PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053

Embedded Systems 1

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

Syllabus

PEN Code: **CPE 053** **Credit:** 3 units
PEN Subject Title: Embedded System 1 **Prerequisite:** 3rd yr standing

A. Course Description:

This course provides advanced topics in embedded systems design using contemporary practice; interrupt-driven, reactive, real-time, object-oriented, and distributed client/server embedded systems.

B. Objectives:

At the end of this term, you should be able to design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability, in accordance with standards.

C. Subject Outline and Time Allotment:

Lesson No.	Topics	Week
1	Introduction to Embedded and Real-Time Systems	1
2	Cross-Platform Development Process (Part 1)	
3	Software Development Methods: Cross-Platform Development Process (Part 2)	2
Quiz	Lessons 1 to 3	3
4	Microprocessor Primer (Part 1)	
5	Microprocessor Primer (Part 2)	4
	P1 Exam	5
6	Interrupts	6
Quiz	Lessons 6	7
7	Embedded System Boot Process	
8	Architecture Modeling in UML (Part 1)	8
	P2 Exam	9
9	Architecture Modeling in UML (Part 2)	10
10	Fundamental UML Behavioral Modeling (Part 1)	11
Quiz	Lessons 9 to 10	
11	Fundamental UML Behavioral Modeling (Part 2)	12

	P3 Exam	13

D. References:

1. Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015
2. Embedded System Design: A Unified Hardware/Software Introduction" by Frank Vahid and Tony Givargis
3. Embedded Systems: Design and Applications" by Barrett

E. Course Requirements

Course Requirements	**For board preparation subjects only.** Part of passing a board preparation subject is passing its final examination. Thus, a student who gets a passing final grade based on the formula despite getting a failing grade in the final examination will get a grade of Incomplete (INC) in the subject. The student may change his INC grade to a passing grade by retaking and passing the final examination. The passing grade that the student gets in his retake of the final examination will be used to compute his final grade in the subject. His previous failing grade in the said final examination will be disregarded.
	If in one academic year the student fails to change his grade from an INC to a passing grade, he will be given a grade of No Credit (NC).
	Periodic examinations, quizzes, assignments, and problem set are given to students. Students who miss an exam will be allowed to take a makeup term examination and that is only be given within five (5) days after the original date of the term examination and those students who is still not able to take the makeup term examination will get a failing grade for the corresponding term examination.

F. Grading System:

The Final Grade is computed as follows:
FG= (33% P1) + (33% P2) + (34% P3)

The 1st Periodical Grade is computed as follows:
P1 = (60% CLASS STANDING) + (40% EXAM)

The 2nd Periodical Grade is computed as follows:
P2 = (60% CLASS STANDING) + (40% EXAM)

The 3rd Periodical Grade is computed as follows:
P3 = (60% CLASS STANDING) + (40% EXAM)

Class Standing = 50% QUIZ + 50% Class Participation
Class Participation = Seatwork, Assignment, Problem set

Passing Grade: 50

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #1

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

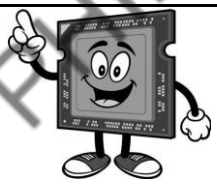
Lesson title: Introduction to Embedded and Real-Time Systems
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand what is an embedded system;**
2. **Understand Real-Time Systems;**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
Track your time! It's difficult to plan your personal time if you don't know how you're spending it. Tracking your hours is an important step toward better time management. Start by breaking down a major project into manageable individual tasks.



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

"To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be."

In computing disciplines (computer engineering, software engineering, computer science, information systems and technology), the term "embedded system" is used to refer to an electronic system that is designed to perform a dedicated function and is often embedded within a larger system.

This lesson first discusses what embedded system is all about. Then we give a brief preview of what real-time systems are.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Embedded System?	
	2 Why are Real-Time Systems?	

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

1.1 Embedded Systems

In computing disciplines (computer engineering, software engineering, computer science, information systems and technology), the term “embedded system” is used to refer to an electronic system that is designed to perform a dedicated function and is often embedded within a larger system.

Embedded systems differ from general-purpose computing devices mainly in two aspects:

• First, an embedded system is designed simply for a specific function, whereas a general-purpose computing device, such as smartphone, laptop, or desktop computer, is not; they can be used as Web servers or data warehouses, or can be used for writing articles, reading news, playing games, or running scientific experiments, to mention only a few applications.

