UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 315 – Computer Interfacing

Final Examination

Instructor:	B. F. Cockburn
Exam date:	April 14, 2014
Exam duration:	120 minutes
Aids permitted:	A copy (paper or electronic) of the lecture slides can be freely consulted. Model solutions for the Winter 2014 assignments & midterm can be consulted. No other model solutions are to be consulted. No Internet access is permitted using any kind of device. Electronic calculators (all kinds) are permitted.
Instructions:	1. Enter your printed name, signature and I.D. number on this cover page. 2. Verify that this booklet contains 10 pages (including this cover page). 3. Neatly enter your answers in the spaces provided. 4. Use the reverse sides of the pages for extra space or rough work.

Student name: _____`Model`____,__`Solutions`____
Last name **First name**

Signature: ____________________________________

Student I.D.: ____________________________________

Question	Time	Worth	Mark	Subject
1.	15	12		Fundamental Concepts
2.	15	12		Timed Output Generation
3.	25	22		Multitasking Using MicroC/OS
4.	15	12		TCP/IP Stacks
5.	15	12		Flow Control
6.	20	18		Microcomputer Busses & DMA
7.	15	12		TCP/IP Networking
Total	**120 mins**	**100**		---

Question #1 (Fundamental Concepts)

In your own words briefly define each of the following concepts. Be sure to explain why each concept is important in embedded microcomputer systems.

(a) ColdFire Interrupt Priority Level

[3 marks] The interrupt priority level (IPL) is encoded in three bits IPL[2:0] in the Supervisor byte of the ColdFire's status register (SR). Each hardware interrupt signal in the microcomputer must be routed to one of the seven IPLs, from "001" up to "111". The IPL bits in the SR are automatically updated to record the priority level of the highest priority active interrupt. When no interrupts are active then IPL = "000". For IPLs "001" up to "110", interrupts of equal or lower priority are ignored (masked) until the current interrupt is serviced. For IPL "111", a new level 7 interrupt will interrupt the current level "111" interrupt.

[1 mark] The IPL allows the priority among the different possible interrupts to be enforced by the hardware so that the more important interrupts are serviced first before the less important interrupts.

(b) Task Control Block

[3 marks] The task control block (TCB) is a data structure that is used to store the context of a task when it is temporarily suspended from executing on the CPU. The task's context here is usually just the contents of the CPU registers. Later on, when the task is scheduled to run on CPU again, the context of the task is first restored to the CPU registers from the TCB. Then the task can resume executing from the same state from which it was suspended.

[1 mark] The TCB is fundamental to the context switching mechanism that is used to stop and restart tasks, a capability that is required to provide a multitasking software environment. Multitasking environments are used to facilitate the development of complex software sytems.

(c) Network layer protocol

[3 marks] The network protocol layer is concerned with the addressing of nodes and the routing of packets through a data communications network. In the Internet, the Internetworking Protocol (IP) performs the functions of the network layer.

[1 mark] The network layer is required to allow packets to be directed from the source node to the destination. IP is used to create a network layer across multiple networks.

Question #2 (Timed Output Generation)

(a) Embedded systems are often required to produce digital outputs that are updated repeatedly with precisely controlled timing. If one were using periodically scheduled state-driven code for two tasks (say TaskA and TaskB) with the same scheduling frequency, what design ideas could one apply to maximize the accuracy of the timing of the output signals that are updated by those two tasks?

[9 marks]
If TaskA and TaskB were blocked on two interleaved streams of hardware timer interrupts at the same required frequency, then the problem of maximizing the time accuracy would be decoupled. The software in each task could then be separately optimized to maximize the accuracy (minimize the jitter) of the outputs.

TaskA and TaskB could be merged into one task with the required scheduling frequency. This would allow the jitter between TaskA’s outputs and TaskB’s outputs to be more easily minimized. The software within that one task would have to be designed to minimize the jitter across all of the outputs.

If TaskA and TaskB are kept separate, and are blocked on the same hardware timer interrupts (with TaskA scheduled before TaskB), then the total execution time of TaskA would have to be kept constant to avoid introducing scheduling jitter affecting TaskB. Also, the software within the two tasks would need to be designed to minimize the output jitter within the tasks.

(b) How would one maximize the accuracy of the timing of all updated outputs if TaskA’s scheduling frequency were to be twice as fast as TaskB’s scheduling frequency?

[3 marks]
In this scenario, you could re-use many of the same ideas as proposed for part (a). You could have two tasks scheduled using two separate, hardware-generated streams of hardware interrupts. You could merge the two tasks into one task, and then use software to control when the two sets of outputs are uptdated. If the two tasks are going to share timer interrupts, the more frequently running task (say TaskA) should be scheduled to run first every time, so that it does not experience scheduling jitter. The second (say TaskB), would not see scheduling jitter as long as the execution durations of the other task (TaskA) were kept constant in software.

Question #3 (Multitasking Using MicroC/OS)

(a) Recall that in a pre-emptive multitasking software environment like MicroC/OS, the highest priority ready-to-run task must execute on the CPU. A consequence of this rule is that when a task calls MicroC/OS functions, it is possible that a context switch will occur to a new task. Consider each of the following functions, and for each, describe the conditions that would cause a context switch if the function were called.

