ECE 315 Assignment #1 Solutions

ECE 315 - Computer Interfacing

Assignment #1 Solutions

Due: In the ECE 315 assignment box at 15:45 on Wednesday, Feb. 5, 2020

---

1. Briefly explain the similarities and differences between a hard real-time embedded system and a soft real-time embedded system. Why is it useful, when selecting an implementation strategy, to distinguish between hard and soft real-time embedded systems?

   [8 marks]
   Both hard and soft real-time embedded systems are **similar** in that they are designed to interact effectively with on-going changes in their operating environment; specifically, the surrounding system that a real-time embedded system controls must be controlled correctly, safely and efficiently as it interacts with its environment. Real-time embedded systems must meet real-time constraints: in other words, they must respond promptly and predictably to inputs, they must record changes in the system state, and they must appropriately update any outputs within specified timing constraints.

   Hard and soft real-tme embedded systems are **different** in that in the case of a **hard real-time system**, violations of the real-time constraints are unacceptable since those violations could cause injury to humans and/or serious losses. In the case of a **soft real-time system** occasional minor violations of the real-time constraints could be tolerated.

2. Using the latest, fashionable technical jargon is an inexpensive way of appearing technically "up-to-date" to your colleagues, your managers, and your customers; however, such terms are sometimes actually quite poorly defined. Consult reputable Internet resources and provide working definitions for the following four computing architectures: cloud computing, edge computing, fog computing, and mist computing. For each term provide your assessment of how well-defined each of the terms is. Also for each term give the advantages that the computing architecture is supposed to provide over existing practice.

   [12 marks]

   - **Cloud computing:** Cloud computing is computing using rented computing resources (computer servers, memory, hardware accelerators) that are accessed over the Internet. The resources are rented by customers as required, say by the minute; customers can thus avoid many of the costs of ownership, and can gain access to computing resources at a relatively low price. The rental price can vary depending on the time of day, and this gives customers the option of waiting for a lower price if they are able to rent the computing resources during low-demand, lower-cost times. In addition, cloud computing resources are generally scalable in size, so a customer can rent more parallel computing resources at a time to get faster computation times. The precise geographical location of cloud computing resources is not generally disclosed to customers; the location is usually of no concern to the customers. The term "cloud computing" is now very well established.

   - **Edge computing:** Edge computing is computing using Internet-connected resources that are located geographically close to the user(s) to minimize the response time and the required communication resources for a demanding application. Edge computing nodes are connected to the rest of the Internet using broadband connections. Edge computing is a relatively well established concept.

   - **Fog computing:** Fog computing is computing that distributes the computation flexibly over computing resources that are located in the Internet core (like cloud computing) and resources that are located out at the Internet edge (like edge computing). Fog computing is supposed to offer the best of all distributed computing models, but it is likely to be complex to manage efficiently. Fog computing can using resources at the edge to provide faster response times to users, but large-scale computing can be moved to the core where there may be more available resources at lower cost. The term "fog computing" is still very new, and is indeed an active area for research.

   - **Mist computing:** Mist computing is computing using smart "Internet of Things" devices, which communicate directly with each other connections (often using wireless connections).

Mist computing can proceed without the need for a connection to the rest of the Internet. In addition, mist computing can in some cases provide faster response with less energy consumption because the computing is done as locally as possible. However, some IoT devices are battery-powered and do not provide the fastest computing resources or the largest memory capacity; these resource limitations can rule out the use of mist computing in some applications. The term "mist computing" is still quite new and not as well defined as, say, cloud computing.

3. There are now many situations where embedded computers provide assistance to human operators. Most typically, however, the actions of the human operator must be given priority over the recommended actions of the embedded systems. However, in some situations it may be warranted to allow the embedded system to lock out and override the control of the human operator. Give three distinct examples of such situations, and for each explain why it would be reasonable for the embedded computer to have priority over the human operator.

   [9 marks]
   Authorized human operators should always have ultimate control of an embedded system, but there are some special situations where an embedded system could reasonably lock out the control of human operator. Here are three possible examples:

   - A password-protected embedded system is required to ignore humans who fail to provide the correct authentication. In this example the embedded system has a clear mandate to enforce security, and ignoring unauthorized potential operators is part of that mandate. An authorized human operator could still gain control of the embedded system using the password for a special "administrator" account.
   - A vehicle may be outfitted with a breathalizer, whose pass/fail result is be passed to the ignition system so that the car cannot be started if the human driver is impaired. In this example, the embedded system was clearly given the mandate to prevent the operation of a vehicle by an impaired operator, thus protecting life and property.
   - In cases where the human operators are deemed to be attempting to deliberately cause loss of life or property (e.g., deliberating crashing an airliner, as in the case of GermanWings Flight 9525 on March 24, 2015) it would be reasonable to allow an embedded system to lock out the pilot and take control to avoid a crash if a crash would otherwise be inevitable.

4. Briefly describe the life cycle of a properly handled pointer value in C. Begin by explaining the different ways in which a pointer value is correctly created. Then explain the various ways in which a pointer value can be correctly modified (not just overwritten). Finally, explain when it is safe to discard the last copy of a pointer value.

