Shared data, interrupt service routines

1) [30 pts] 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.

```
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. Assume an `int` can be read or written atomically.

1. (more important bug) getData() <u>reads</u> the two globals t and d <u>non-atomically</u>. The ISR writes them. If this operation is interrupted in the "Interrupt (1)" location, the returned timestamp and data will be <u>inconsistent</u> with each other.

2. (less important bug) getData() <u>reads</u> t to check data availability, then <u>writes</u> 0 to it. If interrupted in the "Interrupt (2)" location, the data point acquired by inputISR() is <u>lost</u>.

b) [8 pts] 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
}
```

<u>Yes</u>, this version <u>fixes the shared data bug #1</u> 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 <u>once per ISR period</u>. 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) [12 pts] Re-implement inputISR() and getData() to relax the timing constraint for getData() and avoid disabling interrupts. Your code may reference a data structure we discussed in lecture, but you do not need to re-implement the data structure or the functions that access it.

To relax the relative deadline for calling getData(), we can use the FIFO data structure in Lecture 6, corrected to remove the shared data bug. We then define a new DataType and the desired FIFO_SIZE. This FIFO also allows communication with the ISR without disabling interrupts. Now main() can wait longer between attempts to get new data, but it should empty the FIFO once it gets to it (repeatedly call getData() until it returns 0).

```
#define FIFO_SIZE (<desired FIFO size> + 1)

typedef struct {
    int t;
    int d;
} DataType;     // data type for FIFO

void inputISR(void)
{
    DataType a;
    <clear interrupt flag>;      // <...> delineates pseudo-code
    a.t = <time>;
    a.d = <data>;
    fifo_put(a);
}

int getData(int *pt, int *pd);
{
    DataType a;
    int success;
    success = fifo_get(&a);
    if (success) {
        *pt = a.t;
        *pd = a.d;
    }
    return success;
}
```

Reentrancy

2) [25 pts] (This exercise is from D. E. Simon, "An Embedded Software Primer," Addison-Wesley Professional, 1999.) Which of the numbered lines (lines 1-5) in the following function would lead you to suspect that this function is probably not reentrant? **Explain**.

```
static int iCount;

void vNotReentrant (int x, int *p)
{
    int y;

/* Line 1 */  y = x * 2;
/* Line 2 */  ++p;
/* Line 3 */  *p = 123;
/* Line 4 */  iCount += 234;
/* Line 5 */  printf ("\nNew count: %d", x);
}
```

Line 1 is safe: y and x are local variables.

Line 2 is safe: p is a function argument, stored locally. The pointer p is not to be confused with *p, the `int` to which p is pointing.

Line 3 should be treated with caution: It is writing to whatever p is pointing to, which could be a shared variable. If this were the only shared data access in this function, and the write is atomic, this line should be safe. A write to an `int` (at least 16 bits per C standard) is atomic on 32-bit and 16-bit architectures. If the programmer is disciplined enough to never have p point to a shared variable, then this line is also safe.

Line 4 makes this function non-reentrant (on many, but not all architectures): It is performing a read-modify-write, a non-atomic operation, on a global variable.

Line 5 potentially makes this function non-reentrant: printf() accesses output hardware that is a shared resource. It is possible that printf() is reentrant in this particular C library.

> **Aside: printf() should never be called from real-time code, because its execution time is long and potentially non-deterministic. RTOSs usually supply a fast, feature-limited version of printf() that can be called from real-time code for debugging purposes.**

3) [15 pts] (This exercise is slightly modified from D. E. Simon, "An Embedded Software Primer," Addison-Wesley Professional, 1999.) Where in the following code do you need to disable and re-enable interrupts to make the function reentrant? Rewrite this function to make it reentrant.

```
static int iValue;

int iFixValue (int iParm)
{
    int iTemp;

    iTemp = iValue;
    iTemp += iParm * 17;

    if (iTemp > 4922)
        iTemp = iParm;
    iValue = iTemp;

    iParm = iTemp + 179;
    if (iParm < 2000)
        return 1;
    else
        return 0;
}
```

(solution on next page)

```
static int iValue;          // global variable

int iFixValue (int iParm)
{
    int iTemp;

    IntMasterDisable();
    iTemp = iValue;         // read global
    iTemp += iParm * 17;

    if (iTemp > 4922)
        iTemp = iParm;
    iValue = iTemp;         // write to global
    IntMasterEnable();

    iParm = iTemp + 179;
    if (iParm < 2000)
        return 1;
    else
        return 0;
}
```

The code that updates `iValue` is a read-modify-write operation that makes one continuous critical section. The rest of the function only operates on local variables (including function arguments), so does not need to be protected.

RTOS tasks and semaphores

4) [30 pts] Trace the execution of the following multithreaded code fragment on a single-CPU system. The tasks Task1 and Task2 start in the **Ready** state at the beginning of their task functions. Stop when all tasks block. Complete the table below with the task and semaphore state changes, all steps numbered and labeled in the code. The semaphores are counting, initialized as shown in the table. (Note: The "wait list" column is the semaphore waiting list that may contain references to one or more tasks that are blocked on this semaphore.)

```
void Task1(UArg arg0, UArg arg1) {  // high priority task
①  while (1) {
②⑥⑦  Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);
        // Task1 code
    }
}

void Task2(UArg arg0, UArg arg1) {  // low priority task
③  while (1) {
④⑨    Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);
        // Task2 code
⑤⑧    Semaphore_post(semTask1);
    }
}
```

State changes are highlighted:

Step	Action	Task1	Task2	semTask1 count	semTask1 wait list	semTask2 count	semTask2 wait list
	Start	Ready	Ready	0	-	1	-
①	Scheduler runs	Running	Ready	0	-	1	-
②	Task1 pends on semTask1	Blocked	Ready	0	Task1	1	-
③	Scheduler runs	Blocked	Running	0	Task1	1	-
④	**Task2 pends on semTask2**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**
⑤	**Task2 posts to semTask1 (after "Task2 code" runs)**	**Ready**	**Running**	**0**	**-**	**0**	**-**
⑥	**Scheduler runs**	**Running**	**Ready**	**0**	**-**	**0**	**-**
⑦	**Task1 pends on semTask1 (after "Task1 code" runs)**	**Blocked**	**Ready**	**0**	**Task1**	**0**	**-**
⑧	**Scheduler runs**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**
⑨	**Task2 pends on semTask2**	**Blocked**	**Blocked**	**0**	**Task1**	**0**	**Task2**