- OSChangePrio()

  [3 marks] The function is called by a task to change its own priority to a different (and currently unused) priority. If the new priority is lower than another ready-to-run task (i.e., hi, then calling OSChangePrio() will cause a context switch to that other task.

- OSSemPost()

  [3 marks] The function is called by a task to cause a semaphore to be incremented by 1. If one (or more) tasks are blocked on that semaphore, then the highest priority task among them will be moved to the ready-to-run queue. And if that ready-to-run task is also higher in priority than the task that called OSSemPost, there will be a context switch to the newly ready-to-run task.

- OSSemPend()

  [3 marks] The function is called by a task to block that task if the initially encountered count of the semaphore is 0 or less. The OSSemPend function will decrement the function by 1, before the calling task is blocked (if the new count is negative) or allowed to proceed (if the new count is 0 or positive).

- OSUnlock()

  [3 marks] The function is called by a task to re-enable multitasking after a prior call by that same function to OSLock. If right after multitasking is enabled there is a higher priority that is ready-to-run (say one that became ready-to-run as a result of in interrupt), then there will be a context switch to that higher-priority task.

Question #3 (Multitasking Using MicroC/OS, cont'd)

(b) MicroC/OS provides a pre-emptive multitasking software environment, but it can be adapted to provide other kinds of multitasking. In the space below, briefly propose a design that effectively provides a cooperative multitasking environment using MicroC/OS for three application tasks, called TaskA, TaskB and TaskC. Compileable code is not required, but the essential aspects of your design must be clear. These three tasks must of course be assigned different priorities in MicroC/OS, but you are to use one or more semaphores and one or more additional tasks so as to allow the three application tasks to execute as if they were in a true cooperative multitasking environment. In your answer below, be sure to specify the relative priorities of any new tasks that your design might require.

[10 marks]

Whenever TaskA, TaskB or TaskC wishes to give up the CPU, it should pend on its own semaphore, say SemA, SemB and SemC, respectively. Right after the pend, the code should continue to the next block of code to be executed during the task's next interval on the CPU. These three semaphores must be initialized by the UserMain task to 0.

A new task, say TaskRef, must also be created, with a lower priority (higher priority number) than either TaskA, TaskB or TaskC. TaskRef will execute only when the other three application tasks are all blocked.

Task UserMain creates TaskA, TaskB, TaskC and TaskRef in the usual way, then either suspends itself or schedules itself to run very rarely. TaskRef is then the only application task that is running at this point. It then enters an execution loop where it posts to each of SemA, SemB and SemC in turn. After SemA has been posted by TaskRef, TaskA will be unblocked and will run for as long as it wishes. When TaskA is finished, it pends on SemA and gets blocked. TaskRef then resumes executing and immediately posts to SemB to unblock TaskB. When TaskB is finished, it pends on SemB and gets blocked. TaskRef then posts to SemC to unblock TaskC. When TaskC is finished, it pends on SemC and gets blocked. TaskRef then goes back to the top of its execution loop and posts to SemA to unblock TaskA, to repeat the loop of application tasks.

Question #4 (TCP/IP Stacks)

(a) NetBurner provides an implementation of a TCP/IP stack along with its port of the MicroC/OS kernel for the MOD5234 microcomputer. The stack is implemented using four interacting tasks. This might appear to be a rather complex solution. What would be the main advantages of the task partitioning that was used by NetBurner in their stack? Why would most of the disadvantages of NetBurner's four-task architecture be acceptable using a MCU like the MCF5234?

```
[4 marks] Advantages:
a) The software for the Ethernet driver, IP entity, TCP entity and webserver task can be decoupled. The four tasks can be designed, debugged, verified and modified separately.
b) Access to the frames, datagrams, segments, and byte streams can be controlled more strictly, enforced by task boundaries.
```

```
[4 marks] Disadvantages:
a) Passing the data packets across task boundaries is more complicated and might take extra time and storage space.
b) Developing optimized data structures for the data is trickier since multiple tasks must share those data structures.
```

(b) The software in a TCP/IP stack must have efficient access to different parts of the frame that is transmitted onto the communication medium. What strategy could you use in the software representation of the frame to ensure that each layer of software sees only the parts of the frame for which it is responsible, but no other parts of the frame?

```
[4 marks]
The frame could be partitioned into several separate buffers that are accessed by using separate buffer pointers. (This idea is used in the FEC block of the MCF5234 MCU.) In this way, each layer of software would only be given pointers to the parts of the frame that it needed to see and interact with, but not the other parts. The different buffers in the frame would be re-assembled into one buffer only when absolutely necessary.
```

Question #5 (Flow Control**)**

(a) Briefly compare and contrast the FILO stack and the FIFO queue. Give one example for each data structure that clearly illustrates the benefits of the data structure.

