Internal Assessment Test 1 – September 2016

10CS72 - Embedded Computing Systems

Answers

1.

a. **What is an embedded computer system? Explain the characteristics and constraints of embedded computing applications. (7)**

An Embedded computer system is any device that includes a programmable computer but is not itself intended to be a general-purpose computer. An embedded system is typically an electronic/electro-mechanical system designed to perform a specific function and is a combination of both hardware and software.
Fax machine, clock built from a microprocessor, microwave oven, washing machine, elevator controller are examples of embedded computing system.

Characteristics of Embedded computing applications:

1) Embedded computing systems have to provide sophisticated functionality:
#Complex algorithms: The operations performed by the microprocessor may be very sophisticated. For example, a microprocessor to control an automobile engine must perform complex functions to optimize the performance of the car while minimizing pollution and fuel utilization.
#User interface: Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. For example, moving maps in Global Positioning System (GPS) navigation have sophisticated user interfaces.

2) Embedded computing operations must often be performed to meet deadlines:
#Real time: Many embedded computing systems have to perform in real time— if the data is not ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, it may not create safety problems but does create unhappy customers—missed deadlines in printers, for example, can result in scrambled pages.
#Multirate: Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of multirate behavior. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a

deadline on either the audio or video portions spoils the perception of the entire presentation.

3) Costs of various sorts are also very important:
#Manufacturing cost: The total cost of building the system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.
#Power and energy: Power consumption directly affects the cost of the hardware, since a larger power supply may be necessary. Energy consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

4) Finally, most embedded computing systems are designed by small teams on tight deadlines.

Constraints of an embedded system:

How much hardware do we need?
We need to choose the amount of computing power we apply to our problem. We can not only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more. Since we often must meet performance deadlines and manufacturing cost constraints, the choice of hardware is important—*too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.*

How do we meet deadlines?
The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster. But, this makes the system more expensive. It may seem that increasing the CPU clock rate increases program speed. But, it may not be true, because the program's speed may be limited by the memory system.

How do we minimize power consumption?
In battery-powered applications, power consumption is extremely important. Even in non-battery applications, excessive power consumption can increase heat dissipation. One way to make a digital system consume less power is to make it run more slowly. But too much slowing down of the system leads to missing deadlines. Careful design is required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals.

How do we design for upgradability?
The hardware platform may be used over several product generations or for several different versions of a product in the same generation, with few or no changes. However, we want to be able to add features by

changing software. We need to think ahead when we design a machine that will provide the required performance for software in the future.

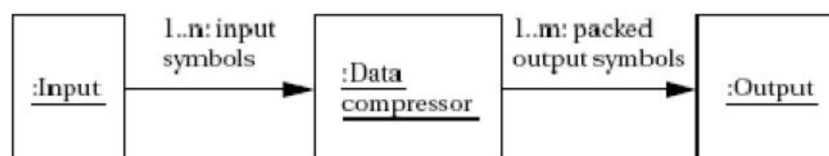
Does it really work?
Reliability is always important when selling products—customers rightly expect that products they buy will work. Reliability is especially important in some applications, such as safety-critical systems. If we wait until we have a running system and try to eliminate the bugs, we will be too late—we won't find enough bugs, it will be too expensive to fix them, and it will take too long as well. Following set of challenges comes from the characteristics of the components and systems themselves:

#Complex testing: Exercising an embedded system is generally more difficult than typing in some data. We may have to run a real machine in order to generate the proper data. The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

#Limited observability and controllability: Embedded computing systems usually do not come with keyboards and screens. This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applications we may not be able to easily stop the system to see what is going on inside.

#Restricted development environments: The development environments for embedded systems (the tools used to develop software and hardware) are often much more limited than those available for PCs and workstations. We generally compile code on one type of machine, such as a PC, and download it onto the embedded system. To debug the code, we must usually rely on programs that run on the PC or workstation and then look inside the embedded system.

b. Draw UML collaboration diagram for the data compressor. (3)



2.

a. Explain with a neat diagram the embedded system design process. (7)

Embedded system design process:

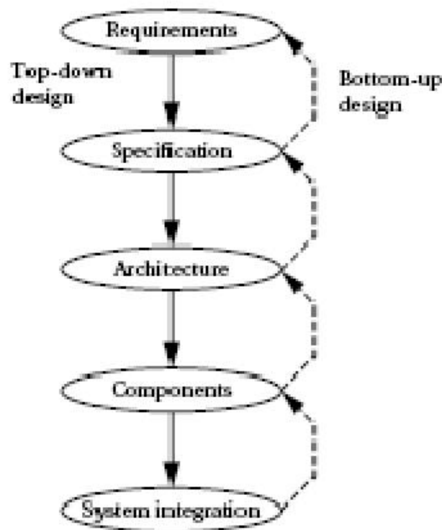


Figure above summarizes the major steps in the embedded system design process. The design process starts with the system ***requirements***, followed by ***specification***, where we create a more detailed description of what we want. But the specification states only how the system behaves, not how it is built. The details of the system's internals begin to take shape when we develop the ***architecture***, which gives the system structure in terms of large components. Once we know the components we need, we can ***design components***, including both software modules and any specialized hardware we need. Based on those components, we can finally ***integrate*** it into a complete system.