   [21 marks]
   The uninitialized pointer value NULL: After a pointer variable is first instantiated, and if that variable is not immediately assigned the value of a reference (i.e., the base address of an object), then that pointer variable should be assigned the standard uninitialized pointer value NULL. This allows potential users of the pointer variable (or its contained value) to check whether or not the pointer has been assigned a non-NULL reference value.

   Creating a pointer value using the reference operator &: The reference operator **&** is one correct way of obtain the reference (that is, the base address in the memory map) for a given object (for example, a variable, a character, a string, an array, a function). Such a reference is one way of creating a valid pointer value. In the case of an array, it is possible to use the reference operator to find the reference to one element of an array. For example, **ptr = &array_name[3];**

   Creating a pointer value by modifying an existing pointer value using addition or subtraction: If a pointer has been initialized to a reference to an element in an array, then the resulting pointer value can be changed to a new pointer value by addition (**+** or **+=**) of an integer value, incrementing (**++**) by one, subtraction by an integer value (**-** or **-=**), or decrementing by one (**--**). In all of these cases, the integer value is automatically multiplied by the size of the array object so that the resulting pointer value stays aligned with the base address of an object in the array. There is no built-in mechanism that prevents a pointer from being incremented (or decremented) beyond the end of an array. In such a scenario, incorrect data could be read from outside the array, or information in memory (data or code) could be overwritten.

   Creating a pointer value without using the reference operator &: As a result of compiler and assembler conventions, the name of an array (or function) has a pointer value that is identical to the a reference to the array (or function). In these two cases the reference operator **&** is not required to create a pointer value. For example, **array_ptr = array_name;** and **array_ptr = &array_name;** and **array_ptr = array_name[0];** have the same meaning. Pointer values can also be allocated for hardware addresses in the memory map, such as the addresses of peripheral registers. Hardware addresses can be provided using either a numerical memory address or (preferably) a meaningful symbolic name from the source code that specifies the memory map.

Creating a pointer value that identifies a dynamically allocated block of memory: A pointer value can be created by calling the dynamic memory allocation function malloc(). This function creates a valid pointer value. (Note in C there also the functions calloc() and realloc(), which also produce pointers to the base address of a dynamically allocated block of memory.)

Receiving a pointer value as the return value of a function: Pointer values can be the return value of a function. But ultimately one of the other methods must be used to create a valid pointer value within the function.

Avoiding dangling pointer values: Pointer values for software objects must not be used after the allocated block of memory is recycled, say by calling the free() function. Such a pointer value is called a dangling pointer. Dereferencing a dangling pointer could easily cause a serious system failure. As soon as the value of a pointer variable becomes a dangling pointer, the pointer variable should either be loaded with the uninitialized pointer value NULL or it should be loaded with a valid new pointer value.

5. In the context of microcomputers, what is meant by a "memory leak"? How can using software objects be used to help ensure that memory leaks and initialization errors are avoided in embedded systems?

   [10 marks]
   A **memory leak** is the situation where regions of memory become unavailable because they were allocated dynamically from the heap at one time, but then never recycled properly so that they could be re-allocated and re-used. One simple source of memory leaks is the failure to call the function (e.g., free(ptr) in C) that informs the dynamic memory allocation system to recycle the chunk of previously allocated memory pointed to by the passed pointer value (e.g. "ptr" in the case of "free(ptr)").

   Software objects provide **constructor** functions (for correctly creating a new object instance), **copy constructor** functions (for correctly creating a separate copy of an existing object instance), and **destructor** functions (for correctly recycling an object instance that is to be deleted from the system). The constructor function for an object class is the place for code that calls the dynamic memory management system so that it correctly allocates any dynamic memory that is required to implement a new instance of the object. The copy constructor function for an object class is the place for code that calls the dynamic memory management system so that it correctly allocates any dynamic memory that is required to implement a new copy instance of an existing object. If the original object had dynamically allocated memory, then the object copy will also have the same amount of newly dynamically allocated memory, and no pointer values will be shared between the original object instance and the copy. The destructor function contains code that correctly recycles any dynamically allocated memory that was associated with the object when the object was first created.

   When an object is declared within a new context (e.g., within a context defined by curly braces), the corresponding constructor function will be called automatically to create a new object instance. When the context is exited, the corresponding destructor function will be called automatically to recycle any memory that was dynamically allocated to the object when it was constructed. Similarly, when an object is passed as a parameter into a C function, the corresponding copy constructor function will be called to create a new instance of the object that will be used within the context of the function. When the function is exited, corresponding destructor function will be called to correctly recycle any memory that was dynamically allocated to the object copy. In this way automatic function calls help to avoid memory leaks associated with objects.

6. What is meant when it is said that an embedded system is vulnerable to buffer overflow? What can cause buffer overflow and why is buffer overflow a serious problem? What would be the main strategies that a designer could use to avoid the possibility of buffer overflow?

