Embedded Computing Systems

VTU Exam 2017 Solutions

Part A

1.a.

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.

1. b .
   Embedded system design process:

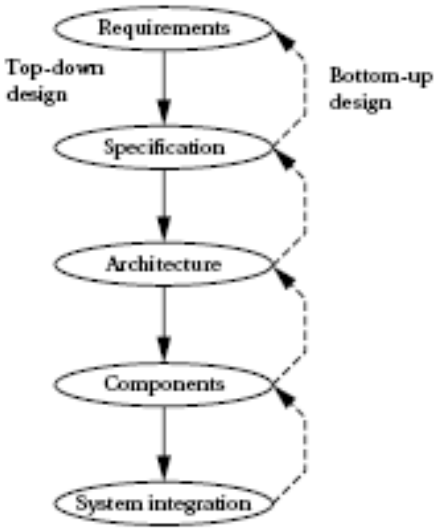


Figure above summarizes the major steps in the embedded system design process. The design process starts with the system ***requirements***, followed by ***specification***, where we create a more detailed description of what we want. But the specification states only how the system behaves, not how 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 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.

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.

Figure 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.

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.

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:

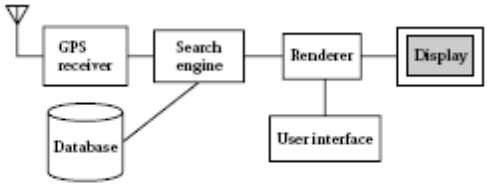


FIGURE 1.3
Block diagram for the moving map.

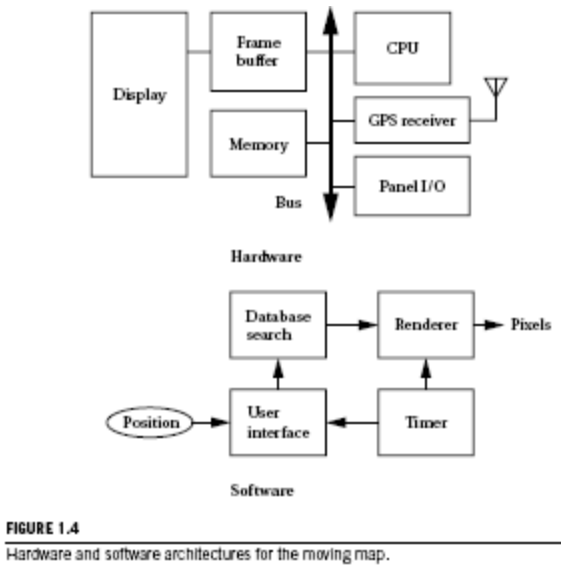


FIGURE 1.4
Hardware and software architectures for the moving map.

Figure above 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.

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.

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.

1.c.

Name	Model train controller
Purpose	Control speed of up to eight model trains
Inputs	Throttle, inertia setting, emergency stop, train number
Outputs	Train control signals
Functions	Set engine speed based upon inertia settings; respond to emergency stop
Performance	Can update train speed at least 10 times per second
Manufacturing cost	$50
Power	10W (plugs into wall)
Physical size and weight	Console should be comfortable for two hands, approximate size of standard keyboard; weight <2 pounds

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.

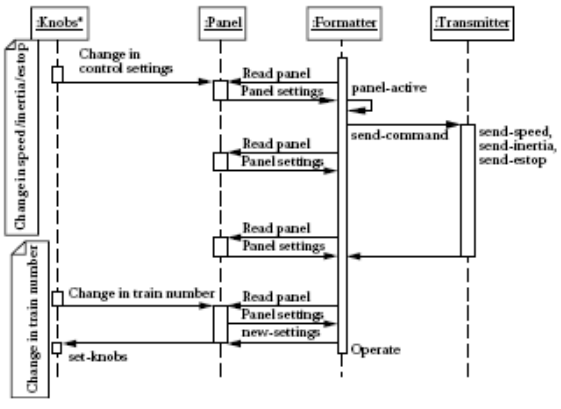


FIGURE 1.24
Sequences diagram for transmitting a control input.

2.a.

	Von Neumann	Harvard
1	The memory holds both data and instructions.	Has separate memories for data and program
2	Has single port to memory, hence, processor needs two clock cycles to fetch an instruction (1 for instruction, and 1 for data)	Has 2 ports, 1 for Data, and 1 for instruction. Hence, instruction and data can be fetched simultaneously.
3	Easy to program, especially self modifying programs.	Harder to write self-modifying programs.
4	Not suitable for real-time data (e.g. DSP).	Very much suitable for real-time data. Gives high performance with DSP.

