18-540
Distributed Embedded Systems
Test #1
September 22, 1999

SOLUTIONS

Name (please print): _____________________________________________________

Instructions:  

DO NOT OPEN TEST UNTIL TOLD TO START
This test lasts from 8:45 AM to 10:15 AM (normal class hours)

The test is composed of four problems containing a total of 11 sub-problems, adding up to 100 points overall. The point value is indicated for each sub-problem. Attempt all problems and budget your time according to the problem’s difficulty. Show all work in the space provided. If you have to make an assumption then state what it was. Answers unaccompanied by supporting work will not receive full credit. The exam is closed book, closed notes, and “closed neighbors.” You are on your honor to have erased any course-relevant material from your calculator prior to the start of the test. Please print your initials at the top of each page in case the pages of your test get accidentally separated. You may separate the pages of the test if you like, and re-staple them when handing the test in.

Good luck!
Example System.

This example system is used for all of the questions in this test. A single distributed system is being used to help reduce the time you spend absorbing system information rather than answering questions. However, this is a two-edged sword. Make sure that you really understand this example so you don’t make a systematic error across the entire test. Not every piece of information below is required for every problem.

Consider a 100-spot parking garage implemented with distributed embedded systems for parking fee payment. The system is designed to work as you would expect with a parking meter system, but with a few enhancements as described. (This means, assume normal, everyday parking meter behavior unless specifically stated otherwise; but ignore the possibility of meter jams. If you are unsure of what this behavior might be, please ask the course staff specific questions about areas of doubt.) The system is distributed in that each of the sensors, actuators, and objects listed can only communicate via a real time embedded network. Depositing one quarter gives 30 minutes of time.

The system is composed of the following objects, where i is a parking spot number from 1 to 100.
- SPOT[i]: parking spots
- CAR[i]: car parked in SPOT[i], having values {True, False} to describe whether a car is actually parked in SPOT[i] at any given instant
- DRIVER[i]: vehicle operator for CAR[i]
- TIMER[i]: automatically keeps track of the minutes and seconds of paid meter time remaining for SPOT[i]

The following sensors are used:
- CoinSensor[i]: one per SPOT[i]; goes to True when a coin is deposited at SPOT[i]; values {True,False}
- CarPresent[i]: one per SPOT[i]; a magnetic induction sensor that can sense the presence of a car in the associated spot; values {True,False}

The following actuators are used:
- TimeDisplay[i]: one per SPOT[i]; displays an integer value of minutes in the range {0..60}. (We will ignore the conversion to hour:minute display format that happens in real systems.)
- RedFlag[i]: one per SPOT[i]; turns on when a car is deserving of a parking ticket (an electronic “red flag”); values {True,False}

The following performance constraints exist in this system:
- To save costs, the communication network used transmits messages over power wires at the rate of 1 (one) message per second.
- The nodes transmitting on the system network are: TIMER[i], CoinSensor[i], and CarPresent[i]. A very simple network protocol and a time-triggered system design is used that transmits messages from each possible transmitting node (100 copies each of the three transmitting node types) in turn to form a single transmission round, then repeats. The order of message transmission is not guaranteed to be any particular order, so assume worst case.
1) Requirements

1a) (20 points)
List the requirements using the same ground-rules as used in the project assignment Part #1 to implement a normal, every-day parking meter system. Note that such a system does not employ a CarPresent sensor, so don’t mention this in your requirements for this particular sub-question. Be sure to account for all sensors and actuators with respect to all relevant objects. Only the first 35 words of each requirement will be graded. (Hints: correct answers probably only have about a “handful” of requirements – not a huge, elaborate list. Remember NOT to mix implementation into this – it is purely a requirements question!)

- Each distinct time that CoinSensor[i] indicates True, TIMER[i] shall be adjusted to reflect an additional 30 minutes of parking time, upto a maximum of 60 minutes.

- TimeDisplay[i] shall display a reasonably current value from TIMER[i].

- RedFlag[i] shall be set to False whenever TIMER[i] has a non-zero value.