• Second, an embedded system is traditionally built together with the software intended to run on it. Such a parallel model of developing hardware and software together is known as hardware-software co-design. Recently, there has been a trend where an embedded system is built with a well-defined interface open to third-party embedded software providers. In contrast, a general-purpose computing device is often built independently from the software applications that may run on it.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

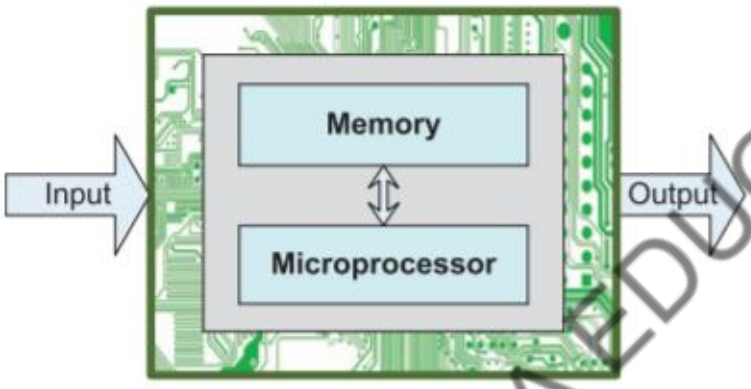


Figure 1.1
A generic embedded system architecture.

An embedded system is a combination of computer hardware and software, and sometimes mechanical components as well. Figure 1.1 gives a bird's-eye view of a generic embedded system architecture, where the microprocessor and the memory blocks are the heart and the brain, respectively. Embedded software is commonly stored in 3on-volatile memory devices such as read-only memory (ROM), erasable programmable ROM (EPROM), and flash memory. The microprocessor also needs another type of memory—random-access memory (RAM)—for its run-time computation.

When an embedded system is powered on, its microprocessor reads software instructions stored in memory, executes the instructions to process input information from peripheral components (through sensors, signals, buttons, etc.), and produces output to meet the needs of the external embedding system.

Given that the hardware components are chosen, most of the design effort is in the software, including application, device drivers, and sometimes an operating system. In many cases, it is possible to build a customized integrated circuit (IC) that is functionally equivalent to an embedded system. An IC-based solution is a hardwired solution that does not contain software and a microprocessor. However, the embedded system solution is more flexible and less expensive, especially when the product needs to be frequently upgraded to accommodate new changes. In response to a new change, for the hardwired solution, a new circuit needs to be designed, constructed, and delivered. In contrast, for the embedded system solution, software patches can be rapidly developed, and the upgrading process can be done over the Internet and

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

may typically take just a few seconds.

Embedded systems are widespread in consumer, industrial, medical, and military applications. Just look around your home or workplace, and you may realize that almost every aspect of our everyday life has been wonderfully touched by embedded systems: dishwasher, garage door opener, TV remote control, microwave oven, programmable thermostat, Xbox controller, and USB memory card reader. This list goes on and on.

Someone said that there are more computers in our homes and offices than there are people who live and work there. If this is true, then there are even more embedded systems that have been and will continue changing every part of our lives. One proof of this statement is that about 98% of microprocessors go into embedded systems, whereas less than 2% of microprocessors are used in computers.

1.2 Real-Time Systems

There are systems that need to respond to a service request within a certain amount of time: they are called real-time systems [22, 45]. To a real-time system, each incoming service request imposes a task (job) that is typically associated with a real-time computing constraint, or simply called its timing constraint.

The timing constraint of a task is normally specified in terms of its deadline, which is the time instant by which its execution (or service) is required to be completed. Depending on how serious missing a task deadline is, a timing constraint can be either a hard or a soft constraint:

• A timing constraint is hard if the consequence of a missed deadline is fatal. A late response (completion of the requested task) is useless, and sometimes totally unacceptable.

• A timing constraint is soft if the consequence of a missed deadline is undesirable but tolerable. A late response is still useful as long as it is within some acceptable range (say, it occurs occasionally with some acceptably low probability).