There are two ways of considering the design:

- ***Top–down***—Design begins with the most abstract description of the system and conclude with concrete details.
- **Bottom–up**— Design starts with the components to build a system. Bottom–up design steps are shown in the figure as dashed-line arrows. We need bottom–up design because:
  - We do not have perfect insight into how later stages of the design process will turn out.
  - Decisions at one stage of design are based upon estimates of what will happen later. *In general, the less experience we have with the design of similar systems, the more we will have to rely on bottom-up design information to help us refine the system.*

We need to consider the major goals of the design:
manufacturing cost;
performance (both overall speed and deadlines); and
power consumption.
We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:
We must *analyze* the design at each step to determine how we can meet the specifications.
We must then *refine* the design to add detail.

And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

1 Requirements

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components. There are two phases:

- First, we gather an informal description from the customers known as requirements.
- Second, we refine the requirements into a specification that contains enough information to begin designing the system architecture.

Separating out requirements analysis and specification is necessary because of the large gap between what the customers can describe about the system they want and what the architects need to design the system. Hence we need to keep in mind the following:

- Consumers of embedded systems are usually not embedded system designers. They can only envision users' interactions with the system.
- Consumers may have unrealistic expectations as to what can be done within their budgets; and they may also express their desires in a language very different from system architects' jargon.
- A Structured approach is to capture consistent set of requirements from the customer and then massaging those requirements into a more formal specification. This helps us manage the process of translating from the consumer's language to the designer's.

Requirements may be ***functional*** or ***nonfunctional***.
Functional Requirements:
We need to capture the basic functions of the embedded system. This described what the embedded system is intended to do.

Non-functional requirements include:
Performance: The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost. Performance is a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed.
Cost: The target cost or purchase price for the system is a key factor. Cost typically has two major components: ***manufacturing cost*** includes the cost of components and assembly; ***nonrecurring engineering*** **(NRE)** costs include the personnel and other costs of designing the system.
Physical size and weight: The physical aspects of the final system can vary greatly depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict

limitations on weight. A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.
Power consumption: Power is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life—the customer is unlikely to be able to describe the allowable wattage.

Validating a set of requirements is ultimately a psychological task since it requires understanding both what people want and how they communicate those needs.

One good way to refine at least the user interface portion of a system's requirements is to build a ***mock-up***.

- The mock-ups use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation.
- Mock-ups give the customer a good idea of how the system will be used and how the user can react to it.

Physical, nonfunctional ***models*** of devices can also give customers a better idea of characteristics such as size and weight.

Name

Purpose

Inputs

Outputs

Functions

Performance

Manufacturing cost

Power

Physical size and weight

FIGURE 1.2
Sample requirements form.

Figure 1.2 above shows a sample ***requirements form*** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of the system. The entries in the form are:
Name: This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people but can also crystallize the purpose of the machine.
Purpose: This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.
Inputs and outputs: These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:
— *Types of data:* Analog electronic signals? Digital data? Mechanical inputs?
— *Data characteristics:* Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?

— *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?
Functions: This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?
Performance: Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. The computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.
Manufacturing cost: This includes primarily the cost of the hardware components. A rough estimate on the cost should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at $10 most likely has a very different internal structure than a $100 system.
Power: A rough idea of how much power the system can consume is very important. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.
Physical size and weight: Some indication of the physical size of the system will guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel mounted voice recorder.

After writing the requirements, we should check for internal consistency.

2 Specification

The specification is more precise—*it serves as the contract between the customer and the architects*. As such, the specification must be carefully written so that it accurately reflects the customer's requirements and does so in a way that can be clearly followed during design.

Characteristics of a good specification:

1) Meet system and customer requirements
2) Should be unambiguous
3) Should be clear and understandable
4) Should be complete

UML is the language that is widely used for describing specifications.

3 Architecture Design

- *The specification only says what the system does, but does not say how the system does things*.

- The purpose of Architecture is to describe how the system implements those functions.

- The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture.
- The creation of the architecture is the first phase of design.

There are 2 major levels of architectural description.

First, high level block diagram as shown in Figure (a) below:

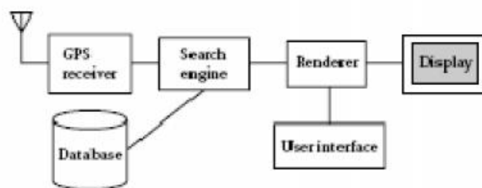


FIGURE 1.3
Block diagram for the moving map.

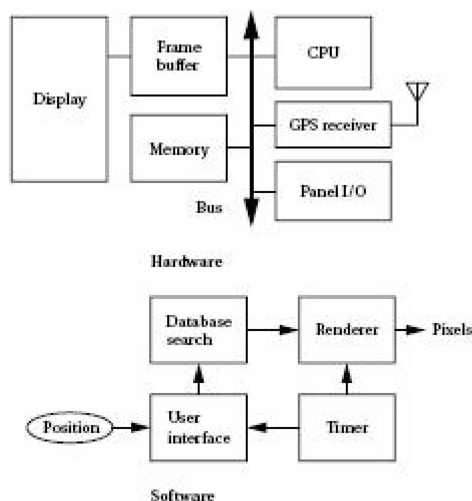


FIGURE 1.4
Hardware and software architectures for the moving map.

Figure 1.3 shows sample system architecture in the form of a block diagram that shows major operations and data flows among them. This block diagram is still quite abstract, and does not specify which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. However, it describes how to implement the functions described in the specification.

After we have designed an initial architecture that is not biased toward too many implementation details should we refine that system block diagram into 2 block diagrams: one for **hardware** and another for **software**. These two more refined block diagrams are shown in Figure 1.4. These include more details such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed in time.