2.b.

Hit rate h = 93% = 0.93
Tcache = 5 ns
Tmem = 80 ns
Average memory access time, Tav = h*Tcache + (1-h)*Tmem
Tav = 0.93*5ns + (1-0.93)*80ns
Tav = 4.65ns + 5.6ns = 10.25ns

2.c.

(i) Traps:
A ***trap***, also known as a ***software interrupt***, is an instruction that explicitly generates an exception condition. The most common use of a trap is to enter supervisor mode. The entry into supervisor mode must be controlled to maintain security—if the interface between user and supervisor mode is improperly designed, a user program may be able to sneak code into the supervisor mode that could be executed to perform harmful operations.
The ARM provides the SWI interrupt for software interrupts. This instruction causes the CPU to enter supervisor mode. An opcode is embedded in the instruction that can be read by the handler.

(ii) Exceptions:

(iii) An ***exception*** is an internally detected error. A simple example is division by zero. One way to handle this problem would be to check every divisor before division to be sure it is not zero,but this would both substantially increase the size of numerical programs and cost a great deal of CPU time evaluating the divisor's value. The CPU can more efficiently check the divisor's value during execution. Since the time at which a zero divisor will be found is not known in advance, this event is similar to an interrupt except that it is generated inside the CPU. The exception mechanism provides a way for the program to react to such unexpected events.

(iv) Supervisor Mode:

Software debugging is important but can leave some problems in a running system; hardware checks ensure an additional level of safety. In such cases it is often useful to have a ***supervisor mode*** provided by the CPU. Normal programs run in ***user mode***. The supervisor mode has privileges that user modes do not. For example, memory management systems allow the addresses of memory locations to be changed dynamically. Control of the memory management unit (MMU) is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers.

It can be executed conditionally, as with any ARM instruction. SWI causes the CPU to go into supervisor mode and sets the PC to 0x08.The argument to SWI is a 24-bit immediate value that is passed on to the supervisor mode code; it allows the program to request various services from the supervisor mode.

In supervisor mode, the bottom 5 bits of the CPSR are all set to 1 to indicate that the CPU is in supervisor mode. The old value of the CPSR just before the SWI is stored in a register called the ***saved program status register (SPSR)***. There are in fact several SPSRs for different modes; the supervisor mode SPSR is referred to as SPSR_svc. To return from supervisor mode, the supervisor restores the PC from register r14 and restores the CPSR from the SPSR_svc.

2.d.

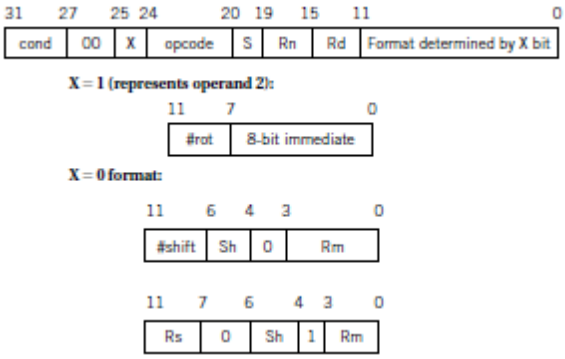


Figure shows the format of an ARM data processing instruction such as an ADD. For the instruction
ADDGT r0,r3,#5
the *cond* field would be set according to the GT condition (1100), the *opcode* field would be set to the binary code for the ADD instruction (0100), the first *operand* register *Rn* would be set to 3 to represent r3, the destination register *Rd* would be set to 0 for r0, and the *operand 2* field would be set to the immediate value of 5.

3.a.

(i) Watchdog timer

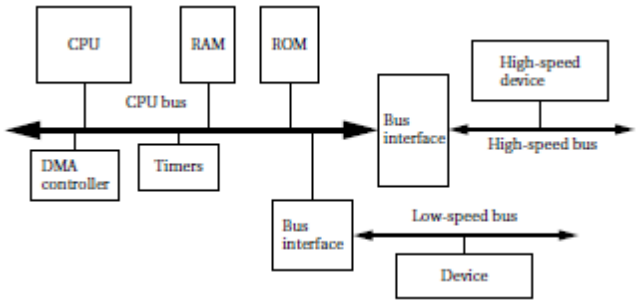
A watchdog timer is an I/O device that is used for internal operation of a system. The watchdog timer is connected into the CPU bus and also to the CPU's reset line. The CPU's

software is designed to periodically reset the watchdog timer, before the timer ever reaches its time-out limit. If the watchdog timer ever does reach that limit, its time-out action is to reset the processor. In that case, the presumption is that either a software flaw or hardware problem has caused the CPU to misbehave. Rather than diagnose the problem, the system is reset to get it operational as quickly as possible.

