

Module 2: Lecture Questions

Module 2 - Lecture 6 (Thursday – 4/1/21)

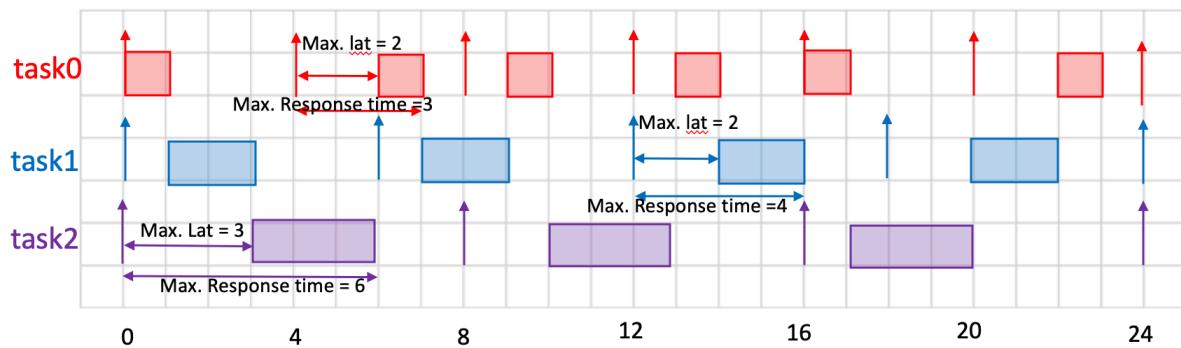
1. What is the only way to pass data into or out of an ISR?
2. What is shared data and when do problems arise because of it?
3. Explain how the ece3849_int_latency example program calculates CPU utilization.
4. Why will the value for CPU Utilization returned from ece3849_int_latency differ from the CPU utilization that we calculated using the RMS Theory?
5. What RMS Theory condition is not met, using the Earliest Deadline First Scheduling strategy?
6. Why not always use Earliest Deadline First (EDF) Scheduling? Under what conditions would you choose to use EDF. This is a nice link that outlines the pros and cons of EDF at the bottom.

<https://microcontrollerslab.com/earliest-deadline-first-scheduling/>

7. The system with the characteristics below uses Earliest Deadline First scheduling.
 - a. Use graphical method to determine the latency, response time and if it is schedulable for each task.
 - b. How many ms do you need to graph to find the worst-case response time values?
 - c. What is the CPU utilization of the system? What is the utilization upper bound for EDF that guarantees that tasks will be schedulable?

https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling

Event	Period	Execution Time	Latency	Response Time	Relative Deadline	Schedulable ?
TaskA	4 ms	1 ms	2	3	4	$3 < 4$ YES
TaskB	6 ms	2 ms	2	4	6	$4 < 6$ YES
TaskC	8 ms	3 ms	3	6	8	$6 < 8$ YES



Module 2 - Lecture 7 (Monday – 4/5/21)

8. In the Time Example, we looked at solving our shared data problems in several ways. One option was to disable some or all of the interrupts. What are the pro's and con's of disabling some and all interrupts?
9. Under what conditions does priority inversion occur? Why is priority inversion to be avoided at all costs?
10. What is an atomic operation?
 - a. Explain why `x++` is not atomic. Explain how using this statement in a low-priority thread could create a shared data bug.
 - b. What do I need to know to determine if `x = y` is an atomic operation?
 - c. Why is it important to use atomic operations during critical shared data sections of the code? Explain two shared data bugs that are caused by non-atomic operations.
11. We saw that the following code had a shared data bug, when the function `SecondsSinceMidnight` was called from a higher priority interrupt

```
long lSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++lSeconds >= 60 * 60 * 24) {
        lSeconds = 0;
    }
}

long SecondsSinceMidnight(void) {
    return lSeconds;
}
```

- a. Describe the cause and the outcome of the data bug.
- b. What are two fixes for this bug?

See next page for more

12. Your system is set up to interrupt when a fault is detected. The ISR increments a global fault counter, and main() handles the faults, decrementing the counter. The following code outlines the communication between the ISR and main(). Assume this code runs on an unspecified 32-bit CPU architecture: 32-bit **reads** and **writes** are atomic.

```

volatile uint32_t fault_count = 0;

void FaultDetectISR(void) {
    <clear interrupt flag>;
    fault_count++;
}

void main(void) {
    <init>;

    while (1) {
        while (fault_count > 0) {
            fault_count--;
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}

```

- a. Find the shared data bug in this code. Explain the sequence of events that may cause an error.

```

void main(void) {
    <init>;

    while (1) {
        while (fault_count > 0) {
            fault_count--; // read-modify-write the shared global
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}

```

The above highlighted code indicates where the global is read, modified and written in main(), which is generally a non-atomic operation (assignment only states that reads and writes are atomic). FaultDetectISR() can interrupt main() after fault_count is read but before it is written, and increment the global (write to it). After the ISR returns, the global is overwritten with a value derived from an outdated local copy in main(). Thus, one fault is missed. This is an intermittent bug.