Actual systems may have both hard and soft timing constraints. A system in which all tasks have soft timing constraints is a soft real-time system. A system is a hard real-time system if its key tasks have hard timing constraints.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1.2.1 Soft Real-Time Systems

A soft real-time system offers best-effort services; its service of a request is almost always completed within a known finite time. It may occasionally miss a deadline, which is usually considered tolerable. It is worth noting that although missing a deadline will not cause catastrophic effects, the usefulness of a result may degrade after its deadline, thereby degrading the system's quality of service.

Soft timing constraints are typically expressed in probabilistic or statistical terms, such as average performance and standard deviation. Table 1.1 gives some example soft real-time systems.

Table 1.1 Example soft real-time systems

Example System	Example Timing Constraint	Consequence of Missed Deadlines
Digital camera	Shutter speed, shown in seconds or fractions of a second, is a measurement of the time the shutter is open. When the shutter speed is set to 0.5 s, the shutter open time should be (0.5 ± 0.125) s 99.9% of the time	Unsatisfied users may switch to other models
Global positioning system	Upon identifying a waypoint, it can remind the driver at a latency of 1.5 s	The driver misses the waypoint
Robot-soccer player	Once it has caught the ball, the robot needs to kick the ball within 2 s, with the probability of breaking this deadline being less than 10%	Its team may lose the game
Wireless router	The average number of late/lost frames is less than 2/min	The user has bad Web surfing experience

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1.2.2 Hard Real-Time Systems

In a hard real-time system, missing some deadlines is completely unacceptable, because this could result in catastrophic effects such as safety hazards or serious financial consequences.

A hard real-time system offers guaranteed services. Hence, the correctness of a hard real-time system is twofold: functional correctness and timing correctness. Here, the timing correctness of a system means that its service of a request is guaranteed to be completed within a strict deadline. Actually, in most cases timing correctness is even more important than functional correctness, because a partially functional system may be used as is and still has its values, whereas a fully functional system is useless if the offered services have no guaranteed service completion time.

Since breaking a hard timing constraint is unaffordable, it is normally a requirement that the designers/developers of a hard real-time system should validate rigorously that the system can meet its hard timing constraints. In the literature, the proof techniques include design-time schedulability analysis, exhaustive simulation, combinatorial performance testing, and symbolic reasoning tools based on temporal logics (e.g., model checking). While many of these techniques are beyond the scope of this book, we will cover basic approaches to schedulability analysis when it comes to real-time scheduling.

Hard timing constraints are typically expressed in deterministic terms. Table 1.2 gives some example hard real-time systems.

What if a system at run time anticipates that a deadline might be missed? A hard real-time system will try its very best to avoid such a bad thing happening. For example, an antimissile system may have implemented two ways of locking onto an incoming missile: one takes a longer but can calculate precise firing coordinates, and the other takes less time but can compute only an approximate firing range, which demands more weapons being activated to cover the firing range. Since the first approach consumes fewer resources, it is the default approach employed by the system to handle incoming missiles. However, the system would switch to the second approach if it predicts that waiting for the precise calculation would take too much time for the incoming missiles to be safely destroyed.

In contrast, for a soft real-time system, it typically does not take any corrective actions until the bad thing really happens. For example, a DVD player has to synchronize the video stream and the audio stream. A missed deadline happens when, owing to data loss or decoding

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

latency, the timing difference of the two streams exceeds a certain tolerable threshold. After a missed deadline has been detected, a DVD player may selectively discard the decoding of some video/audio frames to resynchronize the two streams.

1.2.3 Spectrum of Real-Time Systems

A real-time system is called a real-time embedded system if it is designed to be embedded within some larger system.

Figure 1.2 shows the scope of embedded systems and real-time systems, as well as the spectrum of real-time embedded systems. Note that there is no precise boundary between soft and hard real-time systems. Systems that lie in between the two are often called firm real-time systems. A system, such as the radar system studied in the next section, can be a soft or a hard real-time system, depending on where and how it is to be used. For example, a radar system is definitely a hard real-time system when it is part of a military surveillance system; it can be deemed a soft real-time system when it is used for weather forecasting or for monitoring meteorological precipitation.