(ii) Requirement chart of alarm clock

Name	Alarm clock.
Purpose	A 24-h digital clock with a single alarm.
Inputs	Six push buttons: set time, set alarm, hour, minute, alarm on, alarm off.
Outputs	Four-digit, clock-style output. PM indicator light. Alarm ready light. Buzzer.
Functions	Default mode: The display shows the current time. PM light is on from noon to midnight. Hour and minute buttons are used to advance time and alarm, respectively. Pressing one of these buttons increments the hour/minute once. Depress set time button: This button is held down while hour/minute buttons are pressed to set time. New time is automatically shown on display. Depress set alarm button: While this button is held down, display shifts to current alarm setting; depressing hour/minute buttons sets alarm value in a manner similar to setting time. Alarm on: puts clock in alarm-on state, causes clock to turn on buzzer when current time reaches alarm time, turns on alarm ready light. Alarm off: turns off buzzer, takes clock out of alarm-on state, turns off alarm ready light.
Performance	Displays hours and minutes but not seconds. Should be accurate within the accuracy of a typical microprocessor clock signal. (Excessive accuracy may unreasonably drive up the cost of generating an accurate clock.)
Manufacturing cost	Consumer product range. Cost will be dominated by the microprocessor system, not the buttons or display.
Power	Powered by AC through a standard power supply.
Physical size and weight	Small enough to fit on a nightstand with expected weight for an alarm clock.

3.b.



Personal computers are often used as platforms for embedded computing.
As shown in the figure, a typical PC includes several major hardware components:
■ The CPU provides basic computational facilities.
■ RAM is used for program storage.
■ ROM holds the boot program.
■ A DMA controller provides DMA capabilities.
■ Timers are used by the operating system for a variety of purposes.
■ A high-speed bus, connected to the CPU bus through a bridge, allows fast devices to communicate efficiently with the rest of the system.
■ A low-speed bus provides an inexpensive way to connect simpler devices and may be necessary for backward compatibility as well.

PCI (***Peripheral Component Interconnect***) is the dominant high-performance system bus today. PCI uses high-speed data transmission techniques and efficient protocols to achieve high throughput.
USB (***Universal Serial Bus***) and **IEEE 1394** are the two major high-speed serial buses. Both buses offer high transfer rates using simple connectors. They also allow devices to be chained together so that users don't have to worry about the order of devices on the bus or other details of connection.

A PC also provides a standard software platform that provides interfaces to the underlying hardware as well as more advanced services. At the bottom of the software platform structure in most PCs is a minimal set of software in ROM known as the ***basic input/output system*** (***BIOS***). The BIOS provides low-level hardware drivers as well as booting facilities.

3.c.

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.

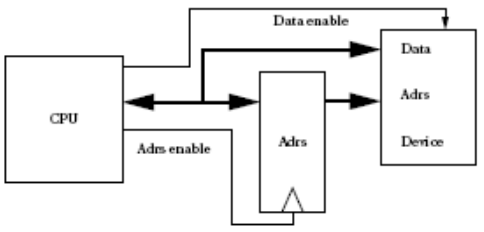


FIGURE 4.8
Bus signals for multiplexing address and data.

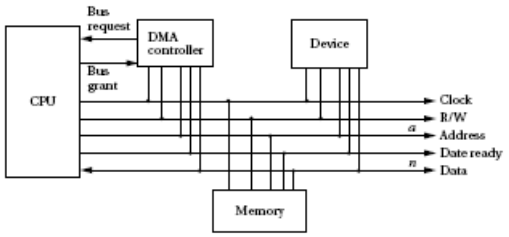


FIGURE 4.9
A bus with a DMA controller.

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. CPU cannot use the bus when DMA is doing a transfer.

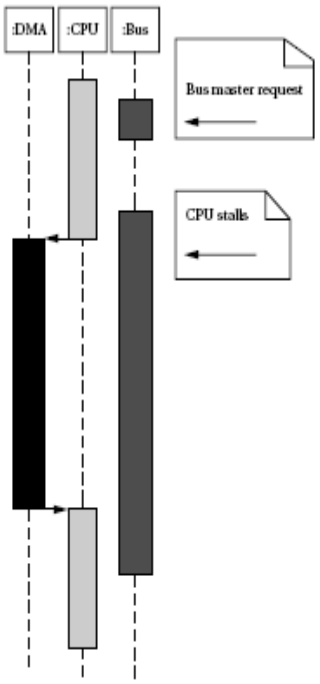


FIGURE 4.10
UML sequence diagram of system activity around a DMA transfer.