There is one other place where main() reads fault_count: the condition for the inner while() loop. However, this code, even when combined with the other access to fault_count, does not introduce other shared data bugs (such as non-atomic read). Reading fault_count to evaluate the while() condition is atomic. The comparison outcome is not invalidated even if the body of the while() loop is interrupted, as the ISR only increments fault_count.

- b. Fix the shared data bug by disabling, then re-enabling interrupts. Exclude `<handle one fault>` from the critical section to minimize impact on interrupt latency.

In real life, you would first check to see if interrupt was enabled like in lecture.

Solution if you recognized that the while() loop condition does not introduce shared data bugs:

```
void main(void) {
    <init>

    while (1) {
        while (fault_count > 0) {
            IntMasterDisable();
            fault_count--; // read-modify-write the shared global
            IntMasterEnable();
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}
```

Solution if you thought the while() loop condition is part of the shared data problem:

```
void main(void) {
    uint32_t local_faults;      // new local variable
    <init>

    while (1) {
        IntMasterDisable();
        local_faults = fault_count; // read global into local var
        fault_count = 0;           // overwrite global
        IntMasterEnable();
        while (local_faults > 0) { // local variable: not shared
            local_faults--;        // local variable: not shared
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}
```

You may come up with other variations, but they should exclude `<handle one fault>` from the critical section (where interrupts are disabled) to receive full credit.

- c. Fix the shared data bug without disabling interrupts. Hints: You want to make the communication between the ISR and main() one-directional: if the ISR writes to `fault_count`, main() only reads it. This implies that `fault_count` cannot be decremented or modified in any other way in main(). Use local variables to replace the lost functionality.

It is not possible to make a read-modify-write operation atomic without disabling interrupts (or resorting to special CPU instructions). The solution, following the hints, is to make main() only read the global, but not write to it. We then need a local variable to count handled faults. The number of new faults can be determined as the difference between `fault_count` and the local `handled_faults`.

```
void main(void) {
    uint32_t handled_faults = 0; // new local variable
    <init>

    while (1) {
        while (fault_count - handled_faults > 0) { // atomic read
            handled_faults++; // local variable
            <handle one fault>; // does not use fault_count
        }
        // other unrelated code
    }
}
```

There is an edge case when the fault counts overflow from 0xffffffff to 0. If you write the while() loop condition as `(fault_count > handled_faults)`, it will fail when `fault_count` overflows before `handled_faults`. However, you can write this condition as `(fault_count - handled_faults > 0)` or alternatively as `(fault_count != handled_faults)` and it will work properly. You do not need to make sure your code is safe in this edge case to receive full credit.

See next page for more questions.

13. A timer ISR keeps track of time in milliseconds. Functions have been written to allow main() to read and adjust the time. This code runs on a **32-bit** CPU: 32-bit reads and writes are atomic.

```

volatile uint32_t time = 0;

void TimerISR(void) {           // periodic ISR, period = 1 ms
    <clear timer interrupt flag>; // <...> delineates pseudo-
    code
    time++;
}

uint32_t GetTime() {           // called from main()
    return time;
}

void AdjustTime(uint32_t adj) { // called from main()
    time += adj;
}

```

- a. AdjustTime() is known to intermittently corrupt the `time`. Why does this happen? What is the worst case deviation from the correct time due to a single occurrence of the shared data issue? Exclude the special case when `time` wraps around.

```

volatile uint32_t time = 0;

void TimerISR(void) {           // periodic ISR, period = 1 ms
    <clear timer interrupt flag>; // <...> delineates pseudo-code
    time++;
}
uint32_t GetTime() {           // called from main()
    return time;
}
void AdjustTime(uint32_t adj) { // called from main()
    time += adj;
}

```

- a) AdjustTime() is known to intermittently corrupt the `time`. Why does this happen? What is the worst case deviation from the correct time due to a single occurrence of the shared data issue? Exclude the special case when `time` wraps around.