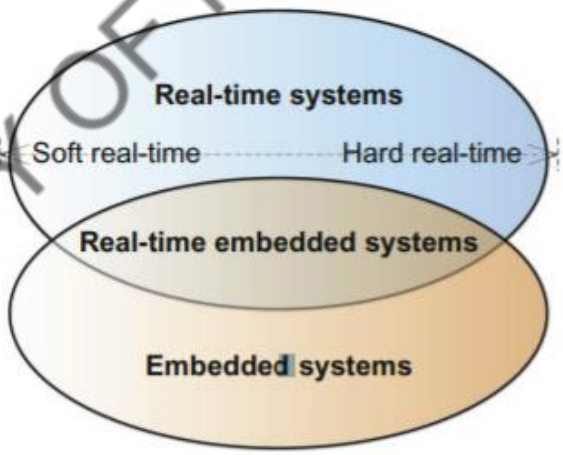


Figure 1.2
System classification.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1.2 Case Study: Radar System

Radar (radio detection and ranging) is an object-detection system that uses radio waves or microwaves to determine the distance range, altitude, direction, or speed of objects. Radar has been widely used in many areas, including airport traffic control systems, aviation control systems, military surveillance systems, antimissile systems, and meteorological precipitation monitoring.

Radar is a very complex electronic and electromagnetic system [51, 58, 70]. Below we briefly describe the functionality of each subsystem. Readers may choose to skip the details for now, but will find them useful when we use radar as an example to explain advanced modeling concepts later in the book.

As shown in Figure 1.3, a radar system is typically composed of several subsystems.

The radio frequency (RF) subsystem generally consists of an antenna, an antenna feed, a duplexer, and some preselector filters. The antenna is the interface with the medium of radio wave propagation, and it is the first stage during signal reception and the last stage during signal transmission. The antenna feed collects energy as it is received from the antenna, or transmits energy as it is transmitted to the antenna. The duplexer switches the radar system between transmit mode and receive mode. The switch operation must be extremely rapid—say, within 5 ms (a real-time constraint)—in order to detect or track fast-moving targets. The switch operation frequency can be adjusted by an operator through the controller, which is part of the data processing subsystem. Preselector filters are used to attenuate out-of-band signals and images during the signal reception phase, and during transmission they are used to attenuate harmonics and images.

The transmitter subsystem consists of a digital waveform generator, an upconverting mixer, and a power amplifier. The digital waveform generator reads a desired waveform design and uses a D/A converter to produce analog signals in the baseband frequency range (intermediate frequency). The generated signals must conform to the timing constraints specified in the design. The upconverting mixer is used to transform the baseband frequency signals into RF signals. The power amplifier is used to amplify the RF signals for transmission.

Figure 1.4 illustrates the radar signal in the time domain. The pulse width (or pulse duration) of the transmitted signal has to be long enough so that the radar can emit sufficient energy to ensure that the reflected pulse is detectable by the receiver. In such a sense, the pulse width

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

constrains the radar's maximum detection range. The pulse width also determines the dead zone at close range. A radar echo takes approximately 1 μs to return from a target 150 m away. This means if the radar pulse width is 1 μs, the radar will not be able to detect targets closer than 150 m because the receiver is blanked while the transmitter is active.

The pulse repetition frequency (PRF) is the rate of pulse transmission. Most radar systems emit pulses multiple times in each direction so that multiple echo signals can be received. While a radar system might not be able to detect a target on the basis of simply one echo signal, the integration of multiple reflected pulses (of similar amplitudes) can reinforce the return signals and make detection easier. PRF can range from a few kilohertz to tens or hundreds of kilohertz. For example, a radar with a 1◦ horizontal beamwidth that sweeps the entire 360◦ horizon every second with a PRF of 2160 Hz can emit six pulses over each 1◦ arc; it thus can receive six echoes from each target within the detection range. Note that the (unambiguous) detection range is reduced as the PRF is increased because the receiver active period is shortened. A lower PRF gives a longer detection range, but often suffers from poorer target painting and velocity ambiguity.

Modern radar systems often use a technique called PRF staggering. With staggered PRF, the transmitter can use different PRFs to produce packets: a packet of pulses is transmitted with a fixed interval between pulses, and then another packet is transmitted with a slightly different interval. By PRF staggering, a radar can force the "jamming" signals from other radar systems to jump around erratically, inhibiting integration and thus suppressing their impacts on target detection. The PRF parameters can be set up by human operators and automatically adjusted by the controller of the data processing subsystem.

