ECE3849
D-Term 2021

Real Time Embedded Systems

Module 2 Part 4

Module 2 Part 4 Overview

- Handling Shared Data
  - Circular Buffers: FIFOs

Circular Buffer: FIFO

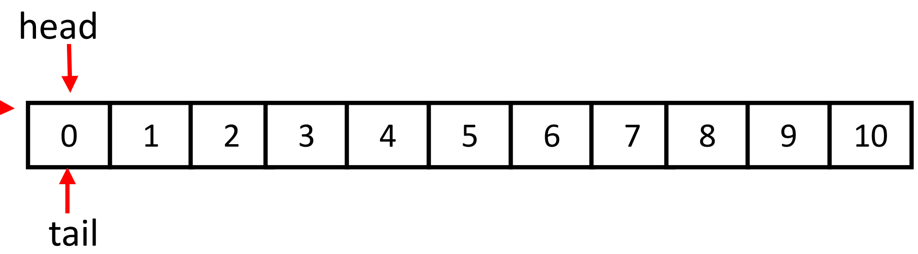
- FIFO stands for First In First Out.
  - FIFOs are implemented with a global data array with fixed maximum length.
  - Two threads share the FIFO, one reading and one writing.
  - As long as the reading rate can keep up with the average writing rate, no data will be lost if the FIFO is sized properly.
- Several globals variables are required to implemented a FIFO.

```
#define FIFO_SIZE 11              // FIFO capacity is 1 item fewer
typedef char DataType;            // FIFO data type
DataType fifo[FIFO_SIZE];         // FIFO storage array
volatile int fifo_head = 0;       // index of the first item in the FIFO
volatile int fifo_tail = 0;       // index one step past the last item
```

  - We write the FIFO with a function fifo_put(), which increments the fifo_tail index.
  - We read the FIFO with a function fifo_get(), which increments the fifo_head.

FIFO: Put and Get operations

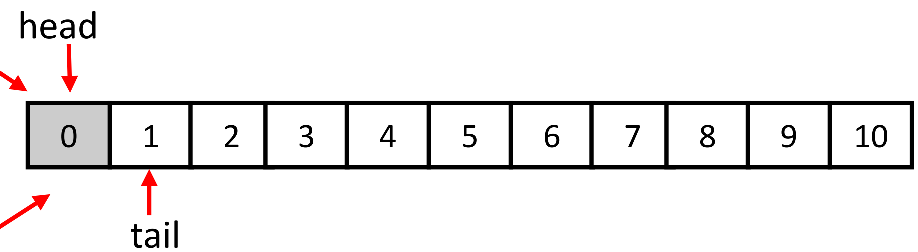
Initialized: FIFO Empty
fifo_head = fifo_tail = 0.



Contains data

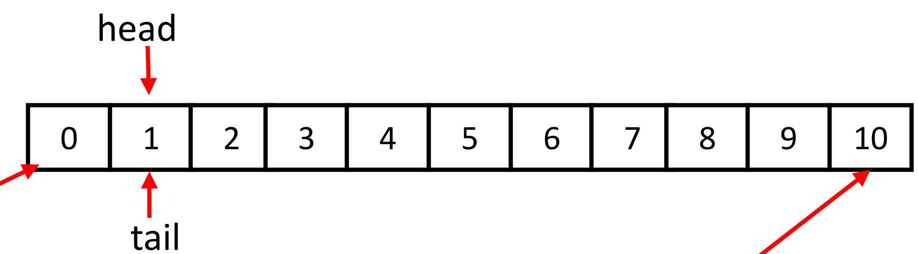
fifo_put(): Writes into FIF0.

- Increments new_tail value.
- If fifo_head != new_tail
  - writes data, increments fifo_tail & is not full



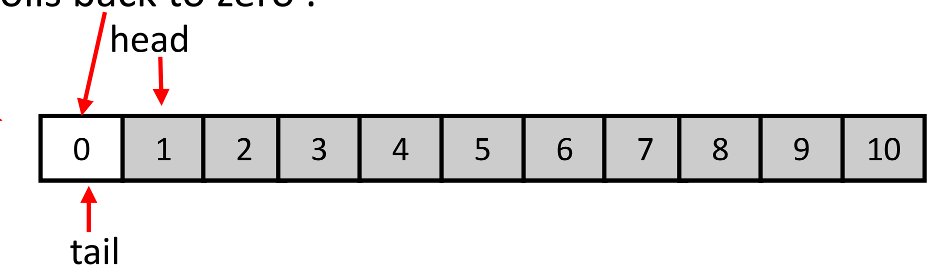
fifo_get(): Reads out of FIF0.