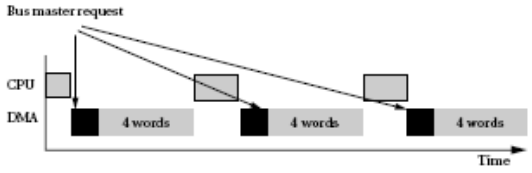


FIGURE 4.11
Cyclic scheduling of a DMA request.

4.a.

Expression Simplification:

This is a useful area for machine-independent transformations. Laws of algebra are used to simplify expressions. For example, distributive law is applied to rewrite the following expression:

a*b+a*c;

Re-written as a*(b+c);

Dead Code Elimination

Code that will never be executed can be safely removed from the program. The general problem of identifying code that will never be executed is difficult, but there are some important special cases where it can be done.

Programmers will intentionally introduce dead code in certain situations.

Consider this C code fragment:

```
#define DEBUG 0
...
if (DEBUG) print_debug_stuff();
```

In the above case, the print_debug_stuff( ) function is never executed, but the code allows the programmer to override the preprocessor variable definition (perhaps with a compile-time flag) to enable the debugging code. This case is easy to analyze because the condition is the constant 0, which C uses for the false condition.

Since there is no else clause in the if statement, the compiler can totally eliminate the if statement, rewriting the CDFG to provide a direct edge between the statements before and after the if.

Some dead code may be introduced by the compiler. For example, certain optimizations introduce copy statements that copy one variable to another. If uses of the first variable can be replaced by references to the second one, then the copy statement becomes dead code that can be eliminated.

4.b.

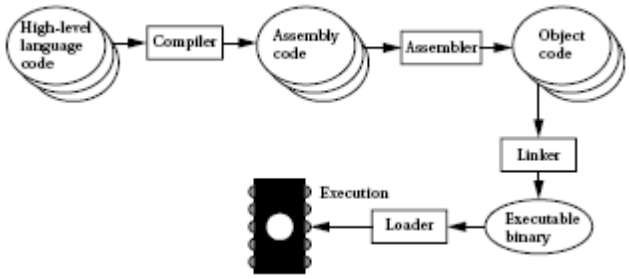


Figure above highlights the role of assemblers and linkers in the compilation process. This process is often hidden from us by compilation commands that do everything required to generate an executable program. As the figure shows, most compilers do not directly generate machine code, but instead create the instruction-level program in the form of human-readable assembly language. Generating assembly language rather than binary instructions frees the compiler writer from details extraneous to the compilation process, which includes the instruction format as well as the exact addresses of instructions and data. The assembler's job is to translate symbolic assembly language statements into bit-level representations of instructions known as ***object code***. The assembler takes care of instruction formats and does part of the job of translating labels into addresses. However, since the program may be built from many files, the final steps in determining the addresses of instructions and data are performed by the linker, which produces an ***executable binary*** file. That file may not necessarily be located in the CPU's memory, however, unless the linker happens to create the executable directly in RAM. The program that brings the program into memory for execution is called a ***loader***. The simplest form of the assembler assumes that the starting address of the assembly language program has been specified by the programmer. The addresses in such a program are known as ***absolute addresses***. However, in many cases, particularly when we are

creating an executable out of several component files, we do not want to specify the starting addresses for all the modules before assembly—if we did, we would have to determine before assembly not only the length of each program in memory but also the order in which they would be linked into the program. Most assemblers therefore allow us to use ***relative addresses*** by specifying at the start of the file that the origin of the assembly language module is to be computed later. Addresses within the module are then computed relative to the start of the module. The linker is then responsible for translating relative addresses into addresses.

4.c.

A ***data flow graph*** is a model of a program with no conditionals. In a high-level programming language, a code segment with no conditionals—more precisely, with only one entry and exit point—is known as a basic block. Before we are able to draw the data flow graph for this code we need to rewrite the code in ***single-assignment form***, in which a variable appears only once on the left side.