Architectural descriptions must be designed to satisfy both *functional and nonfunctional requirements*. Not only must all the required functions be present, but we must meet cost, speed, power, and other nonfunctional constraints.

Starting out with system architecture and refining that to hardware and software architectures is one good way to ensure that we meet all specifications: We can concentrate on the functional elements in the system block diagram, and then consider the nonfunctional constraints when creating the hardware and software architectures.

We must somehow be able to estimate the properties of the components of the block diagrams, such as the search and rendering functions in the moving map system. Accurate estimation derives in part from experience, both general design experience and particular experience with similar systems. However, we can sometimes create simplified models to help us make more accurate estimates. Sound estimates of all nonfunctional constraints during the architecture phase are crucial, since decisions based on bad data will show up during the final phases of design, indicating that we did not meet the specification.

4 Designing Hardware and Software Components

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification.

The components will in general include both hardware—FPGAs, boards, and so on—and software modules. Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components.

In GPS moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component.

We can also make use of standard software modules. One good example is the topographic database. Standard topographic databases exist, and you probably want to use standard routines to access the database—not only is the data in a predefined format, but it is highly compressed to save storage. Using standard software for these access functions not only saves us design time, but it may give us a faster implementation for specialized functions such as the data decompression phase.

We have to design some components ourselves. Even if we are using only standard integrated circuits, we may have to design the printed circuit board that connects them. We have to do a lot of custom programming as well.

When creating these embedded software modules, we must make use of our expertise to ensure that the system runs properly in real time and that it does not

take up more memory space than is allowed. The power consumption of the moving map software example is particularly important.
We need to be very careful about how you read and write memory to minimize power—for example, since memory accesses are a major source of power consumption, memory transactions must be carefully planned to avoid reading the same data several times.

5 System Integration

After the components are built, we need put them together and see the working system. This phase is very critical, and usually consists of lot of bugs. Good planning helps us find the bugs quickly. Building the system in phases and running properly chosen tests can also find bugs more easily.

Only by fixing the simple bugs early will we be able to uncover the more complex or obscure bugs that can be identified only by giving the system a hard workout. We need to ensure during the architectural and component design phases that we make it as easy as possible to assemble the system in phases and test functions relatively independently.

System integration is difficult because it usually uncovers problems. It is often hard to observe the system in sufficient detail to determine exactly what is wrong— the debugging facilities for embedded systems are usually much more limited than what you would find on desktop systems. As a result, determining why things do not stet work correctly and how they can be fixed is a challenge in itself. Careful attention to inserting appropriate debugging facilities during design can help ease system integration problems, but the nature of embedded computing means that this phase will always be a challenge.

b. How would the ARM status word be set after the operation: -4+5 ? (3)

-4 in hex = 0xfffffffc
5 in hex = 0x5
-4+5 = 0xfffffffc+0x5=0x1 with a carry out = 1
Since we are adding a +ve and a –ve number, and getting a carry out, Carry flag will be set.
N: 0
Z: 0
C: 1
V: 0

3.

a. Discuss the requirements chart, with an example. (7)

Figure below shows a sample requirements form that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of the system.

Name

Purpose

Inputs

Outputs

Functions

Performance

Manufacturing cost

Power

Physical size and weight

The entries in the form consist of the following:

Name: This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people but can also crystallize the purpose of the machine.

Purpose: This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.

Inputs and outputs: These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:

— *Types of data:* Analog electronic signals? Digital data? Mechanical inputs?

— *Data characteristics:* Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?

— *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?

Functions: This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?

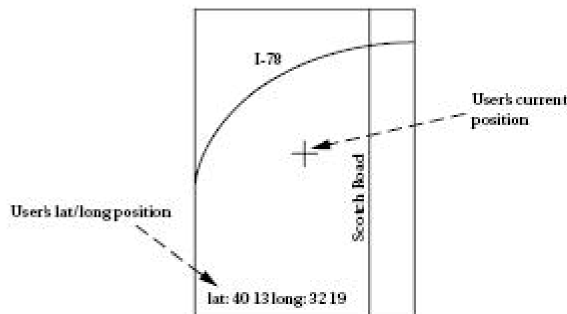
Performance: Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. The computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.

Manufacturing cost: This includes primarily the cost of the hardware components. A rough estimate on the cost should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at $10 most likely has a very different internal structure than a $100 system.

Power: A rough idea of how much power the system can consume is very important. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.

Physical size and weight: Some indication of the physical size of the system will guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel mounted voice recorder.

After writing the requirements internal consistency must be checked.
Example: Let us consider an example of GPS moving map system.



Initial List:
Functionality: This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.
User interface: The screen should have at least 400_600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed to allow the user to make selections to control the system.
#Performance: The map should scroll smoothly. Upon power-up, a display should take no more than one second to appear, and the system should be able to verify its position and display the current map within 15 s.
Cost: The selling cost (street price) of the unit should be no more than $100.
Physical size and weight: The device should fit comfortably in the palm of the hand.
Power consumption: The device should run for at least eight hours on four AA batteries.