- If fifo_head != fifo_tail,
  - It reads data.
  - Increments the head value.
  - Returns not empty.
- If reads again in this state, the FIFO returns an empty status.



Say you write 9 more times, the tail would be at 10.
If the head or the tail is at the maximum index, the index rolls back to zero .

FIFO Full State: fifo_tail + 1 == head
If a fifo_put() is issued now no further data can be written. The tail is not incremented and full is returned.



FIFO: Code Example

```
// put data into the FIFO, skip if full
// returns 1 on success, 0 if FIFO was full
int fifo_put(DataType data)
{
    int new_tail = fifo_tail + 1;
    if (new_tail >= FIFO_SIZE) new_tail = 0; // wrap around
    if (fifo_head != new_tail) {    // if the FIFO is not full
        fifo[fifo_tail] = data;     // store data into the FIFO
        fifo_tail = new_tail;       // advance FIFO tail index
        return 1;                   // success
    }
    return 0;    // full
}
```

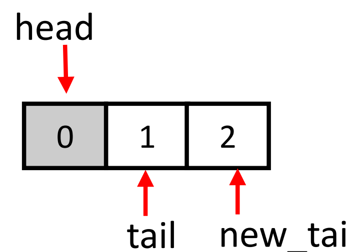
- Local variable new_tail contains incremented tail value.
- If the new_tail is at its maximum ivalue, it wraps around to zero.
- If the new_tail == head then FIFO is full and nothing is written.
- If it is not full then writes data and updates the fifo_tail to the new_tail value.

```
// get data from the FIFO
// returns 1 on success, 0 if FIFO was empty
int fifo_get(DataType *data)
{
    if (fifo_head != fifo_tail) {   // if the FIFO is not empty
        *data = fifo[fifo_head];    // read data from the FIFO
        fifo_head++;                // advance FIFO head index
        if (fifo_head >= FIFO_SIZE) fifo_head = 0; // wrap around
        return 1;                   // success
    }
    return 0;    // empty
}
```

- If fifo_head == fifo_tail then the FIFO is empty and
  Return 0 meaning no data was read
- If it is not empty then it…
  reads data value.
  Increments the fifo_head value.
  If the incremented head value is at its maximum,
  then head wraps around to zero.
  It returns 1 meaning data was read.

- Do we have any shared data problems?
  - ?

Shared data: fifo_put()




```
// put data into the FIFO, skip if full
// returns 1 on success, 0 if FIFO was full
int fifo_put(DataType data)
{
    int new_tail = fifo_tail + 1;
    if (new_tail >= FIFO_SIZE) new_tail = 0;
    if (fifo_head != new_tail) {    // if the
        fifo[fifo_tail] = data;     // store
        fifo_tail = new_tail;       // advanc
        return 1;                   // succes
    }
    return 0;    // full
}
```

If fifo_get() interrupt, it will see old tail value and will not be aware that the fifo_put() is in process.

fifo_head is read only once & atomic.
fifo_tail is written only once & atomic.

- Do we have any shared data problems?
  - No problems.
  - FIFO may report empty while fifo_put() is in process.
  - Once completed fifo_get() will need to be called again to pickup the new data.

Shared data: fifo_get()