- RedFlag[i] shall be set to True as soon as reasonably possible after TimeDisplay[i] begins displaying zero.
1b) (8 points)
The CarPresent sensor was added at the request of students who were irate that staff members “hog” the spaces by feeding quarters to the meters all day long. A duplicate set of RedFlag[i] displays was also added at the security office to speed up ticketing operations with no effect on the distributed system behavior. **List additional requirements** that will suffice to ensure that any particular CAR[i] cannot remain in its SPOT[i] for more than (approximately) 60 minutes without triggering an associated RedFlag[i] and its duplicate in the security office. Only the first 35 words of each requirement will be graded. (Hints: no system is foolproof, but an appropriate solution would require physically starting and moving the CAR[i] approximately every hour rather than simply popping in more quarters to the meter. Don’t forget to deal with the case of someone leaving before the meter runs down to zero.)

- 15 minutes after any quarter has been accepted by CoinSensor[i], CoinSensor[i] shall ignore all future quarters until CarPresent[i] indicates False
2) Modeling & Global Time

2a) (6 points)
How long does it take for one round of messages to be transmitted on the bus for the complete system (i.e., for every possible transmitting node to send its message)? (Hint: the answer to this reveals a severe design defect in this system if you think about it, but we are going to ignore that problem for purposes of this test.)

\[
100 \text{ TIMER messages} + 100 \text{ CoinSensor messages} + 100 \text{ CarPresent[i] messages} = 300 \text{ messages}
\]

At one message/second it takes 300 seconds = 5 Minutes (wow, that’s slow! – but this mistake gets made in real life all the time, wasting $1B in a particular military system case...)

You’re going to have problems with drivers irritated that the time display doesn’t register their quarters for a long time, but we’re going to ignore that for this test... (that’s why we put in the hint)

2b) (8 points)
Given that a DRIVER deposits quarters for 60 minutes of parking at SPOT[i], what is the longest possible time (worst case assuming everything behaves in the worst possible, yet failure-free, way) between depositing the last quarter and the activation of RedFlag[i]?

For this sub-problem, and only this sub-problem, assume that: TIMERs keep perfect time, all computational processes have zero latency, and that depositing a quarter takes zero time.

Worst case is that the message for CoinSensor[i] message to be transmitting just as the quarter is inserted, meaning that 300 seconds go by before the message is sent that the coin was inserted.

An additional worst case is that the TIMER[i] message to TimeDisplay[i] is sent right before the CoinSensor[i] message, requiring a 299 second wait for that message to be sent.

Thus, worst case latency is 599 seconds + 60 minutes of meter time = 69 Minutes, 59 seconds.
2c) (6 points)
Assume that each COUNT_DOWN (the actual “clock” in each TIMER) uses an inexpensive oscillator circuit to generate its micro-ticks, and can drift in value by up to 0.01 seconds/second of elapsed time (otherwise called a $10^{-2}$ drift rate). What is the maximum possible error in time measurement for a 60-minute interval?

60 minutes = 3600 seconds * 0.01 seconds/second drift = 36 seconds error

2d) (8 points)
Suppose you were a detective and trying to use parking system logged data to determine which of two cars arrived in the lot first to settle a door-dinging dispute. If you have complete information about all CarPresent[i] messages sent on the network as logged in a single data logging device that passively collects all network messages sent. In the worst case, how far apart would two cars have to arrive in real-world time (called “reference time” in the course) for the logged data to unambiguously be able to show the sequence of arrival.

Each CarPresent[i] messages are sent every 5 minutes. Worst case is that CAR[a] arrives, just missing the CarPresent[a] message, 300 messages for it to register True the next CarPresent[a] message. You then need to see a CarPresent[b]=False message to disambiguate arrival ordering. In the worst case, this could require another 299 messages depending on message ordering. Thus, the cars must arrive at least 599 messages = 9 minutes 59 seconds apart to establish an unambiguous arrival ordering.
3) Control Loops & Worst Case Timing

The CoinSensor uses a small electric motor to remove each deposited quarter from an intake slot and put it in a coin “bucket” pending eventual collection by the parking lot owner. The amount of current used by the electric motor to move the coin is used to estimate its weight and (in at least most cases) detect incorrect or fraudulent coins, which are simply ignored. Assume that the entire movement and weighing operation takes exactly 1 second to perform and that the microcontroller in the CoinSensor both supplies current to the motor and measures motor speed, attempting to keep the speed constant at 600 RPM (revolutions per minute), and requiring at least 9 complete revolutions worth of current measurements at full operating speed. It is OK for the motor to be operating at full speed at the end of the operation – ignore the ramp-down time and reset time to be ready to accept the next quarter.

