UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CMPE 401 – Computer Interfacing

Final Examination

Instructor:	B. F. Cockburn
Exam date:	December 17, 2010
Exam duration:	120 minutes
Aids permitted:	A hardcopy of the course lecture slides can be freely consulted. Model solutions for the Fall 2010 assignments and midterm can be consulted. No other model solutions are to be consulted. One two-sided 8.5" x 11" formula sheet can be consulted. 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.	20	17		Multitasking Systems
3.	20	17		The Fast Ethernet Controller
4.	15	12		Message Queues in MicroC/OS
5.	20	17		Flow Control in TCP/IP
6.	30	25		Interfacing the MCF5234 bus to PCI
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 Computer Interfacing.

(a) Instruction Operation Code

[4 marks] The **operation code** (opcode) in a machine language instruction is a left-justified field in the instruction that determines the type of the instruction (e.g., MOVE, ADD, JMP). The opcode, which may vary in width for the different instructions, also determines the format for the rest of the instruction. The CPU uses the opcode to find all of the necessary operands, then performs the correct operation, and then write the results to the correct destination. The opcode is located in the first fetched word of the instruction to maximize the speed with which the CPU can interpret the instruction.

(b) Nonmaskable Interrupt

[4 marks] A **nonmaskable interrupt** (NMI) is an interrupt (usually driven by an externally-triggered hardware signal) that cannot be ignored using the interrupt mask in the CPU. When a NMI occurs, the CPU will stop what it currently doing and will begin to service the NMI interrupt. An NMI will thus override any other interrupt service routine that may be executing. In the 68000-class CPUs, which includes the v2 ColdFire, all level 7 interrupts are nonmaskable. NMIs will typically be reserved for emergency handling, such as a detected power failure or the detection of an imminent risk to safety.

(c) Critical Section

[4 marks] A **critical section** is a region of software that needs to be executed to completion without being interleaved with other code. Common examples of critical sections are (1) updates to data structures and (2) updates to hardware registers. It is important to use appropriate software techniques to protect critical sections so that data structures and/or system hardware is not corrupted by interleaved task execution or interrupt handling. Appropriate techniques include masking out all interrupts or protecting access to the critical section using a binary semaphore.

Question #2 (Multitasking Systems)

(a) Partitioning a software system into interacting tasks has both possible advantages and disadvantages. Briefly comment on the suitability of a multitasking software architecture for each of the following: (i) the ability to enhance an existing system design with new external interfaces; (ii) the ability to protect critical system data stored in memory; (iii) the ability to ensure that the system responds in a timely and predictable way to externally-generated signals; (iv) the ability to debug and then correct problems.

[8 marks]

(i) The multitasking architecture should simplify the addition of new interfaces. New interfaces can be controlled using newly introduced tasks that are relatively independent of the existing tasks. Alternatively, existing tasks might be modified to handle the new interface(s). Existing inter-task mechanisms can be re-used to ensure that the new task(s) fit in. Care will be required to assign priorities to any new tasks.

(ii) The multitasking architecture must be provided with mechanisms, such as binary semaphores or interrupt lock-out, that will allow critical sections to be protected. Such features are widely available in standard kernels, such as MicroC/OS. It is thus straightforward to protect critical system data using known technical solutions.

(iii) Externally-handled signals can be handled satisfactorily in a multitasking system using interrupts and/or task priorities. The most time-critical signals should be handled by interrupts; high-priority tasks can also be used as appropriate. Widely available multitasking kernels, such as MicroC/OS, provide all of the necessary features.

(iv) Although the modularity provided by tasks can help, debugging problems in a multitasking system can be difficult since it is frequently difficult to recreate the situations when problems were encountered. It may be necessary to add temporary software features to record the precise sequence of key events that led up to the problem. Assertion checks may also prove useful. It may also be the case that debugging and problem correction is simplified in a multitasking system since the problem may be more easily isolated to one or a small number of tasks.

Question #2 (Multitasking Systems, cont'd)

(b) In many process monitoring and control applications, it is convenient to adopt a multitasking architecture in which the tasks are expressed using a collection of interconnected states. Each task state is implemented as a (usually short and simple) segment of software. Once the present state of a task is executed, the task gives up the CPU. The next state of a task may be fixed for each possible current state, or may depend on the values of one or more status conditions. Discuss the likely advantages and disadvantages of such an architecture in the following two applications: (i) a controller must take measurements of many different process parameters, and the measurements might need to be performed at different frequencies for different parameters; (ii) a controller operates an Internet router, responding to the arrival of data packets, while also operating a user interface.

[9 marks]

(i) The state-driven multitasking architecture has many advantages for this particular application. The sequence of measurements of each parameter can be taken care of using states in separate tasks that are scheduled to execute at the desired different measurement frequencies. There should be no major disadvantages to the architecture since fast real-time response to inputs is not mentioned in the specifications.

(ii) The state-driven architecture is not especially well suited for this application. While the user interface can be handled satisfactorily using polling controlled by periodically-scheduled state-driven tasks, handling the very rapid arrival of data packets will be more difficult with such a task architecture. The handling of data packets, both the receive and transmit directions, is likely best handled using interrupt-driven software and direct memory access (DMA) rather than by the state-driven tasks.