Requirements Chart:

Name	GPS moving map
Purpose	Consumer-grade moving map for driving use
Inputs	Power button, two control buttons
Outputs	Back-lit LCD display 400 _ 600
Functions	Uses 5-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude
Performance	Updates screen within 0.25 seconds upon movement
Manufacturing cost	$30

Power	100mW
Physical size and weight	No more than 2” _ 6, ” 12 ounces

b. **What is the meaning of these ARM condition codes: EQ, NE, MI, VS, GE, and LT? (3)**

Code	Meaning	Flags Tested
EQ	Equal.	Z==1
NE	Not equal.	Z==0
CS or HS	Unsigned higher or same (or carry set).	C==1
CC or LO	Unsigned lower (or carry clear).	C==0
MI	Negative. The mnemonic stands for "minus".	N==1
PL	Positive or zero. The mnemonic stands for "plus".	N==0
VS	Signed overflow. The mnemonic stands for "V set".	V==1
VC	No signed overflow. The mnemonic stands for "V clear".	V==0
HI	Unsigned higher.	(C==1) && (Z==0)
LS	Unsigned lower or same.	(C==0) || (Z==1)
GE	Signed greater than or equal.	N==V
LT	Signed less than.	N!=V
GT	Signed greater than.	(Z==0) && (N==V)
LE	Signed less than or equal.	(Z==1) || (N!=V)
AL (or omitted)	Always executed.	None tested.

4.

a. **Draw and explain the sequence diagram for transmitting control input in a model train controller. (7)**

The role of the formatter during the panel’s operation is illustrated by the sequence diagram below. The figure shows two changes to the knob settings: first to the throttle, inertia, or emergency stop; then to the train number. The panel is called periodically by the formatter to determine if any control settings have changed. If a setting has changed for the current train, the formatter decides to send a command, issuing a *send-command* behavior to cause the transmitter to send the bits. Because transmission is serial, it takes a noticeable amount of time for the transmitter to finish a command; in the meantime, the formatter continues to check the panel’s control settings. If the train number has changed, the formatter must cause the knob settings to be reset to the proper values for the new train.

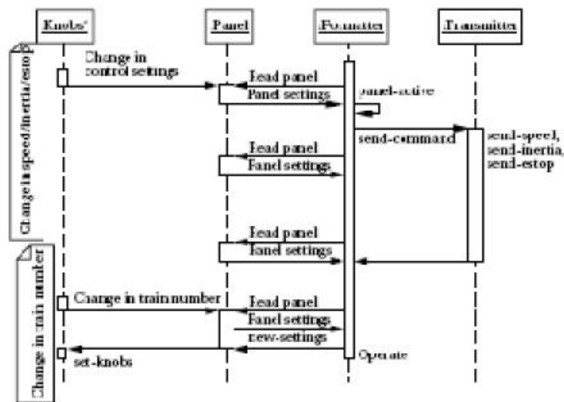


FIGURE 1 24
Sequences diagram for transmitting a control input.

b. Write ARM assembly code to implement the following C assignment. (3)

- **y=(a<<3) | (b&15);**

```
ADR r4, a ; get address for a
LDR r0, [r4] ; get value of a
MOV r0, r0, LSL 3 ; perform shift
ADR r4, b ; get address for b
LDR r1, [r4] ; get value of b
AND r1, r1, #15 ; perform logical AND
ORR r1, r0, r1 ; compute final value of z
ADR r4, z ; get address for z
STR r1, [r4] ; store value of z
```

5.

a. Define Digital Command Control (DCC). Explain the conceptual specification of a model train controller system. (7)

Digital Command Control (DCC) is a standard for a system to operate model railways digitally. DCC specifies some important aspects of the system, particularly those that allow equipment to interoperate. But DCC deliberately does not specify everything about a model train control system.

The DCC standard is given in two documents:
Standard S-9.1, the DCC Electrical Standard, defines how bits are encoded on the rails for transmission.
Standard S-9.2, the DCC Communication Standard, defines the packets that carry information.

Conceptual Specification of model train controller:
A ***conceptual specification*** allows us to understand the system a little better. We will use the experience gained by writing the conceptual specification to help us write a detailed specification to be given to a system architect.

A train control system turns ***commands*** into ***packets***. A command comes from the command unit while a packet is transmitted over the rails. Commands and packets may not be generated in a 1-to-1 ratio. Sometimes command units should resend packets in case a packet is dropped during transmission.

There are two major subsystems in model train controller: the command unit and the train-board component as shown in Figure 1.16. Each of these subsystems has its own internal structure. The basic relationship between them is illustrated in Figure 1.17, shown as a UML ***collaboration diagram***. The command unit and receiver are each represented by objects; the command unit sends a sequence of packets to the train's receiver, as illustrated by the arrow. The notation on the arrow provides both the type of message sent and its sequence in a flow of messages; since the console sends all the messages, the arrow's messages are numbered as 1…*n*. The messages are carried over the track.

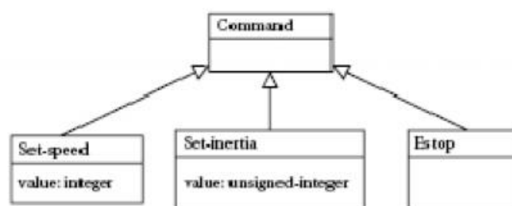


FIGURE 1.16
Class diagram for the train controller messages.

FIGURE 1.17
UML collaboration diagram for major subsystems of the train controller system.

The command unit and receiver are further broken down into their major components.
The console needs to perform three functions: read the state of the front panel on the command unit, format messages, and transmit messages.
The train receiver must also perform three major functions: receive the message, interpret the message (taking into account the current speed, inertia setting, etc.) and actually control the motor.
The UML class diagram for the above is shown in Figure 1.18. The basic characteristics of these classes are:
The *Console* class describes the command unit's front panel, which contains the analog knobs and hardware to interface to the digital parts of the system.
The *Formatter* class includes behaviors that know how to read the panel knobs and creates a bit stream for the required message.
The *Transmitter* class interfaces to analog electronics to send the message along the track.