3a) (5 points)
What is the longest possible \( \text{rise} \) for the motor to reach full speed in moving the quarter? (Recall that \( \text{rise} \) is the step function response of the system; make an approximation of this as time from zero to essentially full speed rather than the strictest possible definition of \( \text{rise} \).)

600 RPM = 10 revolutions per second. Since 9 revolutions have to be at full speed, this gives only 1 revolution rise time (maximum possible), giving 100 msec maximum rise time.

3b) (6 points)
The CoinSensor uses a 100 KHz microcontroller to avoid problems with generating electromagnetic interference. Each instruction takes exactly two clock cycles. Ignoring time to take current measurements and based on your answer to (3a), what is the maximum number of instruction execution time periods available to run the motor control loop using good design practice as taught in class?

Sample time should be 10 times faster than rise time. This means 100 msec / 10 = 10 msec sample time.

10 msec * (100,000 clocks/sec / 2 clocks/instruction) = 500 instructions
3c) (7 points)
After the system is deployed, customers complain about the machines rejecting good quarters. During the ensuing design review it becomes apparent that the designers ignored some timing effects on the CoinSensor computer. In particular, the CoinSensor computer is subject to the following effects. Accounting for these effects and your answer to (3b), how many instructions execution periods are available to run the motor control loop?

- Every 1 msec, the CPU may be stalled for up to 10 clock cycles to permit the network interface to access shared memory space.
- Multiplies use an “early-out” algorithm based on data value, and can therefore range in execution time from 2 to 16 clock cycles. There are 8 multiplies in the motor control loop. The original algorithm was designed assuming 2 clock cycles for a multiply. In your answer, each multiply counts as 1 instruction execution period regardless of execution time.

Start with 500 instructions over 10 msec.

You lose 10 clocks every millisecond * 10 msec = 100 clocks to the network interface

You lose up to 14 clocks * 8 multiplies = 112 clocks to long multiplies

100 + 112 = 212 clocks lost = 106 instruction times, leaving 394 instructions for the control loop
4) State Diagram

Consider the TIMER object in the full example system design (including the CarPresent[i] sensor). Assume that:

- Each TIMER object has an integral COUNT_DOWN register that can store any integer number of seconds, and that COUNT_DOWN counts down to zero automatically, then stops at value zero.
- Each COUNT_DOWN register accepts the following commands and provides the following outputs from/to the TIMER (these are hard-wired control pins, not messages):
  - **ResetToZero.** Command that sets the COUNT_DOWN register to value zero.
  - **Add30.** Command that adds 30 minutes to the meter time (don’t worry about time delay between inserting quarters, although that has to be addressed in a real-world parking system). Due to a defective chip design, Add30 applied to a COUNT_DOWN timer where the result would exceed 60 minutes crashes the system.
  - **Decrement.** Output from the COUNT_DOWN register that provides a signal to the TIMER every time it counts down to a multiple of 30 minutes.
- It is NOT POSSIBLE to read, write, or otherwise access the register value directly (except, of course, using ResetToZero, Add30, and Decrement as just described). The register value is of course automatically sent out as a message, but assume you do not have access to that value for this problem.

4a) (6 points)
Concisely define all required incoming messages to TIMER from the system network (maximum 35 words per message description + a distinct message name) for a reasonable system design; but don’t bother to detail where the messages come from or go to. The definition must include a name and a list of data fields (but not number of bits or other such details) for each incoming message.

Incoming messages:

- **Coins(spot#, number_coins);** number of coins arriving since last such message arrived
- **CarPresent(spot#, True/False)**

(solution note: this system is so slow, you can’t possibly wait for a separate message for each coin; it will be bad enough that it takes many minutes for the coins to show up on the displays!)
4b) (20 points)
**Draw a finite state machine diagram** for the operation of TIMER[i], showing states, transitions, and conditions used for transitions. Correct answers shall have the following characteristics:

- Labeled states, with no more than 5 states (in fact, you may or may not need even this many depending on your approach).
- Labeled transitions, with labels consisting only of one of:
  - messages from (4a) having data fields specified as appropriate
  - interface signals: ResetToZero, Add30, Decrement
- No “variables” or other hacks around using explicit states