The receiver subsystem consists of a low-noise amplifier (LNA) and a downconverting mixer. After signal transmission, a radar system typically positions its antenna in that direction for a few milliseconds up to a few hundreds of milliseconds to collect the echo signals (such a dwell time is another example of a real-time constraint). Echo signals often contain noises. The LNA is used to separate desired signals from undesired signals, such as thermal noise, clutter (echoes returned from objects that are not targets of interest, such as precipitation and birds), and interference (signals originating from active sources other than the radar). The LNA can also boost the power of the desired signals without introducing undesired distortions. The downconverting mixer is used to transform the RF signals into baseband frequency signals. Finally, the receiver subsystem uses A/D converters to sample an analog signal and save the digital values in shared memory accessible to the signal processing subsystem.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The antenna position subsystem is used to automatically rotate and position the radar antenna. The error indicator detects the error angle between the output shaft and the input shaft, and applies a feedback signal to the antenna positioner, which causes the servomotor to exert a torque on the output shaft in a direction to reduce the error. Through the controller, an operator can change the operation mode of the antenna positioner (say, from searching to tracking a target), or can alter the scan strategy (say, a conical scan, a unidirectional sector scan, or a circular scan).

Echoes from targets must be detected and processed before the transmitter emits the next pulse. After transmission of a pulse, if there is a reflective object (say, an unmanned aerial vehicle) at a distance of x meters from the antenna, the echo signal reflected by the object returns to the antenna in approximately 2x/c s, where c = 3 × 108 m/s is the speed of light in a vacuum. To measure the target distance, the radar's detection range is divided into many range intervals or range bins, the length of which is equal to the desired range resolution (say, 200 m). The antenna's pulse repetition period can then be divided into many time frames, where the length of a time frame equals the time it takes the radio signal to propagate exactly one range interval. The digital values collected within each time frame form one sample of the situation in the corresponding range bin. Since the radar can transmit pulses multiple times in each direction, multiple samples for each range bin can be collected, and these are the inputs to the signal processing subsystem.

In general, the number of range bins can be in the hundreds or even thousands, and PRFs can range from a few to tens or hundreds of kilohertz. Such a high-demanding real-time constraint typically requires many digital signal processors (DSP) to form a computing cluster. The DSP cluster needs to produce a discrete Fourier transform of the samples for each range bin, and calculate the frequency spectrum of the echoes.

The object detector can determine whether there are objects in the direction in which the antenna is pointing and their positions and velocities, if applicable. If there is a moving object, the frequency of the reflected signal must be different from the frequency of the transmitted signal, which is called Doppler shift. The amount of Doppler shift is proportional to the velocity of the object.

A statistical hypothesis testing approach can be used to determine whether or not a measurement represents the influence of a target or merely interference [58]. The decision rule is simple: if the calculated "likelihood ratio" exceeds the detection threshold, declare a target to be present; otherwise, declare that a target is not present. Notice that the object detector

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

relies on several unknown parameters that need to be replaced by their maximum likelihood estimates. The probability distribution functions that form the likelihood ratio may depend on one or more parameters. The position of the threshold can be dynamically adjusted to maximize the detection probability while keeping the false-alarm probability under a certain acceptable level. These parameters can be adjusted automatically by the controller. Again, when the number of range bins is large and the PRF is high, the object detector may need to make many thousands to millions of detection decisions per second.

The data processing subsystem is used to track targets of interest, interact with human operators, generate commands to control the radar, and adjust parameters of the other subsystems.

The controller module is the kernel of the radar system; it is not only responsible for synchronizing the behavior of the whole system to meet various timing constraints, but is also critical in tuning the performance of each component by adjusting certain parameters. For instance, it can alter the signal processing parameters such as the detection threshold and transform types, as well as the parameters for the statistical models used by the object detector. In addition, it can regulate the behavior of the digital waveform generator by changing the pulse parameters (such as the pulse width, pulse frequency, PRF, etc.).
The role of the plot manager is to manage real-time updates (also called plots) received from the object detector (say, once every 5 s) of the signal processing subsystem. The monitor module can be connected to a decision-support system so that human operators can make sense of the current situation and make real-time decisions.