There will be one instance of the *Console* class and one instance of each of the component classes, as shown by the numeric values at each end of the relationship links. We have also shown some special classes that represent analog components, ending the name of each with an asterisk:
*Knobs** describes the actual analog knobs, buttons, and levers on the control panel.
*Sender** describes the analog electronics that send bits along the track.
Likewise, the Train makes use of three other classes that define its components:
The *Receiver* class knows how to turn the analog signals on the track into digital form.
The *Controller* class includes behaviors that interpret the commands and figures out how to control the motor.
The *Motor interface* class defines how to generate the analog signals required control the motor.
We define two classes to represent analog components:
*Detector** detects analog signals on the track and converts them into digital form.
*Pulser** turns digital commands into the analog signals required to control the motor speed.
We have also defined a special class, *Train set*, to help us remember that the system can handle multiple trains (up to t trains).

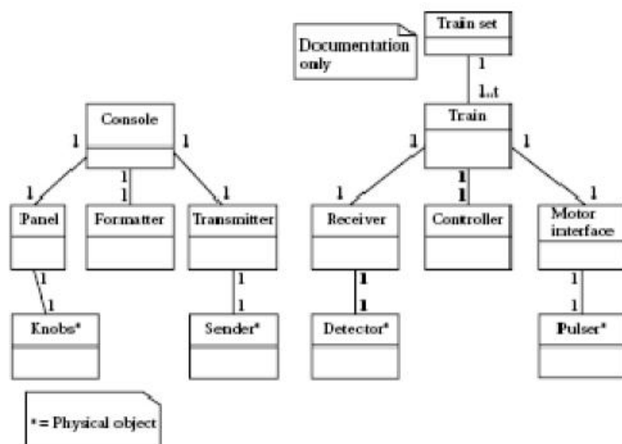


FIGURE 1.18
A UML class diagram for the train controller showing the composition of the subsystems.

b. How would the ARM status word be set after the operation: 2-3 ? (3)

2 in 32-bit hex = 0x2
-3 in 32-bit hex (2's complement form) = 0xFFFFFFFD
So, 2-3 = 0x2 + 0xFFFFFFFD = 0xFFFFFFFF = -1
Only Negative flag will be set.

N: 1

Z: 0
C: 0
V: 0

6.

a. Explain with a neat diagram the bus with a DMA controller. (7)

Standard bus transactions require the CPU to be in the middle of every read and write transaction. However, there are certain types of data transfers in which the CPU does not need to be involved. For example, a high-speed I/O device may want to transfer a block of data into memory. While it is possible to write a program that alternately reads the device and writes to memory, it would be faster to eliminate the CPU's involvement and let the device and memory communicate directly. This capability requires that some unit other than the CPU be able to control operations on the bus.

Direct memory access (DMA) is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a ***DMA controller***, which requests control of the bus from the CPU. After gaining control, the DMA controller performs read and write operations directly between devices and memory.

Figure below shows the configuration of a bus with a DMA controller. The DMA requires the CPU to provide two additional bus signals:
The ***bus request*** is an input to the CPU through which DMA controllers ask for ownership of the bus.
The ***bus grant*** signals that the bus has been granted to the DMA controller.

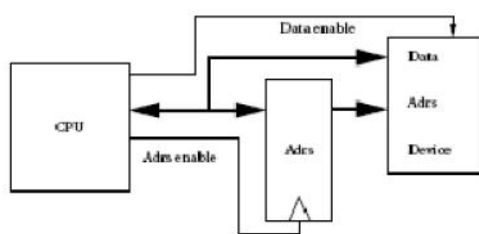


FIGURE 4.8
Bus signals for multiplexing address and data.

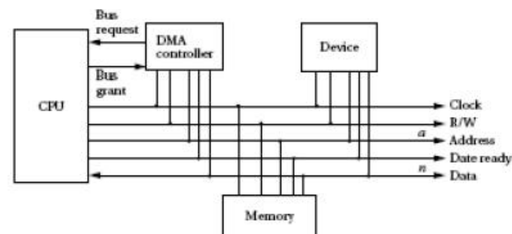


FIGURE 4.9
A bus with a DMA controller.

A device that can initiate its own bus transfer is known as a ***bus master***. Devices that do not have the capability to be ***bus masters*** do not need to connect to a bus request and bus grant. The DMA controller uses these two signals to gain control of the bus using a classic four-cycle handshake. The bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready.

The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the

other bus signals: R/W, address, and so on. Upon becoming bus master, the DMA controller has control of all bus signals (except, of course, for bus request and bus grant).
Once the DMA controller is bus master, it can perform reads and writes using the same bus protocol as with any CPU-driven bus transaction. Memory and devices do not know whether a read or write is performed by the CPU or by a DMA controller.

After the transaction is finished, the DMA controller returns the bus to the CPU by deasserting the bus request, causing the CPU to deassert the bus grant.

The CPU controls the DMA operation through registers in the DMA controller.

A typical DMA controller includes the following three registers:
 A starting address register specifies where the transfer is to begin.
 A length register specifies the number of words to be transferred.
 A status register allows the DMA controller to be operated by the CPU.

The CPU initiates a DMA transfer by setting the starting address and length registers appropriately and then writing the status register to set its start transfer bit. After the DMA operation is complete, the DMA controller interrupts the CPU to tell it that the transfer is done.

