UNIVERSITY OF ALBERTA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 315 – Computer Interfacing

Final Examination

Instructor: B. F. Cockburn
Exam date: December 18, 2012
Exam duration: 120 minutes
Aids permitted: A copy (paper or electronic) of the lecture slides can be freely consulted.
Model solutions for the Fall 2012 assignments and 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.	17	14		The Hardware-Software Interface
3.	22	18		Software Architecture
4.	22	18		Computer Communications
5.	26	23		Interrupt-driven I/O
6.	18	15		Microcomputer Busses
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) FIFO buffering combined with flow control

[4 marks] FIFO buffering involves temporarily storing a stream of data in a memory that enforces the first-in first-out data storage order. A FIFO buffer placed between a data producer and a data consumer allows short-term mismatches in data rates to be safely accommodated without changing the data order and without requiring communication between the data producer and consumer. Flow control is used to avoid long-term mismatches in data rates. A data consumer process sends feedback information to a preceding data producer that causes the data producer to slow down (or speed up) or even to temporarily stop (or restart) its production of new data. Both FIFO buffering and flow control are required together to safely decouple the data rates of a pair (or a pipeline) of communicating systems.

(b) End-to-end protocol

[4 marks] An end-to-end protocol is a protocol that involves the exchange of communication control information between the source and destination nodes of a communication path through a network. The intervening nodes in the path do not participate in an end-to-end protocol; their role is restricted to forwarding the end-to-end protocol information to the two end nodes. An end-to-end protocol is employed by high-level layers in a layered communication protocol. For example, the session layer and the transport layer are end-to-end protocols that need only involve the source and destination nodes.

(c) Memory-mapped peripheral

[4 marks] A memory-mapped peripheral is a sub-system in a microcomputer that is accessed by the CPU over the system as a set of registers that have been assigned memory addresses. The registers in the peripheral can be conveniently written and read just like memory registers. However, unlike memory registers, the flip-flop states can have side effects on the operation of the peripheral and/or the operations of the peripheral can change the values of the flip-flops. Memory-mapped peripherals are convenient because they can be accessed just like memory locations without requiring special CPU instructions and without requiring special ports to the CPU. Access to peripheral registers can be restricted in the same way as access to memory locations.

Question #2 (The Hardware-Software Interface)

(a) The interface between the software and hardware of an embedded computer system must be designed to manage conflicting priorities. The specific features of the hardware need to be accessed directly and efficiently to maximize performance. However, it is also important to control software development costs and to maximize portability across different potential hardware platforms. Briefly explain why using the C programming language and the MicroC/OS kernel, and minimizing the use of assembly language programming, would be a suitable initial strategy for managing the design of the hardware-software interface.

[4 marks] Designing the system's software in C has multiple benefits. C is supported by efficient compilers for virtually all computer architectures. Using C for most software will ensure high portability as well as efficient compilation to the underlying machine language. Assembly language should be avoided as much as possible since it is not portable and is expensive to develop and maintain.

[6 marks] Using MicroC/OS (or another similar C-based kernel) would be an excellent strategy for building the software of an embedded system. MicroC/OS is implemented mostly in C (with only minimal assembly language), and so the kernel software can be readily recompiled for new architectures. The hardware-dependent parts of MicroC/OS are contained to a relatively small "port". MicroC/OS is scalable, and so the software size can be scaled back to include only the required features. MicroC/OS has fast and deterministic response and is thus suitable for hard real-time and other less demanding applications.

(b) Briefly describe what a "device driver" is and what it is used for. How can the design of device drivers be constrained so that software portability is maximized while still ensuring satisfactory access to the underlying hardware?

[4 marks] A device driver is a piece of software that is used to operate a peripheral subsystem. In the case of device drivers for communications interfaces, it is helpful to generalize and standardize the CPU/software side of the interface so that multiple kinds of communication hardware (say for accessing different kinds of communication media) can be accessed in the same way. The driver software should be written in C (to simplify compilation to different CPUs) and should make minimal use of assembly language.

Question #3 (Software Architecture)

(a) In some embedded computing applications it is clearly not appropriate to use a kernel or operating system since that would be overly complex. One could instead use a foreground-background software architecture, with interrupts employed to ensure fast response for time-critical events. But as the complexity of the application increases, what problems might occur if one used a foreground-background system? What properties would be important to have in a kernel to allow it to be considered as an alternative to foreground-background systems in the less complex applications?

[4 marks] The complexity of the application would increase as the number of foreground and background activities increased in number and in computational complexity. The foreground-background architecture would continue to give priority to the foreground activities, but it would become difficult to manage the sharing of foreground time and background time. Interrupts impose a rigid preemptive priority among the foreground activities. Also, the number of hardware interrupts may be rather limited. The background time would be hard to share in any way other than with a strictly round-robin approach, which would become increasingly sluggish as more activities were to be included in the same background loop.