Question #3 (The Fast Ethernet Controller)

(a) The Fast Ethernet Controller (FEC) in the MCF5234 microcontroller is a subsystem that simplifies interfacing to 10-Mbps and 100-Mbps Ethernet LANs. The FEC includes shares a FIFO and a DMA controller for the receive and transmit directions. Briefly explain the reasons why both a FIFO and DMA controller are required in the FEC. What would happen if either the FIFO or the DMA controller were to be omitted?

[8 marks]

The first-in first-out (FIFO) buffer is very useful since it helps the FEC to handle the shorter-term burstiness of data that is transferred into the FEC in both the transmit (from the CPU side) and receive (from the line side) directions.

The DMA controller is useful since it allows more efficient transfer of data between system memory and the FEC's FIFO buffer. Efficiency is higher because DMA provides block moves that require fewer clock cycles to move the same volume of data compared to a series of CPU move instructions. If the DMA controller were to be omitted, then the system bus would have to be busier to keep up with the Ethernet transfer rate.

If the FIFO were to be omitted, then there would be very little timing flexibility when data is transferred in constant bitrate bursts between the high-speed Ethernet medium and the system memory. In the receive direction, the DMA would need to be immediately ready to accept data from the Ethernet and to transfer it from the FEC to the CPU's memory. This is difficult to guarantee since the FEC DMA controller would need to request and compete for the use of the system bus. In the transmit direction, a FIFO makes it much easier to ensure that the outgoing data bytes are available to send out at the Ethernet line bitrate. If the CPU side cannot keep up, the FIFO buffer would empty out prematurely and the Ethernet packet would either have to be abandoned or sent out in a truncated size. Without the FIFO in the transmit direction, then the CPU must keep up rather rigidly with the data being transmitted onto the Ethernet.

(b) There is a hard-coded controller inside the FEC that interprets the fields of the transmit and receiver buffer descriptors (BDs). The BDs are used to point to transmit (Tx) and receiver (Rx) buffers. The Tx buffers are variable in size, whereas the Rx buffers are all of the same size. What would be the main advantage(s) of expressing transmitted Ethernet frames as a series of variably-sized buffers? What would be the main advantage(s) of loading received Ethernet frames into a series of fixed-size buffers? Why should the Rx buffers be of some fixed size?

[9 marks]
Expressing the transmit-direction Ethernet frames in a series of variably-sized buffers would have the advantage of allowing the different headers of the frame (e.g., Ethernet, IP, TCP, application-level headers, payload data, any trailers) to be loaded into separate buffers, possibly by different pieces of software. This would support the partitioning of the communication interface into relatively independent layers.

In the receive direction, some aspects of the frame structure (e.g., the Ethernet, IP and TCP headers) could be anticipated to allow each part to be stored in variably-sized frames, but this would violate the principle of avoiding the exploitation of higher-layer details by lower-level layers. Also, the length and structure of the payload is difficult to predict, and so determining the best buffer sizes to use for each received packet would be difficult or at least inconvenient. The cleanest solution is of the FEC to use constant-size buffers in the receive direction. Constant-size buffers have the additional benefit of being easier to allocate and recycle by the require buffer management system.

Question #4 (Message Queues in MicroC/OS)

The message queue in MicroC/OS is a flexible feature that allows messages to be passed in a controlled way between user tasks. Consider the following three code fragments, which appear in UserMain task, a first user task A, and a second user task B:

```
// Fragment in UserMain
OS_Q MyQueue;
void * MyQueueArray[ QSIZE ];
OSQInit( &MyQueue, MyQueueArray, QSIZE );

// Fragment in Task A
char * Pmsg;
Pmsg = "Hello World!";
OSQPost( &MyQueue, (void *) Pmsg );

// Fragment in Task B
char * Pmsg;
BYTE Err;
Pmsg = (char *) OSQPend( &MyQueue, 0, &Err );
```

(a) Briefly describe the difference between data structures MyQueue and MyQueueArray. Why are both data structures required? Why are the details of the OS_Q type kept private hidden inside MicroC/OS? What information must be maintained inside MyQueue?

[6 marks] MyQueue is a data structure of the hidden data type OS_Q. Presumably it keeps track of the tasks that are pending on the queue. The queue data itself, which is a fully public array of void pointers, is stored as array data structure MyQueueArray.

The details of OS_Q are kept private to preserve de-coupling between the MicroC/OS kernel and the application software. OS_Q can then be changed/updated more easily by the owners.

MyQueue does not actually hold the pointers in the queue, but it holds all other information concerning the queue, such as the queue of tasks that are blocked on the message queue.

(b) In the code fragment from Task B, briefly describe what happens when the third line of code (the call to OSQPend) is executed. Be sure to justify all parts of this particular line of code.