What is the CPU doing during a DMA transfer? It cannot use the bus. As illustrated in Figure 4.10, if the CPU has enough instructions and data in the cache and registers, it may be able to continue doing useful work for quite some time and may not notice the DMA transfer. But once the CPU needs the bus, it stalls until the DMA controller returns bus mastership to the CPU.

To prevent the CPU from idling for too long, most DMA controllers implement modes that occupy the bus for only a few cycles at a time. For example, the transfer may be made 4, 8, or 16 words at a time. As illustrated in Figure 4.11, after each block, the DMA controller returns control of the bus to the CPU and goes to sleep for a preset period, after which it requests the bus again for the next block transfer.

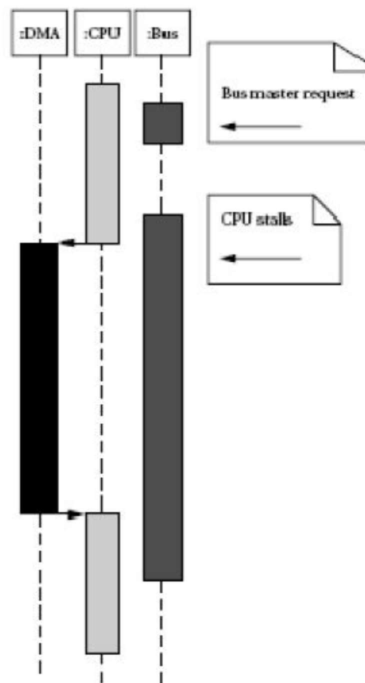


FIGURE 4.10
UML sequence diagram of system activity around a DMA transfer.

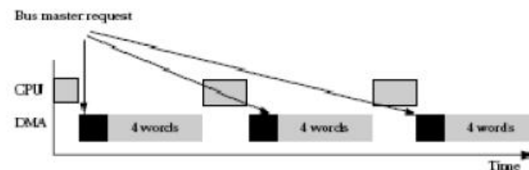


FIGURE 4.11
Cyclic scheduling of a DMA request.

b. **Write ARM assembly code to implement the following C assignment. (3)**

- **z = a*(b+c)-d*e;**

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result: (b+c)
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MUL r2,r2,r0 ; compute partial result: a*(b+c)
ADR r4,d ; get address for d
LDR r0,[r4] ; get value of d
ADR r4,e ; get address for e
LDR r1,[r4] ; get value of e
MUL r3,r0,r1 ; compute partial result: d*e
SUB r2, r2, r3 ; compute the final result
ADR r4,y ; get address for z
STR r2,[r4] ; store value of z at proper location
```

7.

a. **What is interrupt? Discuss its mechanism, with a neat diagram. (7)**

Interrupt is a mechanism that allows a device to request service from the CPU. The interrupt mechanism allows devices to signal the CPU and to force execution of a particular piece of code. When an interrupt occurs, the program counter's value is changed to point to an interrupt handler routine (also commonly known as a device driver) that takes care of the device: writing the next data, reading data that have just become ready, and so on. The interrupt mechanism of course saves the value of the PC at the interruption so that the CPU can return to the program that was interrupted. Interrupts therefore allow the flow of control in the CPU to change easily between different

contexts, such as a foreground computation and multiple I/O devices. To allow parallelism, we need to introduce interrupt mechanism into the CPU.

In the following example, we repeatedly read a character from an input device and write it to an output device. We assume that we can write C functions that act as interrupt handlers. Those handlers will work with the devices in much the same way as in busy-wait I/O by reading and writing status and data registers. The main difference is in handling the output—the interrupt signals that the character is done, so the handler does not have to do anything.
We will use a global variable achar for the input handler to pass the character to the foreground program. Because the foreground program doesn't know when an interrupt occurs, we also use a global Boolean variable, gotchar, to signal when a new character has been received. The code for the input and output handlers follows:

```
void input_handler() { /* get a character and put in global */
   achar = peek(IN_DATA); /* get character */
   gotchar = TRUE; /* signal to main program */
   poke(IN_STATUS,0); /* reset status to initiate next transfer */
}
void output_handler() { /* react to character being sent */
/* don't have to do anything */
}
```

The main program is reminiscent of the busy-wait program. It looks at gotchar to check when a new character has been read and then immediately sends it out to the output device.

```
main() {
   while (TRUE) { /* read then write forever */
         if (gotchar) { /* write a character */
               poke(OUT_DATA,achar); /* put character in device */
               poke(OUT_STATUS,1); /* set status to
initiate write */
         gotchar = FALSE; /* reset flag */
         }
   }
}
```

It should be noted that the use of interrupts has made the main program somewhat simpler. But this program design still does not let the foreground program do useful work.

Interrupt based I/O programming with buffers:

This method is an extension of the interrupt based I/O programming. In this, we use a more sophisticated program design to let the foreground program work completely independently of input and output. This is achieved by using buffers.
The following example shows copying characters from input to output with interrupts and buffers. Usage of buffers eliminated the need to wait for each

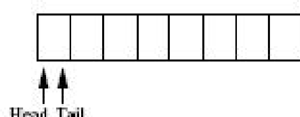
character. Rather than reading a single character and then writing it, the program performs reads and writes independently. The read and write routines communicate through the following global variables:

A character string `io_buf` will hold a queue of characters that have been read but not yet written.

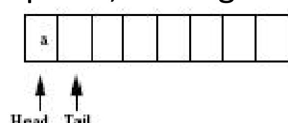
A pair of integers `buf_start` and `buf_end` will point to the first and last characters read.

An integer `error` will be set to 0 whenever `io_buf` overflows.

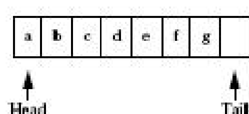
The global variables allow the input and output devices to run at different rates. The queue `io_buf` acts as a wraparound buffer—we add characters to the tail when an input is received and take characters from the tail when we are ready for output. The head and tail wrap around the end of the buffer array to make most efficient use of the array. Here is the situation at the start of the program's execution, where the tail points to the first available character and the head points to the ready character. As seen below, because the head and tail are equal, we know that the queue is empty.



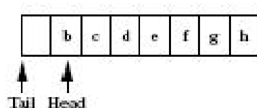
When the first character is read, the tail is incremented after the character is added to the queue, leaving the buffer and pointers looking like the following:



When the buffer is full, we leave one character in the buffer unused. As the next figure shows, if we added another character and updated the tail buffer (wrapping it around to the head of the buffer) we would be unable to distinguish a full buffer from an empty one.



Here is what happens when the output goes past the end of `io_buf`:



The following code provides the declarations for the above global variables and some service routines for adding and removing characters from the queue. Because interrupt handlers are regular code, we can use subroutines to structure code just as with any program.