[1 mark] A feature-rich kernel, like MicroC, would still be a viable alternative if it could be scaled down in complexity and cost to match the needs of the less complex application.

(b) Some multitasking kernels, including MicroC/OS, allow a task to change its priority to a higher or lower priority. Before acting on a request to change a task's priority, what would the kernel need to check first? If the kernel decided to go ahead and change a task's priority, what additional task-related actions might need to occur as a result?

[5 marks] The kernel would need to check that no other task already exists with the requested priority. If none exists, then the calling task's priority could be safely changed; otherwise, a failure return code should indicate failure to change the task priority.

If the calling task's priority is successfully changed, then the new priority might be lower than the priority of some other ready-to-run task, in which case the kernel will cause a context switch to that other task.

Question #3 (Software Architecture, cont'd)

(c) There are several calls to MicroC/OS functions that can cause the currently running task to be blocked. Briefly explain the conditions that would lead to blocking when each of the following functions is called. For each call, indicate how the same task could later on become unblocked:

- OSTimeDelay()

  [2 marks] This function blocks the calling task if the specified time delay is greater than 0 "ticks". In that case the calling task is blocked and a timer is set that will move the blocked task to the ready-to-run queue once the specified time delay has elapsed. The task will then become resume executing as soon as it becomes the highest-priority ready-to-run task.

- OSQPend()

  [2 marks] This function blocks the calling task if there are no unread pointers stored on the given pointer queue. A blocked task will be moved to the ready-to-run queue once a pointer has been "posted" to the queue when it happens to be the highest priority task blocked on the queue. In addition, the task will be moved to the ready-to-run queue if a user-specified time-out expires.

- OSTaskCreate()

  [2 marks] This function preempts the calling task if the created task has a higher priority. However, the calling task is not blocked in this case: it is merely moved to the ready-to-run queue.

- OSTaskDelete()

  [2 marks] This function deletes the calling task. It is not really blocked; rather, it is put into a dormant state from which it can be recreated later on by another task using the function OSTaskCreate().

Question #4 (Computer Communications)

(a) Briefly explain the primary roles of the MAC and the PHY in the hardware implementation of a computer communications interface. What are the main advantages and possible disadvantages of enforcing a partitioning between the MAC and PHY?

[2 marks] The Media Access Control (MAC) layer implements the protocol for acquiring access to the communication medium to allow communication from the current node to the next node connected to the same medium. This is governed by the so-called "Data Link" protocol.

[2 marks] The PHY transfers bits and data packets to and from the analog communication medium as analog signals.

[2 marks] Enforcing a strict partitioning between the MAC and PHY, say according to an industry standard, would allow the design and manufacture of the MAC and PHY to be conveniently decoupled from each other and optimized independently. However, some opportunities for closer interaction and joint design optimization (e.g., integration onto one system-on-a-chip) might be missed by enforcing the MAC-PHY partitioning.

(b) What is the relationship between a TCP/IP connection, a Unix socket, and a client-server interaction? Your answer should define what each of these concepts means.

[2 marks] A TCP/IP connection is an effective bidirectional "byte pipe" between two TCP entities on two nodes (but possibly on the same node) of a computer network. TCP is a transport-level protocol that creates the virtual connection using a bidirectional flow of IP data packets between the two nodes. TCP ensures that the data streams are reconstructed reliably despite lost, duplicate and/or out-of-order packets.

[2 marks] A Unix socket is a data structure that is wrapped around a TCP/IP connection to allow that connection to be read and written as easily as with a regular file. A socket thus simplifies the design client-server interactions in a Unix environment.

[2 marks] A client-server interaction is a conveniently high-level mode of interaction between tasks or processes on two nodes in a network. On one node the server process stores information and/or performs services for client processes on the network, but it only does so in response to calls from those clients. Thus a server must wait passively for calls from clients before entering into a client-server interaction. A client process initiates an interaction with a server process on another node (but possibly the same node).

Question #4 (Computer Communications, cont'd)

(c) The Lightweight IP (lwIP) stack is an implementation of TCP/IP that was intended for small embedded systems. Briefly describe the major features in the software architecture of lwIP that maximize its portability and efficiency across different embedded systems.

[6 marks]

The architecture of lwIP partitions the software between a generic TCP/IP stack core buffered with adaption layers that deal with variations in (1) the underlying communications hardware, (2) the host's operating system, and (3) the application software than needs to use the TCP/IP stack.

All elements of lwIP are implemented in C, which is a highly portable programming language with broad compiler support.

Adaptation layer (1) is a series of device drivers that make the underlying communication hardware look the same by having the sample access functions.

