnter:"); |
| ch = pc.getc(); |
| pc.putc(ch); |
| DisplayLed(ch); |
| } |
| } |