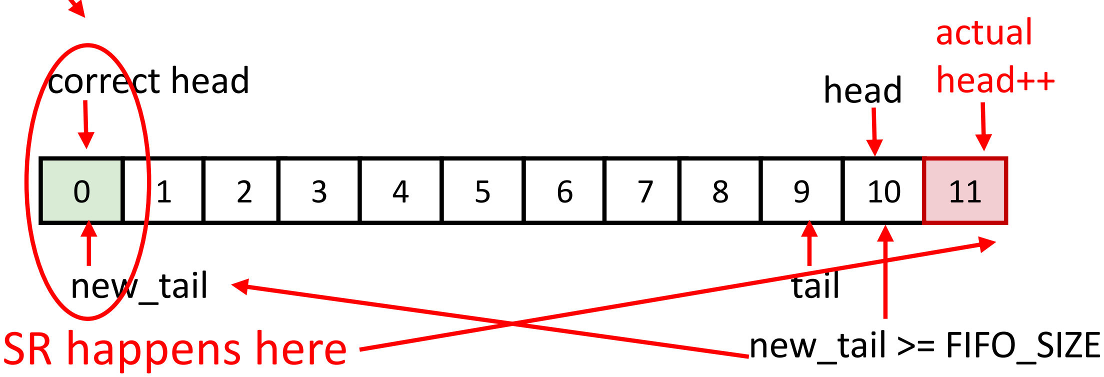
```
// returns 1 on success, 0 if FIFO was full
int fifo_put(DataType data)
{
    int new_tail = fifo_tail + 1;
    if (new_tail >= FIFO_SIZE) new_tail = 0; // wra
    if (fifo_head != new_tail) {     // if the FIFO
        fifo[fifo_tail] = data;      // store data i
        fifo_tail = new_tail;        // advance FIFC
        return 1;                    // success
    }
    return 0;     // full
}
```

If interrupt has two consecutive fifo_put() calls new_tail should = fifo_head = FULL.

BUG: head incorrect, new_tail writes to location 0 and now thinks it is not full and can overwrites the data in the whole buffer.

No problem: If fifo_put() happens here, tail will either think it is full or will increment fifo_tail and add data. fifo_get() still has data to read.

```
int fifo_get(DataType *data)
{
    if (fifo_head != fifo_tail) {    // if the FIFO
        *data = fifo[fifo_head];     // read data fr
        fifo_head++;                 // advance FIFC
        if (fifo_head >= FIFO_SIZE) fifo_head = 0;
        return 1;                    // success
    }
    return 0;     // empty
}
```



ISR happens here

head++ = 11 >= FIFO_SIZE, head should be 0.

Shared data: get_fifo(): fix

```
// get data from the FIFO
// returns 1 on success, 0 if FIFO was empty
int fifo_get(DataType *data)
{
    if (fifo_head != fifo_tail) {   // if the FIFO
        *data = fifo[fifo_head];    // read data fr
        fifo_head++;                // advance FIFO
        if (fifo_head >= FIFO_SIZE) fifo_head = 0;
        return 1;                   // success
    }
    return 0;     // empty
}
```

fifo_head +1 stored in local variable.

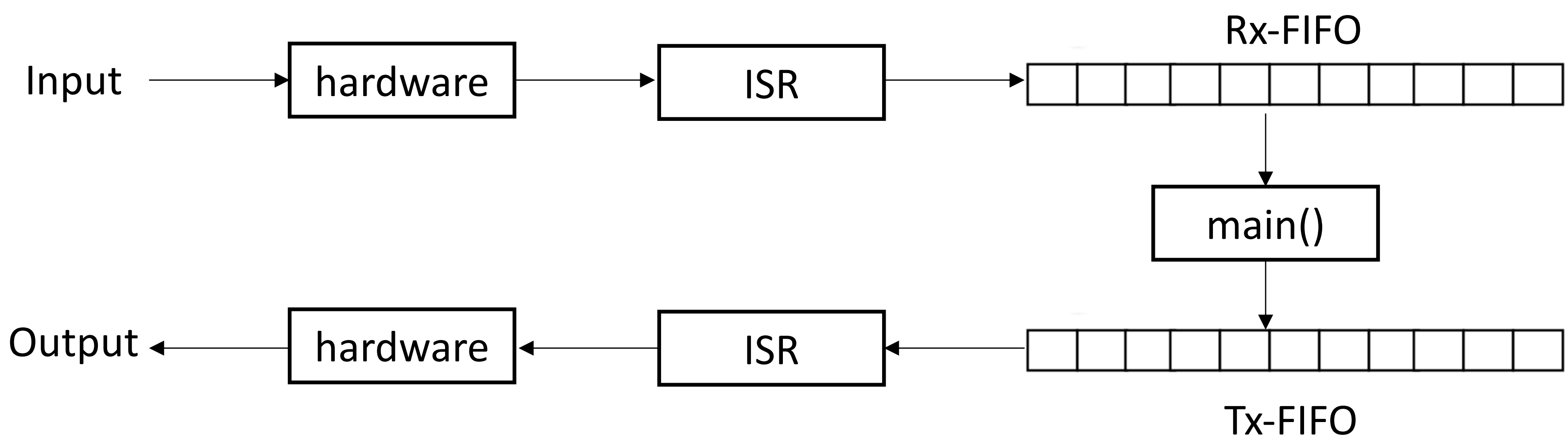
```
if (fifo_head >= FIFO_SIZE-1) {
        fifo_head = 0;
}
else {
        fifo_head++;
}
```