Adaptation layer (2) is the so-called OS Emulation Layer. It accommodates differences among host's operating system. Only minimal assumptions are made about the operating system. For example, lwIP uses its own buffer management system instead of relying on buffers provided by the operating system.

Adaptation layer (3) is provided as an Application Program Interface (API) that provides a set of function calls that implement sockets for the high-level applications.

Question #5 (Interrupt-driven I/O)

Consider the design of an interrupt-driven software driver for an output interface. A function TX_BUF is to be designed that is to be called whenever a task wishes to transmit the contents of a 16-byte data buffer. The peripheral hardware can be enabled by setting bit TX_IRQ_EN to 1 to produce a hardware interrupt whenever the transmit data register has been emptied and can now be written with the next byte of data to be transmitted. Transmit interrupts are turned off by clearing TX_IRQ_EN to 0. The transmit interrupt status bit TX_IRQ is set to 1 automatically by the peripheral hardware whenever the byte in the transmit data register TX_DR has been transmitted and the next byte can be written to the TX_DR. The TX_IRQ bit is cleared to 0 by writing TX_IRQ to 1. The software driver contains a 16-byte data buffer that is intended to hold data obtained from the calling task as an input parameter to TX_BUF. The software driver also contains a semaphore, which is not visible outside the driver, that can block the calling task until the buffer is fully transmitted.

(a) Using words and/or in pseudo-code in the space below, design the algorithmic structure of the function TX_BUF that is called by tasks to transmit the contents of a new 16-byte buffer. The function should hide from the calling tasks the presence of the internal data buffer and the semaphore. Also specify how the various variables and hardware would need to be initialized.

[10 marks]

```
function TX_BUF( dataBuf byte [16] )

   copy all data from dataBuf to internal 16-byte buffer;

   clear byte counter variable BYTE_CNT to 0;

   clear TX_SEM semaphore to 0;    /* precaution */

   enable interrupts by setting TX_IRQ_EN to 1;

   pend on semaphore TX_SEM;

end function TX_BUF;
```

Question #5 (Interrupt-driven I/O, cont'd)

(b) In the space below enter your high-level design (using English description and/or pseudocode) for the interrupt service routine (ISR) that would accompany the function designed above. As well as correctly operating the transmitter hardware, the ISR is to count the number of bytes and buffers that have been transmitted.

[13 marks]

```
routine TX_ISR
   write TX_IRQ to 1 to clear the interrupt;
   if BYTE_CNT < 16
      increment BYTE_CNT by 1;
      copy next byte from data buffer to TX_DR;
   else  /* BYTE_CNT == 16 */
      disable interrupts by clearing TX_IRQ_EN to 0;
      post to semaphore TX_SEM;
   end if;
return_from_exception;
```

Question #6 (Microcomputer Busses)

(a) Briefly explain what is mean by Direct Memory Access (DMA)? What the major benefits and possible disadvantages of using DMA?

[8 marks] Direct Memory Access (DMA) is a mechanism that is provided in many computer systems to speed up the transfer of blocks of data over the bus. A CPU would need to execute many MOVE instructions to transfer a large block of data over the bus. Instead, a DMA controller 9DMAC) shares the bus with the CPU and independently produces the bus signals that cause the data block to be moved from source to destination. The DMAC contains registers that must be initialized by the CPU. The DMA transfer can be started either by software or a hardware signal. The end of the DMA transfer is often signaled by an interrupt back to the CPU.

Major benefits of DMA: Faster data transfer. CPU is freed up to do other tasks during DMA transfer.

Major disadvantages of DMA: DMA transfers must be coordinated in some way with cache memory and virtual memory so that only updated data is transferred. This coordination requires additional design complexity.

(b) The PCI bus is an example of an expansion bus for PCs. Like most modern microcomputer busses, PCI uses a semisynchronous bus protocol. Briefly explain what this means. Describe the PCI bus signals IRDY and TRDY in your explanation.

[7 marks]
A semisynchronous bus protocol is a bus protocol that uses a clock signal in the bus to provide flexible timing for operations (e.g., read and write) that occur over the bus. Bus operations take a variable number of clock cycles to complete, and this timing flexibility allows peripheral devices of different response times to be accommodated.

IRDY (Initiator Ready) is an active low timing signal that indicates that the initiator (e.g., the reader or writer) of the present bus operation is ready to do the data transfer at the next rising edge of the clock.

TRDY (Target Ready) is an active low timing signal that indicates that the target (e.g., the read or written device) is ready to do the data transfer at the next rising edge of the clock.

The next data transfer (e.g., read or write) occurs at the next rising edge of the clock when both IRDY and TRDY are low in voltage (i.e., active).