The read-modify-write operation `AdjustTime()` performs on the `time` shared variable is not atomic on this CPU architecture. If interrupted after the read, but before the write, the ISR increments `time`. After returning from the ISR, `AdjustTime()` overwrites `time` with a modified local copy from before the ISR call. Thus, one `time` increment is lost, and the expected time deviation due to one occurrence of the shared data issue is 1ms (one timer period). Two or more timer interrupts may occur if there are other ISRs with high CPU load in the system. In that case the time error may be several ms.

- b. Why do shared data issues not occur in `GetTime()`?

GetTime() only reads the time shared variable. This is an atomic operation on a 32-bit CPU architecture.

- c. Fix AdjustTime() to resolve the shared data issue with the minimum changes to the existing code.

In real life, you would first check to see if interrupt was enabled like in lecture.

Here is the solution that disables interrupts to fix the shared data bug:

```
void AdjustTime(uint32_t adj)
{
    IntMasterDisable();
    time += adj;
    IntMasterEnable();
}
```

Module 2 - Lecture 8 (Tuesday – 4/6/21)

14. For the problem in 13, assume that AdjustTime() is called at a rate slower than the TimerISR period. Implement a Mailbox such that TimerISR is the only thread writing to time.
15. A read-modify-write operation is occurring in a low priority task like main(). A high priority ISR can interrupt it at any time. What conditions need to be met in the ISR not to have a shared data problem?
16. Rewrite this non-atomic operation to be atomic using the bit-banding macro provided by TI. $x \&= \sim(0x1 \ll 5);$
17. In class we saw that by doing repeated Non-Atomic writes we could eliminate the shared data problem in the TimerISR / SecondsSinceMidnight example. Rewrite the TimerISR and SecondsSinceMidnight function to use a sequence number instead of reading all the values. What advantages does this have?

18. A periodic ISR is sampling input data and recording a timestamp for each data point. The function getData() is provided for main() to read one data point and its timestamp. It is important that the timestamp and data point come from the same sample. Assume an `int` can be read or written atomically.

```

int t = 0;
int d;
void inputISR(void)
{
    <clear interrupt flag>;      // <...> delineates pseudo-code
    t = <time>;                  // assume time is always non-zero
    d = <data>;                  // sample data
}
int getData(int *pt, int *pd)
{
    if (t != 0) {
        *pt = t;
        *pd = d;
        t = 0;
        return 1; // success
    }
    else
        return 0; // no data available
}

```

- (a) Find all the shared data bugs in the current implementation. Explain the errors they may cause.

```

int t = 0;
int d;

void inputISR(void)
{
    <clear interrupt flag>;      // <...> delineates pseudo-code
    t = <time>;                  // assume time is always non-zero
    d = <data>;                  // sample data
}

int getData(int *pt, int *pd)
{
    if (t != 0) {
        *pt = t; ← Interrupt (1)
        *pd = d; ← Interrupt (2)
        t = 0; ←
        return 1; // success
    }
    else
        return 0; // no data available
}

```

- a) [10 pts] Find all the shared data bugs in the current implementation. Explain the errors they may cause.

1. (more important bug) `getData()` reads the two globals `t` and `d` non-atomically. The ISR writes them. If this operation is interrupted in the “Interrupt (1)” location, the returned timestamp and data will be inconsistent with each other.
2. (less important bug) `getData()` reads `t` to check data availability, then writes 0 to it. If interrupted in the “Interrupt (2)” location, the data point acquired by `inputISR()` is lost.

- (b) Is the following modified implementation free of shared data bugs that cause data corruption? How often must this getData() be called such that no data points are lost?

```
int getData(int *pt, int *pd)
{
    if (t != 0) {
        IntMasterDisable();
        *pt = t;
        *pd = d;
        t = 0;
        IntMasterEnable();
        return 1; // success
    }
    else
        return 0; // no data available
}
```

Yes, this version fixes the shared data bug #1 that cause data corruption. Interrupting the if() statement in the middle does not invalidate its outcome.

If the time between getData() calls in main() is ever longer than one ISR period, it will miss some data points, as the ISR simply overwrites the globals without waiting for main() to retrieve the data. In order to not lose data points, getData() must be called once per ISR period. It is a good idea to call it more often than that.

(One minor issue is that main() should expect interrupts to be re-enabled when getData() returns.)