Writes fifo_head only once in an atomic operation, no intermediate values possible.

- In addition, you could read fifo_head into a local variable, local_head, similar to what is done in the fifo_put() function.
  - Although not required reduces the number of accesses to the global variables and the potential for shared data bugs.

Common FIFO Usage

- FIFO are commonly used to communicate between high priority (low latency) ISRs and low priority (high latency) ISRs or main() functions.
- Often one task can be bursty, delivering a large packet of data quickly and then nothing for a while. While the other task processes data at a regular interval.
  - FIFOS are perfect for rate matching these high priority burst accesses with the slow and steady performance of a lower priority task.
  - The larger the bursts or the longer the latency, the larger the FIFO needs to be.



Example: ece3849_shared_data

- Similar to ece3849_int_latency, the program calculates latency, response_time and missed deadlines.
- It uses the fifo_put() and fifo_get() functions from the lecture slides.
- It has only one ISR: event0_handler().
  - The ISR uses fifo_put() to write up to 5 characters at a time in alphabetical order to the FIFO.
  - The ISR is configured to trigger every 10 msec.
- In the main() while loop for low priority tasks.
  - The loop waits 100msec.
  - Calls fifo_get() once to read a single character from the FIFO.
  - If there is a character in the FIFO,
    - It prints the character to the LCD screen.
    - Increments the location of the string for the next character.
    - If the location is at the last character on the screen it clears the screen and starts printing at the top.

ece3849_shared_data ISR code

Create a static variable to hold next character value.

- Loops 5 times
- If there is room in the FIFO, fifo_put() writes the character into the FIFO
  - Increments the character value for next time.
  - If the character is U wraps back around to A.

```c
void event0_handler(void)
{
    uint32_t t;
    t = TIMER0_PERIOD - TIMER0_TAR_R; // read Timer A count using direct register access
    if (t > event0_latency) event0_latency = t; // measure latency
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT); // clear interrupt flag

    int i;
    static char c = 'A';
    for (i = 0; i < 5; i++) { // attempt to put multiple items into the FIFO
        if (fifo_put(c)) {
            c++;  // go to the next char only if put was successful
            if (c > 'U') c = 'A';
        }
    }
    delay_us(EVENT0_EXECUTION_TIME); // handle event0

    if (TimerIntStatus(TIMER0_BASE, 1) & TIMER_TIMA_TIMEOUT) { // next event occurred
        event0_missed_deadlines++;
        t = 2 * TIMER0_PERIOD; // timer overflowed since last event
    }
    else t = TIMER0_PERIOD;
    t -= TimerValueGet(TIMER0_BASE, TIMER_A); // read Timer A count using driver
    if (t > event0_response_time) event0_response_time = t; // measure response time
}
```

Event0_handler writes A through U in alphabetical order up to 5 characters if the FIFO is not full.

Static Variables

- Static variables
  - They behave similar to global variable except their scope is limited to the function or block they are declared in.
  - They stay in memory from call to call and preserve their value. They are not dynamically allocated.
    - Static variables are stored in the data segment instead of the stack.
- Lab 1:
  - It is important to minimize what is on the stack because the stack size is extremely limited.
    - Stack overflows will result in a Fault ISR and demonstrate in unpredictable ways.
  - Large data structures should be declared either globally or using static so as not to run out of stack space.

ece3849_shared_data: main()