```
#define BUF_SIZE 8
char io_buf[BUF_SIZE]; /* character buffer */
int buf_head = 0, buf_tail = 0; /* current position in
buffer */
int error = 0; /* set to 1 if buffer ever overflows */
void empty_buffer() { /* returns TRUE if buffer is empty */
      buf_head == buf_tail;
}

void full_buffer() { /* returns TRUE if buffer is full */
```

```
        (buf_tail+1) % BUF_SIZE == buf_head ;
  }

  int nchars() { /* returns the number of characters in the
  buffer */
        if (buf_head >= buf_tail) return buf_tail - buf_head;
        else return BUF_SIZE + buf_tail - buf_head;
  }

  void add_char(char achar) { /* add a character to the buffer
  head */
  io_buf[buf_tail++] = achar;
  /* check pointer */
  if (buf_tail == BUF_SIZE)
        buf_tail = 0;
  }

  char remove_char() { /* take a character from the buffer head
  */
  char achar;
  achar = io_buf[buf_head++];
  /* check pointer */
  if (buf_head == BUF_SIZE)
        buf_head = 0;
  }
```

Assume that we have two interrupt handling routines defined in C, input_handler for the input device and output_handler for the output device. These routines work with the device in much the same way as did the busy-wait routines. The only complication is in starting the output device: If io_buf has characters waiting, the output driver can start a new output transaction by itself. But if there are no characters waiting, an outside agent must start a new output action whenever the new character arrives. Rather than force the foreground program to look at the character buffer, we will have the input handler check to see whether there is only one character in the buffer and start a new transaction.

Here is the code for the input handler:

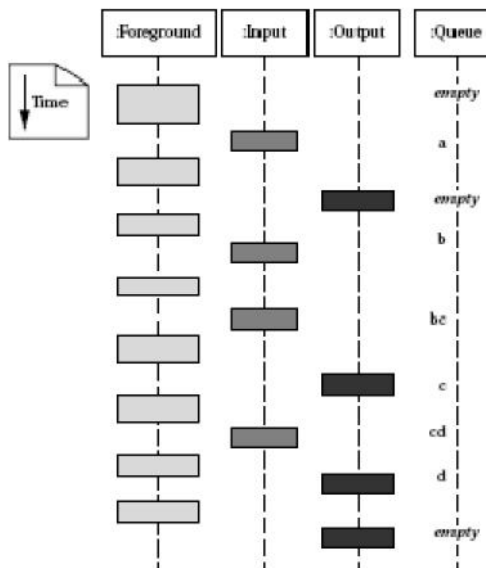
```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
      char achar;
      if (full_buffer()) /* error */
            error = 1;
      else { /* read the character and update pointer */
            achar = peek(IN_DATA); /* read character */
            add_char(achar); /* add to queue */
      }

poke(IN_STATUS,0); /* set status register back to 0 */
/* if buffer was empty, start a new output
transaction */
      if (nchars() == 1) { /* buffer had been empty until this
interrupt */
            poke(OUT_DATA,remove_char()); /* send character */
            poke(OUT_STATUS,1); /* turn device on */
      }
}