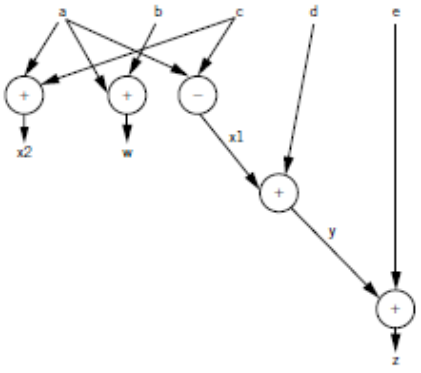
```
w = a + b;
x = a - c;
y = x + d;
x = a + c;
z = y + e;
```

FIGURE 5.2
A basic block in C.

```
w = a + b;
x1 = a - c;
y = x1 + d;
x2 = a + c;
z = y + e;
```

FIGURE 5.3
The basic block in single-assignment form.

The single-assignment form is important because it allows us to identify a unique location in the code where each named location is computed. The data flow graph is generally drawn in the form shown in figure below. Here, the variables are not explicitly represented by nodes. Instead, the edges are labelled with the variables they represent. As a result, a variable can be represented by more than one edge. However, the edges are directed and all the edges for a variable must come from a single source. We use this form for its simplicity and compactness. The data flow graph for the code makes the order in which the operations are performed in the C code much less obvious. This is one of the advantages of the data flow graph. We can use it to determine feasible reorderings of the operations, which may help us to reduce pipeline or cache conflicts. We can also use it when the exact order of operations simply doesn't matter. The data flow graph defines a partial ordering of the operations in the basic block. We must ensure that a value is computed before it is used, but generally there are several possible orderings of evaluating expressions that satisfy this requirement.



Part B

5.a.

RTOS stands for Real-Time Operating System, which is a type of operating system that implements policies and rules concerning time-critical allocation of system resources. RTOS decides which applications should run in which order, and how much time needs to be allocated for each application. E.g. Windows CE, QNX, VxWorks MicroC/OS-II.

Services of RTOS:

1. Real-time Kernel
    a. Task/Process management
    b. Task/Process scheduling
    c. Task/Process synchronization
    d. Error/Exception handling
    e. Primary and Secondary Memory Management
    f. File System Management
    g. I/O system/ Device Management
    h. Interrupt Handling
    i. Time Management
    j. Protection systems
2. Hard Real-Time
3. Soft-Real-Time

5.b.

A task is a program or a part of it in execution. It is synonymous with job or process.
A Task Control Block (TCB) is a data structure that is used for holding the information corresponding to a task.
Structure of TCB:
Task ID: Task identification number
Task State: Current state (e.g., running, idle, etc.)
Task type: Hard real-time/soft-real-time, or background task.
Task Priority: Priority of the task
Task Context Pointer: Pointer for saving context.
Task Memory Pointers: Pointer to code memory, data memory, and stack memory
Task System Resource Pointers: Pointers to system resources (semaphores, mutex, etc.)
Task Pointers: Pointers to other TCBs
Other Parameters: Other relevant task parameters.

5.c.

The selection of scheduling algorithm should consider the following factors:

1. CPU Utilization: The scheduling algorithm should always make the CPU utilization high. CPU utilization is a direct measure of how much the CPU is being utilized.

2. Throughput: This gives an indication of the number of processes getting executed per unit time. The throughput of a good scheduler should always be higher.
3. Turnaround time: It is the amount of time for a process to complete execution. It includes the time spent by the process waiting for memory, IO, time spent in ready queue, and time spent in execution. TAT should be minimum for a good scheduling algorithm.
4. Waiting time: It is the amount of time the process spends waiting in the ready queue to get CPU time for execution. WT should be minimum for a good scheduling algorithm.
5. Response time: It is the time elapsed between the submission of a process and the first response. Response time should be minimum for a good scheduling algorithm.

5.d.

(i) Task: Task refers to something that needs to be done. In OS context, task is defined as a program in execution and the related information maintained by the OS for the program.

(ii) Process: A program or a part of it in execution is called a process.

(iii) Thread: A thread is a single flow of control within a process. Thread is also known as light weight process. Different threads which are part of the same process share the data memory, code memory and heap memory.

6.a.

Inter Process Communication (IPC) is the mechanism provided by the OS as part of the process abstraction through which the processes/tasks communicate with each other.

Some of the important IPC mechanisms adopted by various kernels are explained below:

1. Shared Memory
   Processes share some area of the memory to communicate by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area.