- (c) Re-implement inputISR() and getData() to avoid disabling interrupts but still fix the data corruption problem (data point and timestamp mismatch). State any assumption you have made about thread timing.

```

84 int t;
85 int d;
86 int bMailboxFull = 0;
87
88 void inputISR(void)
89 {
90     if (!bMailboxFull) {
91         t = time;
92         d = data;
93         bMailboxFull = 1;
94     }
95 }
96
97 int getData(int *pt, int *pd) {
98     if (bMailboxFull) {
99         *pt = t;
100        *pd = d;
101        bMailboxFull = 0;
102        return 1;
103    }
104    else {
105        return 0;
106    }
107 }
```

19. A periodic ISR reads position and speed sensors and stores the results in circular buffers. The function getxv(), called from main(), returns the latest position and speed from the buffers. Assume an `int` can be read or written atomically.

```

#define N 1000          // number of samples in the circular buffers
volatile int x[N];  // circular buffer for position
volatile int v[N];  // circular buffer for speed
volatile int i = 0; // index of the latest sample in each buffer

void ISR_xv(void) {           // periodic ISR
    <clear interrupt flag>; // <...> delineates pseudo-code
    i++;                     // increment and wrap index
    if (i >= N) i = 0;
    x[i] = <read position>; // store data in circular
buffers
    v[i] = <read speed>;
}
void getxv(int *px, int *pv) { // called from main()
    *px = x[i];
    *pv = v[i];
}
```

- a. In the function `getxv()`, underline all accesses to shared global variables and label them “read,” “write” or “read-modify-write.” Explain the shared data problem(s) that can occur. Explain why some of the accesses to shared data do not cause problems, stating any conditions, if required.

Hint: Note the similarity to the Lab 1 ADC buffer.

```
#define N 1000      // number of samples in the circular buffers
volatile int x[N]; // circular buffer for position
volatile int v[N]; // circular buffer for speed
volatile int i = 0; // index of the latest sample in each buffer

void ISR_xv(void) {           // periodic ISR
    <clear interrupt flag>; // <...> delineates pseudo-code
    i++;                      // increment and wrap index
    if (i >= N) i = 0;
    x[i] = <read position>; // store data in circular buffers
    v[i] = <read speed>;
}                                read read
void getxv(int *px, int *pv) {   // called from main()
    *px = x[i];                ← (1)
    *pv = v[i];
}                                read read
```

Shared data issue:

Global variable `i` is read twice (non-atomic read), returning two different values if the code is interrupted around the location marked (1). The outcome is that the position and speed returned by `getxv()` are from different buffer indices, so different points in time. This inconsistency is likely undesirable.

Accesses to globals without shared data issues:

The read accesses to the arrays `x[]` and `v[]` are not atomic when taken together, but should return consistent results if the index were the same. This is because the `ISR_xv()` needs to overwrite the entire circular buffer before it modifies the accessed entries. The conditions for this to work is that in the buffer index must be saved in a local variable, and the execution time between reading x[] and v[], including any interruptions, must be less than the time to overwrite the circular buffers. One situation where this may fail is when there are other ISRs with long execution time.

Note: You must mention why `x[]` and `v[]` can be read safely, either in part (a) or part (b) to receive full credit.

*`px` and `*pv` could also access shared data, which may result in share data bugs. We cannot say anything specific about these hypothetical bugs (not required for credit).

- b. Fix the shared data bug(s) in `getxv()` without disabling interrupts.

It is safe to access the latest items in `x[]` and `v[]` without disabling interrupts, as long as the time between reading `x[]` and `v[]` is less than the time required to fill the buffers. The buffer index `i` must be read only once, atomically. Reading an `int` is atomic per problem definition.

```
void getxv(int *px, int *pv) {
    int j = i; // atomic read
    *px = x[j]; // index in a local variable
    *pv = v[j];
}
```

Module 2 - Lecture 9 (Thursday – 4/8/21)

20. For the problem in 13, assume that AdjustTime() can be called at most every 100 usec. Explain how you might use a FIFO to fix the shared data problem such that TimerISR is the only thread writing to time.

- a. The TimerISR() is called every msec, assuming TimerISR() will read all the entries in the FIFO when it is called what is the minimum FIFO length that can be used?

If AdjustTime() is called every 100usec, then it could place as many as 10 values in the FIFO in a 1 msec duration. So the minimum FIFO length would be $10 + 1 = 11$.