#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
void output_handler() {
```

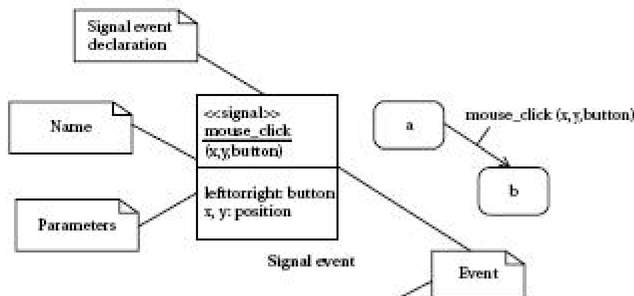
```
        if (!empty_buffer()) { /* start a new character */
              poke(OUT_DATA,remove_char()); /* send character */
              poke(OUT_STATUS,1); /* turn device on */
        }
}
```

The foreground program does not need to do anything—everything is taken care of by the interrupt handlers. The foreground program is free to do useful work as it is occasionally interrupted by input and output operations. The following sample execution of the program in the form of a UML sequence diagram shows how input and output are interleaved with the foreground program. (We have kept the last input character in the queue until output is complete to make it clearer when input occurs.) The simulation shows that the foreground program is not executing continuously, but it continues to run in its regular state independent of the number of characters waiting in the queue.

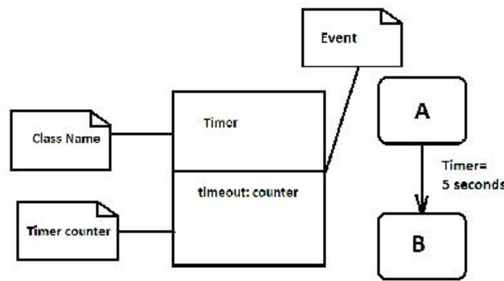


b. Explain the UML class diagram for signal and time out events. (3)

The class diagram below shows the stereotype <<signal>> for a mouse click event. The attributes for the class are leftorright button that specifies which button is clicked, and the (x,y) coordinates of the mouse pointer.



The timer class is shown below. The attribute for this is the counter value.



8.

a) Explain the following with diagram. (7) [*]

i. Two-level cache system

Modern CPUs may use multiple levels of cache, typically two, as shown in figure below. The ***first-level cache*** (commonly known as ***L1 cache***) is closest to the CPU.

The ***second-level cache (L2 cache)*** feeds the first-level cache, and so on. The second-level cache is much larger but is also slower. If $h1$ is the first-level hit rate and $h2$ is the rate at which access hit the second-level cache but not the first-level cache, then the average access time for a two-level cache system is:

$$t_{av} = h_1 tL_1 + h_2 tL_2 + (1 - h_1 - h_2) t_{main}$$

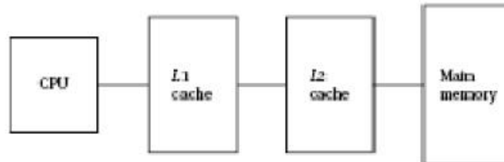


FIGURE 3.7
A two-level cache system.

ii. Direct-mapped cache

The simplest way to implement a cache is a ***direct-mapped cache***, as shown in Figure below. The cache consists of cache ***blocks***, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is divided into three sections. The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location. If the length of the data field is longer than the minimum addressable unit, then the lowest bits of the address are used as an offset to select the required value from the data field. Given the structure of the cache, there is only one block that must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache. Writes are

slightly more complicated than reads because we have to update main memory as well as the cache. There are several methods by which we can do this. The simplest scheme is known as ***write-through***—every write changes both the cache and the corresponding main memory location (usually through a write buffer). This scheme ensures that the cache and main memory are consistent, but may generate some additional main memory traffic. We can reduce the number of times we write to main memory by using a ***write-back*** policy: If we write only when we remove a location from the cache, we eliminate the writes when a location is written several times before it is removed from the cache.

The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power due to its simple scheme for mapping the cache onto main memory. Consider a direct-mapped cache with four blocks, in which locations 0, 1, 2, and 3 all map to different blocks. But locations 4, 8,12,…all map to the same block as location 0;locations 1, 5, 9,13,…all map to a single block;and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. This can create program performance problems.

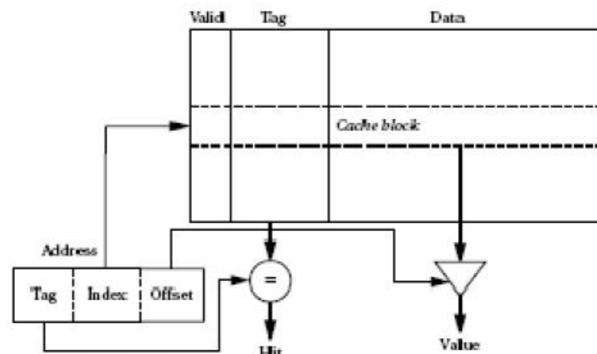


FIGURE 3.8
A direct-mapped cache.

iii. Set-associative cache

The limitations of the direct-mapped cache can be reduced by going to the
set-associative cache structure shown in Figure below. A set-associative cache is characterized by the number of ***banks*** or ***ways*** it uses, giving an *n*-way set-associative cache. A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit. Although memory locations map onto blocks using the same function, there are *n* separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that happen to map onto the same cache block. The set associative cache structure incurs a little extra

overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

The set-associative cache generally provides higher hit rates than the direct mapped cache because conflicts between a small number of locations can be resolved within the cache. The set-associative cache is somewhat slower, so the CPU designer has to be careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs have good behavior in the cache. It is relatively easy to determine when two memory locations will conflict in a direct-mapped cache. Conflicts in a set-associative cache are more subtle, and so the behavior of a set-associative cache is more difficult to analyze for both humans and programs.

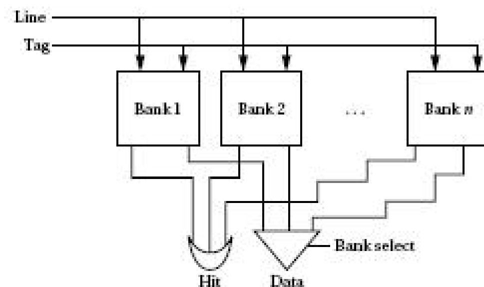


FIGURE 3.9
A set-associative cache.

b) How would the ARM status word be set after the operation: 2^31 - 1 + 1 ?(3)

2^31-1 in 32-bit hex = 0x7fffffff
+1 in 32-bit hex = 0x1
So, (2^31-1)+1 = 0x7fffffff + 0x1 = 0x80000000 = -2^31
Adding 2 positive numbers giving a negative number output, hence an overflow. Since the result is negative, N will also be set.

N: 1
Z: 0
C: 0
V: 1