   Some of the different mechanisms adopted by different kernels are as below:
   a. Pipes: Pipe is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client. It can be unidirectional, allowing information flow in one direction or it can be bidirectional, allowing bi-directional information flow. Generally, there are two types of pipes supported by the OS. They are:
      Anonymous pipes: They are unnamed, unidirectional pipes used for data transfer between two processes.

Names Pipes: They are named, unidirectional or bidirectional for data exchange between two processes.

b. Memory Mapped Objects: This is a shared memory technique adopted by some real-time OS for allocating shared block of memory which can be accesses by multiple process simultaneously. In this approach, a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area ir a block of it to its virtual address space. All read-write operations to this virtual address space by a process is directed to its committed physical area. Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

2. Message Passing

Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread communication. The major difference between shared memory and message passing is that through shared memory lots of data can be shared whereas only limited amount of data is passed through message passing. Also, message passing is relatively fast and free from synchronization overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into:

a. Message Queue: Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'message queue', which stores the message temporarily in a system defined memory object, to pass it to the desired process. Messages are sent and received through *send* and *receive* methods. The messages are exchanged through the message queue. It should be noted that the exact implementation is OS dependent. The messaging mechanism is classified into synchronous and asynchronous based on the behavior of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted. Whereas in synchronous messaging, the thread which the message is posts the message enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

b. Mailbox: Mailbox is an alternative to 'message queues' used in certain RTOS for IPC, usually used for one way messaging. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification. The process of creation, subscription, message reading and writing are achieved through OS kernel provided API calls.

c. Signaling: Signaling is a primitive way of communication between processes/threads. *Signals* are used for asynchronous notifications where one process/trhead fires a

signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data.

3. Remote Procedure calls and Sockets
   Remote Procedure Call (RPC) is the IPC mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In object oriented language terminology RCP is also known as *Remote Method Invocation (RMI)*. RPC is mainly used for distributed applications like client-server applications. The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client.
   Sockets are used for RPC communication. Socket is a logical endpoint in a two-way communication link between two applications running on a network. Sockets are of different types, namely, Internet Sockets (INET), UNIX sockets, etc. The INET sockets works on internet communication protocols, such as TCP/IP and UDP. They are classified into stream sockets and datagram sockets.
   Stream sockets are connection oriented, and they use TCP to establish a reliable connection.

Datagram sockets rely on UDP for communication. The UDP connection is unreliable when compared to TCP.

6.b.

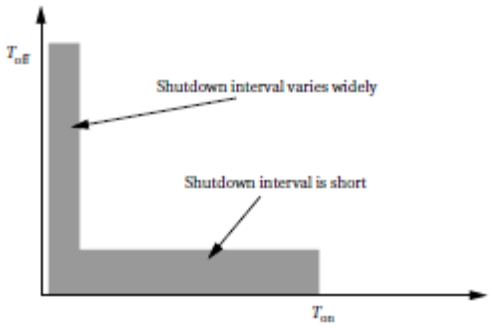
(i) ACPI: Advanced Configuration and Power Interface (ACPI) is an open industry standard for power management services. ACPI provides some basic power management facilities and abstracts the hardware layer, the OS has its own power management module that determines the policy, and the OS then uses ACPI to send the required controls to the hardware.

ACPI supports the following five basic global power states:

- G3, the mechanical off state, in which the system consumes no power.
- G2, the soft off state, which requires a full OS reboot to restore the machine to working condition. This state has four sub-states"
  - S1, a low wake up latency state with no loss of system context.
  - S2, a low wake up latency state with loss of CPU and system cache state.
  - S3, a low wake up latency state in which all system states except for main memory is lost, and
  - S4, the lowest power sleeping state in which all devices are turned off.
- G1, the sleeping state in which the system appears to be off and the time required to return to working condition is inversely proportional to power consumption.
- G0, the working state in which the system is fully usable.

The legacy state in which the system does not comply with ACPI.

(iii) L-shaped Graph:



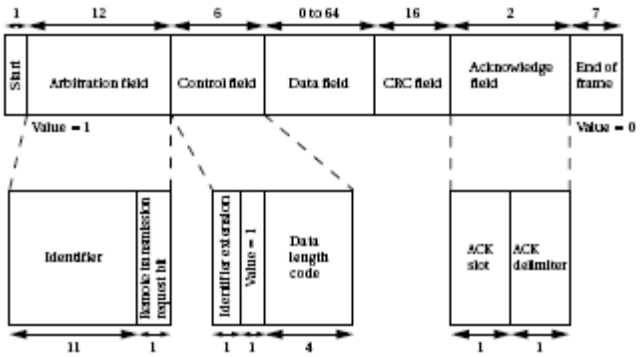
A very simple technique is to use fixed times. For instance, if the system does not receive inputs during an interval of length *T*on, it shuts down; a powered-down system waits for a period *T*off before returning to the power-on mode. The choice of *T*off and *T*on must be determined by experimentation. Srivastava and Eustace [Sri94] found one useful rule for graphics terminals. They plotted the observed idle time (*T*off*)* of a graphics terminal versus the immediately preceding active time (*T*on*)*. The result was an L-shaped distribution as illustrated in Figure 6.17. In this distribution, the idle period after a long active period is usually very short, and the length of the idle period after a short active period is uniformly distributed. Based on this distribution, they proposed a shutdown threshold that depended on the length of the last active period—they shut down when the active period length was below a threshold, putting the system in the vertical portion of the *L* distribution.

7.a.

An embedded system built around a network or one in which communication between processing elements is explicit.

An important advantage of a distributed system with several CPUs is that one part of the system can be used to help diagnose problems in another part. Whether you are debugging a prototype or diagnosing a problem in the field, isolating the error to one part of the system can be difficult when everything is done on a single CPU. If you have several CPUs in the system you can use one to generate inputs for another and to watch its output.

7.b.



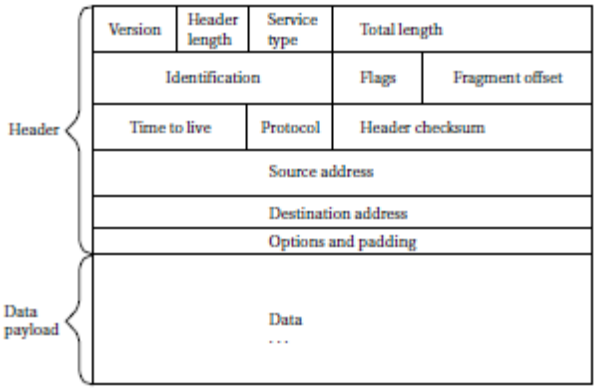
A data frame starts with a 1 and ends with a string of seven zeroes. (There are at least three bit fields between data frames) The first field in the packet contains the packet's destination address and is known as the arbitration field. The destination identifier is 11 bits long. The trailing remote transmission request (RTR) bit is set to 0 if the data

frame is used to request data from the device specified by the identifier. When RTR_1, the packet is used to write data to the destination identifier. The control field provides an identifier extension and a 4-bit length for the data field with a 1 in between. The data field is from 0 to 64 bytes, depending on the value given in the control field. A cyclic redundancy check (CRC) is sent after the data field for error detection. The acknowledge field is used to let the identifier signal whether the frame was correctly received: The sender puts a recessive bit (1) in the ACK slot of the acknowledge field; if the receiver detected an error, it forces the value to a dominant (0) value. If the sender sees a 0 on the bus in the ACK slot, it knows that it must retransmit. The ACK slot is followed by a single bit delimiter followed by the end-of-frame field.

7.c.

(i) I2C

(ii) IP



8.a.

1. List file (.lst)
2. Preprocessor output file
3. Object file (.obj)
4. Map file (.map)
5. Hex file (.hex)

i. .obj file

Cross-compiling/assembling each source file converts the various embedded C/assembly instructions and other directives present in t he module to an object (.OBJ) file. The format of the .OBJ file is cross-compiler dependent. OMF51 or OMF2 are the two objects file formats supported by C51 cross compiler. The object

file is a specially formatted file with data records for symbolic information, object code, debugging information, library references, etc.

The list of some of the details stored in an object file is given below.

1. Reserved memory for global variables
2. Public symbol (variable and function) names
3. External symbol (variable and function) names
4. Library files with which to link
5. Debugging information to help synchronize source lines with object code.

The object code present in the object file are not absolute, meaning, the code is not allocated fixed memory location in the code memory. It is the responsibility of the linker/locator to assign an absolute memory location to the object code. Suring cross-compilation process, the cross compiler sets the address of references to external variables and functions as 0. The external references are resolved by the linker during the linking process. Hence, it is obvious that the code generated by the cross-compiler is not executable without linking it for resolving external references.

ii. .map file

In a project with multiple source files, the cross-compilation of each module generates a corresponding object file. The object files so created are re-locatable codes, meaning their location in the code memory is not fixed. It is the responsibility of a linker to link all these object files. The locator is responsible for locating absolute address to each module in the code memory. Linking and locating of re-locatable object files will also generate a list file called 'linker list file' or a 'map file'. Map file contains information about the link/locate process and is composed of a number of sections. The different sections listed in a map file are cross compiler dependent. The information generally held by map files is listed below. It is not necessary that the map file must contains these.