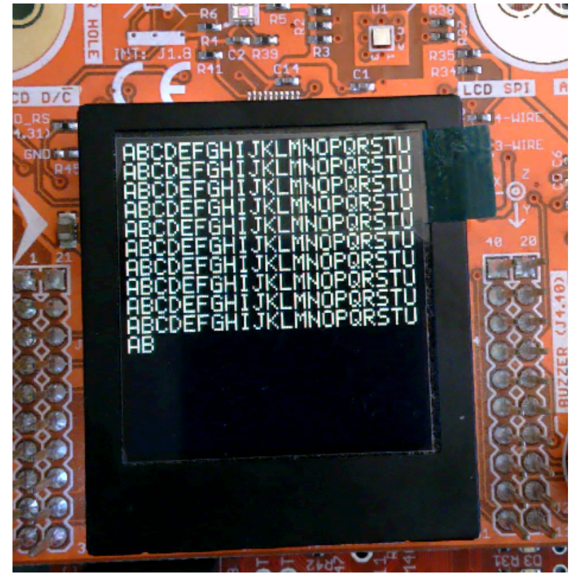
Waits 100 msec.

- Reads the FIFO.
- If the FIFO is not empty, prints the character to the LCD buffer.
- Increments the LCD screen location for the next character.
- If it is at the bottom of the screen. Clears the screen and sets the next character location to the top left of the LCD.
- Updates the LCD display with the buffered data.

```
    // main loop
    while (1) {
        delay_us(100000); // 0.1 sec
        if (fifo_get(&c)) {
            GrStringDraw(&sContext, &c, /*length*/ 1, /*x*/ x, /*y*/ y, /*opaque*/ false);
            x += 6;
            if (x > LCD_HORIZONTAL_MAX - 6) {
                x = 0;
                y += 8;
                if (y > LCD_VERTICAL_MAX - 8) {
                    y = 0;
                    GrContextForegroundSet(&sContext, ClrBlack);
                    GrRectFill(&sContext, &rectFullScreen); // fill screen with black
                    GrContextForegroundSet(&sContext, ClrYellow);
                }
            }
        }
        GrFlush(&sContext); // flush the frame buffer to the LCD
    }
}
```

Functional results

- Default conditions with shared data bug in fifo_get().
  - No errors seen, A through U printed normally.
  - Chance of hitting bug extremely low.



- We increase chances of hitting error by adding a delay, just after the fifo_head++ statement.

```
int fifo_get(DataType *data)
{
    if (fifo_head != fifo_tail) {   // if the FIFO is not empty
        *data = fifo[fifo_head];    // read data from the FIFO
//          IntMasterDisable();
        fifo_head++;                // advance FIFO head index
       delay_us(1000);
        if (fifo_head >= FIFO_SIZE) fifo_head = 0; // wrap around
//          IntMasterEnable();
        return 1;                   // success
    }
    return 0;   // empty
```

- With the delay we can see 11 missing characters when the bug occurs.
- To fix the issue, we have two options.
  - Disable interrupts around critical area
  - Implement the fix in get_fifo() from lecture.

Summary of Shared Data Handling

- Preemptive scheduling using interrupts and ISRs can give high performance / low latency at the cost of shared data problems.
- There are several ways to handle shared data problems.
  - Disabling interrupts guarantees mutually exclusive accesses, only one task at a time can access the data.
    - Pro: Easy to implement.
    - Con: Increases latency.
    - Con: Disabling specific interrupts can lead to priority inversion.
  - Careful Coding and Analysis
    - Use local variables to remove shared data possibilities.
    - Avoid the most common shared data bugs by using atomic operations and minimizing multiple reads and writes to global variables.
    - Con: Difficult to verify proper implementation, bugs can be very infrequent and intermittent.
  - Synchronization techniques
    - Con: Adds complexity and protocol, can be resource intensive requiring additional variables and increasing response times.
    - Con: Need to understand rate at which tasks are happening to guarantee no data loss.
    - Reading multiple times/sequence numbers: Used when interrupts are infrequent, to verify an interrupt did not occur and change the data in the middle of the critical data section.
    - Buffering – limit reading from one part of the buffer while writing to another part.
    - Binary Semaphores – using a Boolean flag to control when shared data accesses and critical functions occur.
    - Mailboxes – using a binary Semaphore to limit access to specific data and data structures.
    - FIFOs – allows read and write data tasks to operate independently at different times and rates without conflicts.