[6 marks] OSQPend is called by Task B to pend, and possibly block forever, on the message queue at address MyQueue, which is the first parameter. The second parameter, which has value 0, allows Task B to be blocked forever on the queue. The third parameter is the address of BYTE variable Err, which is loaded with a return code for OSQPend by the time the function has returned to Task B. The return value of OSQPend is a pointer to a void, but the syntax "(char *)" casts the type of the pointer to a pointer to an array of char's, and then assigns that pointer value to pointer variable PMsg.

Question #5 (Flow Control in TCP/IP)

(a) The Transmission Control Protocol (TCP) uses a window-based flow control method, which simultaneously solves several challenges of operating a data connection over a potentially unreliable network. TCP provides a reliable bidirectional byte transportation service between a transmitting node and a receiving node. Briefly explain how TCP would handle the situation where the arrival order of two IP datagrams is reversed in the receiving node compared to the order in which the two datagrams were transmitted.

[10 marks]
TCP uses 32-bit sequence numbers to keep track of the order of transported TCP segments. The ascending sequence numbers are actually associated with the payload bytes, and the sequence number of a segment is simply the sequence number of the first payload byte in that segment. The range of sequence numbers is large compared to the maximum size of a segment, so it is easy for a receiver to determine if two received segments have been received in reversed order: the sequence numbers will appear to reduce unexpectedly (not due to address wrap-around) for the premature segment, and then will increase again for the delayed segment.

The receiver uses a FIFO buffer to store the bytes as they are received in arriving segments. The segment sequence number determines where in the FIFO the corresponding payload bytes should be written. If two segments are received out of order, then the receiver may still have loaded the bytes correctly into the FIFO (depending on the addressing method that it uses). As long as no segments are completely lost, then all of the bytes could still be written into the FIFO, and can thus be read out of the FIFO and passed along for further processing. In such a case, the receiver may carry on with no further action (apart from a possible delay in passing along the bytes from the segment that was received out-of-order).

The receiver might also choose to refuse to acknowledge the two out-of-order segments. After the time-outs have expired for those packets, they will be retransmitted and will hopefully be received in the correct order.

Question #5 (Flow Control in TCP, cont'd)

(b) Now explain how TCP would recover from the situation where exactly one byte is repeated twice (by mistake) in the payload of an IP datagram while it is passing through a network node somewhere along the route from a transmitting node to a receiving node.

[7 marks]
According to the TCP prototocl, the TCP entity in the transmitter computes a checksum over each TCP segment and places that checksum in a field in the TCP header. The TCP entity in the receiver recomputes the checksum of each arriving segment and compares that checksum with the checksum that is in the TCP header. If the two differ, then the TCP entity will withhold acknowledgement of the segment. This refusal will cause a time-out in the transmitter, which will cause the same segment to be retransmitted.

If one byte in a segment is repeated twice, then several things can go wrong. The problem will almost certainly be detected when the segment checksum is verified in the receiver. The total length field in the IP datagram header will also be incorrect. This error will likely cause the next IP entity to discard the datagram, which will eventually trigger a transmission of the same IP datagram (and enclosed TCP segment).

Question #6 (Interfacing the MCF5234 bus to PCI)

There are many similarities between the external MCF5234 system bus and the PCI bus. For example, both busses use active low byte enable signals. However, the two busses have many differences that would need to be overcome if they were to be interfaced together. Consider the problem of designing an interface circuit that would allow an MCF5234-based bus to operate in a PCI-based system. In particular, briefly describe in words how the PCI signals "IRDY", "TRDY", "FRAME", "DEVSEL" and "AD31-0" could be interfaced with the available MCF5234 bus signals (CS, R/W, TS, TIP, TA, TSIZ, etc.) Consider only the case where the ColdFire CPU in the MCF5234 on one PCI board needs to read a memory location (byte, word or long word) that is located in a second PCI board.

[25 marks]
IRDY\: The initiator ready signal is asserted active (low) for the rising clock edges when the CPU is ready to latch the data off of the data bus. This time occurs when MCF5234 signal OE\ is low and TS\ is high.

TRDY\: The target ready signal is asserted active (low) when the second PCI board asserts the TA\ signal active (low). Thus TA\ can be sent back to the CPU as MCF5234 signal TRDY\.

FRAME\: The FRAME\ signal is identical to the MCF5234 TIP\ signal, except that FRAME\ is de-asserted (high) before the last rising clock edge that has a data transfer. A state machine would need to be designed that monitors the TSIZ\ and BS\ signals to determine whether or not the last bytes are being transferred: When the last bytes are being transferred at the next rising clock edge, then FRAME\ should be forced high; otherwise, FRAME\ should be forced low.

DEVSEL\: The device select signal should be pulled active (low) when a corresponding chip select CSn\ is active (low). One could simply OR together the chip select signals to opain DEVSEL\ on the CPU board.

AD31-0: The multiplexed address-data lines on the PCI bus carry the address for one clock cycle. A register is required to latch the MCF5234 address when TS\ is low and the clock goes low. This address needs to be driven onto the PCI AD bus at that time. When TS\ is high and the clock goes low again, the latched address should stop being driven onto the AD bus. The CPU board should tristate the AD bus drivers to allow the other PCI board to drive the data onto the AD bus.