Page header: Indicates the linker version number, date, time, and page number.

Command line: The command that was used for invoking the linker.

CPU details: Details about the target CPU and memory model (internal, external, and paged data memory).

Input Modules: This section includes the names of all object modules, and library files and modules that are included in the linking process.

Memory Map: Lists the starting address, length, relocation type and name of each segment in the program.

Symbol table: Contains the value, type, and name for all symbols from the different input modules.

Inter module cross reference: Includes section name, memory type, and the name of the modules in which it is defined and all modules in which it is accessed.

Program size: Contains the size of various memory areas as well as constant and code space for the entire application.

Warnings and Errors: Errors and warnings that are generated while linking a program are written to this section. It is useful for debugging link errors.

iii. .hex file

Hex file is the binary executable file created from the source code. The absolute object file created by the linker/locater is converted into processor understandable

binary code. The utility used for converting an object file to a hex file is known as Object to Hex file converter. Hex files embed the machine code in a particular format. The format of Hex files varies across the family of processors/controllers. Intel HEX and Motorola HEX are the two commonly used hex file formats in embedded applications. Intel HEX file is an ASCII tect file in which the HEX data is represented in ASCII format in lines. The lines in an Intel HEX file are corresponding to a HEX record. Each record is made up of hexadecimal numbers that represent machine-language code and/or constant data. Individual records are terminated with a carriage return and a linefeed. Intel HEX file is used for transferring the program and data to a ROM or EPROM which is used as code memory storage.

iv. .lst file

Listing file is generated during the cross compilation process and it contains an abundance of information about the cross compilation process, like cross compiler details, formatted source text (C code), assembly code generated from the source file, symbol tables, errors, and warnings detected during the cross-compilation process. The type of information contained the list file is cross-compiler specific. The list file contains the following sections:

Page header: A header on each page of the listing file which indicates the compiler version number, source file name, date, time, and page number.

Command Line: Represents the entire command line that was used for invoking the compiler.

Source code: The source code listing outputs the line number as well as the source code on that line. Special cross compiler directives can be used to include or exclude the conditional codes in the source code listings. Apart from the source code lines, the list file will include the comments in the source file and depending on the list file generation settings the entire contents of all include files may also be included. Special cross compiler directives can be used to include the entire contents of the include file in the list file.

Assembly listing: This contains the assembly code generated by the cross compiler for the C source code. Assembly code generated can be excluded from the list file by using special compiler directives.

Symbol listing: This contains symbolic information about the various symbols present in the cross compiled source file.

Module information: The module information provides the size of initialized and un-initialized memory areas defined by the source file.

Warnings and Errors: This section records the errors encounters or any statement that may create issues in application (warnings), during cross-compilation. The warning levels can be configured before cross-compilation. You can ignore certain warnings, while some require prompt attention.

(V) Preprocessor output file:

Contains pre-processor output for pre-processor instruction. To verify the operation of macros/ conditional pre-processor directives. The output of a pre-processor file is a valid C source file.

The functions are:

1. Inclusion of header files
2. Macro substitution
3. Conditional compilation

8.b.

Simulator based debugging:

Advantages:

- No need for original target board: Simulators are purely software based. IDEs software support simulates the CPU of the target boards. Since real hardware is not required, firmware can start well in advance once device interface and memory maps are finalised. This saves development time.
- Simulate IO peripherals: Simulator allows to simulate various IO peripherals. One can edit the value of IO registers and can be used as IO value in firmware execution, eliminating the need for connecting for debugging the firmware.
- Simulates abnormal conditions: Once can simulate abnormal operational environment for firmware.

Disadvantages:

- Deviation from real behaviour: Developer may not be able to debug the firmware under all possible combinations of input.
- Lack of real timeliness: It is not real-time in behaviour. The debugging is developer driven and it is no way capable of creating a real-time behaviour. Simulation only works for known values.

8.c.

Target System: The hardware on which the code will finally run is known as the target. The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should generally leave the smallest possible footprint in the target to avoid interfering with the application software.

Host System: In typical embedded computing system development, it is common to do at least part of the software development on a PC or workstation. This system is known as the host. The host should be able to do the following:

■ load programs into the target,
■ start and stop program execution on the target, and
■ examine memory and CPU registers.

END