[6 marks]
The First-In Last-Out (FILO) stack is a data structure that provides temporary storage that exactly reverses the order of the data passing through. The main benefit is that the FILO provides an elastic memory that makes it easy to recycle memory locations after they are no longer required. A stack is implemented simply with one TOS pointer into the "top of the stack" entry in an array of elements. The TOS pointer is adjusted in one direction when new elements are added to the FILO, and adjusted in the opposite direction or old elements are deleted. An example of a stack is the stack frame that is used by microprocessors when exception handling routines are started. Also, stacks are used by compilers for passing parameters into subroutines and for creating space for local variables inside subroutines.

The First-In First-Out (FIFO) queue is a data structure that provides temporary storage that preserves the order of the data passing through. The main benefit is that the FIFO provides temporary storage that accommodates small short-term fluctuations in the data rate that is being received at the input and the data that is being withdrawn at the output. FIFO queues are widely used to decouple the successive processing stages in a communications connection.

(b) In most situations where a data transmitter is physically separated from the data receiver, it is not practical to use the hardware flow control method because the cost of using separate wires to carry the flow control signals would be too expensive. Flow control information must therefore be sent back using a shared communication channel that connects from the receiver back to the transmitter. In order to implement some robust form of flow control, why would timers be required in the transmitter and/or the receiver in the absence of hardware flow control?

[6 marks]
As soon as a shared communications channel is used to carry control commands over some distance through a communication network, it becomes much more possible for the control commands to get lost. Timers are used in the transmitter to determine whether or not the last command or data packet was received at the far end. A time-out can then trigger a retransmission to replace the lost command or data. Timers can also be used in the receiver, in some scenarios, for example, to replace lost "clear-to-sent" messages going back to the transmitter.

Question #6 (Microcomputer Busses & DMA)

(a) Most modern microcomputers use semisynchronous busses. Briefly describe what is meant by a semisynchronous bus. Use the system bus of the MCF523x family of MCUs to illustrate your answer.

[3 marks] A semisynchronous bus is a bus that has a clock signal as a basic time reference. Bus operations can take a variable number of clock cycles to complete, thus providing some flexibility in the duration of the bus operation.

[6 marks] The MCF523x system bus has a clock signal that is output by the MCU. The start of a bus operation is indicated by an active (low) pulse of the Transfer Start (TS*) signal produced by the CPU for the first clock cycle of the bus operation. The end of the bus operation is triggered when the Transfer Acknowledge (TA*) signal from the addressed device goes active low for a rising edge of the clock. The next clock cycle is then the last cycle of the bus operation.

(b) What is meant by single-address direct memory access? How is it possible for data to move from a source address to a destination address when only one address is supplied on the microcomputer bus? Use the MCF523x bus to illustrate your answer.

[4 marks]
Direct memory access (DMA) is a method for transferring blocks of data over the system bus of a microcomputer under the control of a DMA controller (DMAC) circuit. The DMAC must negotiate access to the system bus with the CPU and any other users of the system bus. DMA data transfer is fast and efficient because it avoids a large number of MOVE instructions, which would take time to decode and execute. A DMA transfer is like a large block move instruction that is executed by a separate dedicated CPU.

Single-address DMA is DMA from a range of addresses in a data buffer to (or from) a single memory-mapped register in a peripheral interface.

[5 marks]
An external device that wishes to start the DMA transfer asserts active (low) one of the DMA request signals (DREQn*). When the CPU is ready to start, it assert active (high) the corresponding DMA acknowledge signal (DACK) for a rising edge of the clock. The DMA transfer will start at the next rising edge of the clock. In single-address DMA, the transfer starts when Transfer Start (TS*) is pulsed low for a clock cycle. The data transfer occurs at the end of the clock cycle when the data memory asserts Transfer Acknowledge (TS*) is pulsed low. If the data port sizes are the same, the data flows directly between the data buffer and the peripheral register. If the data port sizes are different, the data must pass through buffers in the DMACn block.

Question #7 (TCP/IP Networking)

What would happen if a byte in the payload of a TCP segment were to be repeated twice as a result of a processing error in an intermediate node along the route from the source node to the destination node through a TCP/IP network? How could the error be detected by the IP entity in the next node that receives the IP datagram? How would the error be detected by TCP if the IP entities in all nodes did not detect the problem before the datagram reached the destination node?

[12 marks]

The byte duplication error in the TCP segment payload would certainly be detected and corrected by the Transfer Control Protocol (TCP) on an end-to-end basis. But it should also be detected by the Internetworking Protocol (IP) at the next node.

The IP header contains a 16-bit total datagram length field as well as a checksum for the IP header (not including the IP payload). The actual length of the received IP datagram can be recomputed and compared with the expected length. If the length is wrong, then the IP entity can simply treat the datagram as corrupted and then discard it. TCP would then detected the missing datagram, and would retransmit a fresh copy.

Now assume that the IP datagram reaches the destination node. The header of the enclosed TCP segment contains a 16-bit checksum for the entire segment, including the data payload. The recomputed checksum of the received TCP segment would be changed by the repeated payload byte, and so the recomputed checksum would be different from the expected checksum in the TCP header. TCP would then discard the entire segment, not increment the acknowledgement number, and wait for a retransmission of a fresh copy of the TCP segment from the other end of the connection.