- b. Using the fifo_get() and fifo_put() commands, update the code below to implement the fifo. Remember to add the required defines and global variables to implement the fifo. You do not have to write the fifo_get() and fifo_put() functions, just use them.

```
42 #define FIFO_SIZE 11           // FIFO capacity is 1 item fewer
43 typedef uint32_t DataType;    // FIFO data type
44 volatile DataType fifo[FIFO_SIZE]; // FIFO storage array
45 volatile int fifo_head = 0; // index of the first item in the FIFO
46 volatile int fifo_tail = 0; // index one step past the last item
47
48 int fifo_put(DataType data);
49 int fifo_get(DataType *data);
50
51 volatile uint32_t time = 0;
52 void TimerISR(void) {
53     // clear interrupt and non-critical stuff
54     uint32_t adj = 0;
55     uint32_t local_time = time + 1;
56     while (fifo_get(&adj)) {
57         local_time += adj;
58     }
59     time = local_time;
60 }
61
62 uint32_t GetTime() {
63     return time;
64 }
65
66 void AdjustTime(uint32_t adj) {
67     fifo_put(adj);
68 }
```

- c. If getTime() could be called from a higher priority interrupt than the TimerISR, would we have a shared data problem? Explain.

It depends on how you wrote the TimerISR. If it writes to the time value multiple times then yes the GetTime could read the value at an intermediate value returning a not fully adjusted time value. The below implementation will have shared data problems. The implementation in part b will not, as it reads time once atomically and the writes time once atomically. If it gets interrupted the return value will be the time before the TimerISR was called or the time after the TimerISR made its updates. Both values are valid and correct.

```

51 volatile uint32_t time = 0;
52 void TimerISR(void) {
53     // clear interrupt and non-critical stuff
54     uint32_t adj = 0;
55     time++;
56     while (fifo_get(&adj)) {
57         time += adj;
58     }
59 }
```

```

volatile uint32_t time = 0;

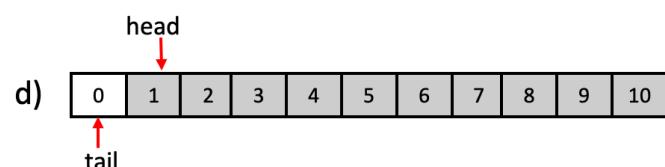
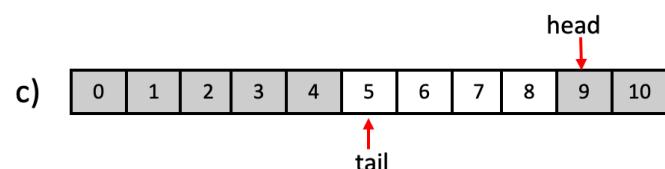
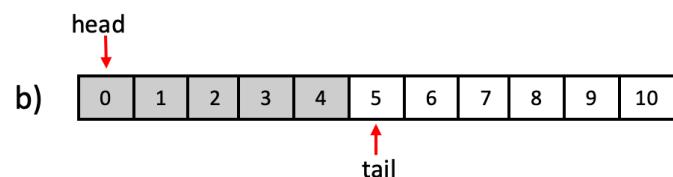
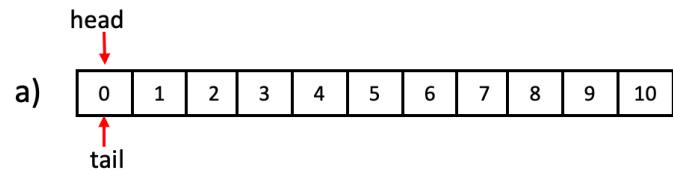
void TimerISR(void) {           // periodic ISR, period = 1 ms
    <clear timer interrupt flag>; // <...> delineates pseudo-code
    time++;
}

uint32_t GetTime() {           // called from main()
    return time;
}

void AdjustTime(uint32_t adj) { // called from main()
    time += adj;
}
```

21. In the shared_data example demonstrated in class, the ISR wrote up to 5 characters every 10 msec into a FIFO. The main function read 1 character every 100 msec. When the ISR saw that the FIFO was full it just stopped writing until there was space in the FIFO.
Not all applications can just stop sending, some will have to write 5 characters of data every 10 msec or data will be lost. Explain how you would modify the FIFO and the main() functionality such that it still reads from the FIFO every 100 msec and prints the new data to the screen but without data loss.
22. In class we discussed the shared data problem with the fifo_get() function. There were two possible solutions to the problem. What were they? Which one was preferred and why?
23. What does the keyword volatile mean? Why do global variables such as fifo_head and fifo_tail have to be volatile?

24. In the figures showing the fifo head and tail index locations for a FIFO holding characters. How many characters are stored in each FIFO? Is the FIFO full or empty?
- Head = tail. Fifo is empty no data in the buffer.
 - 5 characters stored. Not empty , Not full.
 - 7 characters stored, Not empty, Not full.
 - 10 characters stored, Fifo is full.



25.