   [10 marks]
   In some programming languages (most notably, in C), there is no mechanism that prevents a pointer into an array buffer from being advanced beyond the end of the array. This erroneous situation is called **buffer overflow**. Buffer overflow can allow data to be written to memory locations outside of the expected buffer region, and this overwriting can destroy stored data and/or code; it can also cause malicious code and data to be written into an embedded system. Such malicious code can cause the embedded system to cause damage to itself and to any other systems that it might be controlling. Buffer overflow might also allow private code and/or data to be extracted from an embedded system.

The danger of buffer overflow can be avoided if external users are not allowed to use the original unprotected mechanism of C to access elements in an array. Instead, the array data can be encapsulated as private data inside a software object. All accesses to the array data could be required to use object memory functions, which have parameter value checks that prevent buffer overflow from every happening.

7. Briefly describe the various ways in which the MicroC/OS real-time operating system (RTOS) has been designed and implemented to maximize its portability across different microcomputer platforms.

   [10 marks]

   - MicroC/OS makes minimal assumptions about the hardware: The CPU must be supported by a C compiler, and there must be a software-configurable hardware timer. Virtually all modern microcomputers will satisfy these assumptions.
   - The software architecture of MicroC/OS partitions the software into the relatively small CPU-dependent port layer, the CPU-independent core, and the application-specific configuration. The software that must be modified when migration occurs to a new CPU is thus easily found in only the port layer.
   - Except for a small amount of CPU-specific assembly language in the port layer, MicroC/OS (and the application tasks) are implemented entirely in ANSI C. ANSI is widely supported by compilers for all of the major CPU architectures. Recompiling MicroC/OS will accomplish most of the work that is required to migrate the software system to a new CPU.

8. Sufficient idle time must be present in real-time systems. What would happen to the handling behaviour of externally and internaly initiated events by an embedded system if the amount of idle time were reduced below a safe percentage of the CPU's execution time? What sorts of problems could one expect to see when the idle time started to become insufficient?

   [10 marks]
   External events: An embedded system does not have control over the timing of the arrival of external events. In order to provide guarantees over the worst-case response time and the determinism of those response times, the embedded system must have sufficient idle time that is ready to be applied to handle sudden increases in CPU workload when multiple external events happen in a burst. If there were insufficient idle time, then the real-time performance of the embedded system would likely violate the maximum response times and/or the maximum variability of the response time. If this kind of performance failure were a rare occurrence, then the performance of the system might still be accurately called soft real-time, but it would not be hard real time.

   Internal events: The embedded system has much more control over the timing of internal events compared to the time of external events. But in a complex system with many activities being handled together (such as direct memory access transfers, data-dependent variablilty in calculation times, the effects of cache memory), it may be difficult to ensure that all of the internal real-time constraints are handled in the worst-case. Providing sufficient idle time makes is much easier to guarantee that the real-time response time constraints for internal event handling will be met.

   Perfomance problems: When the amount of idle time becomes insufficient, then the real-time response behaviour of the embedded system will start to violate the system specifications. The response times will start to get too slow. The response times for the same kinds of events will start to have much great varibility. If the response times get very slow, then the external system that the embedded system is supposed to control might have serious failures. For example, the movements of robot arms may overshoot because the embedded system was too slow to react.

9. The single-threaded, non-preemptive loop architecture is recommended when designing hard real time embedded systems that are implemented on "bare metal", without an operating system. In such an architecture the execution times of the single thread are controlled by a hardware timer. First, briefly describe the major elements of this architecture. Then briefly explain how adopting this architecture simplifies the problem of meeting each of the following kinds of real-time specifications: (a) the maximum response times, (b) the maximum variability in the response times, (c) the accuracy of repetition frequencies, (d) the maximum variability in the repetition frequencies, and (e) the control of degradation behaviour.

   [10 marks]
   Major elements of the architecture: In this architecture there is a single thread (an executing program) that begins to executing a fixed list of actions (e.g., polls) when triggered by an

interrupt from hardware timer. The timer has a fixed period, which is long enough in duration to allow the thread to execute all of the polling actions while still leaving some idle time before the next timer interrupt. There is no operating system, only the single thread and the timer interrupt service routing. It is possible to have other interrupts in the system, which would trigger the execution of additional interrupt service routines.

Ways in which this architecture simplifies the problem of meeting real-time specifications:

a. The maximum response time is determined by the fixed period of the timer. The period is chosen to be long enough that all of the polling actions can be completed within one period.

b. The maximium variablity in the response time is determined by the period of the timer. An external event could occur just before its polling operation, which would produce a fast response; alternatively, the external event could occur just after its polling operation, which would produce a slow response that is about the period of the timer.

c. The accuracy of the repetition frequencies should be high since the accuracy is determined by that of a crystal-stabilized hardware timer.

d. There should be very small variability in the repetition frequency since high accuracy and stability is ensured by the use of a hardware timer.

e. If the timer loop allows for idle time, then the degradation behaviour of the system can be controlled since bursts of workload will be accommodated by the idle time in each loop iteration. If the idle time approaches being entirely used up in each iteration, then some of the less critical polling actions could be omitted from some of the loop iterations to free up more time to perform the more critical polling actions. As a more drastic measure, the period of the timer could be increased to provide more idle time, but this measure could cause many of the polling actions to violate their maximum response times with potentially serious consequences.