The role of the tracker module is to process plots and to determine which plot belongs to which target, while rejecting any false alarms. Tracking is a computationally intensive activity, which typically has several steps:

• Track prediction. The tracker keeps a record (called a track) for each active target of interest. Each track has a unique ID, a state (including position, acceleration, speed, and heading), and possibly a target motion model (which is constructed to fit the motion history of the target). In this step, for each track the tracker needs to employ the associated motion model to predict its new state. It is particularly difficult when the target movement is very unpredictable or when the density of targets is high or the number of false returns is large. In such a situation, the tracker can employ a multiple-hypothesis approach where a track can be branched into many possible directions, with the most unlikely potential branches removed over time to reduce computational cost.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• Track gating. The gating process tentatively assigns a plot to an established track if the plot is within a threshold distance away from the predicted position of the track.

• Track association. In practice, it is very likely that a plot is assigned to more than one established track or more than one plot is assigned to one track. Such an ambiguous assignment could be exploited by a multiple-hypothesis tracker as mentioned above. However, when this feature is not desired, the ambiguity has to be resolved. Algorithms such as the nearest-neighbor approach can be effective in many cases to form one-to-one relationships between plots and tracks. The tracker will then invoke some smoothing algorithm (such as the Kalman filter) to combine a track with the associated plot to produce an improved estimate of the target state as well as a revision to the target motion model.

• Track initiation. After track association, a number of plots may remain unassociated with existing tracks. The tracker will create a new track for each of the unassociated plots. A new track is typically given the status of "tentative" until plots from subsequent radar updates have been successfully associated with the new track. Before being reinforced and confirmed by subsequent radar updates, tentative tracks are transparent to the operator so that false alarms can be washed away from the screen display.

• Track maintenance. Some existing tracks may be missing updates for a while (say, for the past five consecutive radar update opportunities). There is a chance that the target may no longer be there. Depending on the nature of the applications, such a track can be terminated—say, if the target was not seen for the past five out of the 10 most recent update opportunities.

Being a real-time system, a radar system typically operates with many timing constraints at different levels. For example, the object detector may produce batches of radar updates every 5 s, and it is thus necessary for the tracker to process all those radar updates within 5 s. The cluster of DSPs may need to complete its processing at the millisecond level per round, while the receiver may need to get its job done at the microsecond level per echo signal.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

1.1 What is an embedded system? Identify some embedded systems used in your everyday life.

1.2 Are the laboratory computers embedded systems? Why or why not?

1.3 Is Google Glass an example of embedded systems? Why or why not?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Embedded System?	
	2 Why are Real-Time Systems?	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is a combination of computer hardware and software, and sometimes mechanical components as well.
_____2. These are systems that need to respond to a service request within a certain amount of time.
_____3. It offers best-effort services; its service of a request is almost always completed within a known finite time.
_____4. It generally consists of an antenna, an antenna feed, a duplexer, and some preselector filters.
_____5. It is the interface with the medium of radio wave propagation, and it is the first stage during signal reception and the last stage during signal transmission.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FAQs

1. How does embedded system works?

When an embedded system is powered on, its microprocessor reads software instructions stored in memory, executes the instructions to process input information from peripheral components (through sensors, signals, buttons, etc.), and produces output to meet the needs of the external embedding system.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #2

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

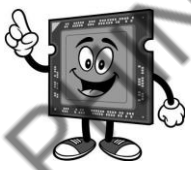
Lesson title: Cross-Platform Development Process (Part 1)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand what is the process of Cross-Platform Development**
2. **Understand Hardware Architecture;**
3. **Understand Software Development.**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
Focus on being "productive" instead of "busy". So, let's fuel up your mind and aim for that finish line!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about what an embedded system is all about. importance of it. Now, let's dig deeper into discussion as we will tackle the Cross-Platform Development Process (Part 1) which covers the Hardware Architecture and Software Development.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Cross-Platform Development?	
	2 What is Hardware Architecture?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

2.1 Cross-Platform Development Process

Compared with non-embedded software, the development of embedded software renders many unique challenges to software engineers:

• For embedded systems, especially for real-time embedded systems, timing correctness is equally important as functional correctness. Sometimes, a timing constraint is so important that it becomes an integral part of the functional requirements. For example, although a GPS navigation system can always identify waypoints correctly (functional correctness), it is useless if it is not able to report the waypoints before it is too late for the user to take actions.

• Most embedded systems are “dumb” devices that cannot run a debugger; this makes it hard to detect and clear program defects. For example, the processing system embedded inside a refrigerator can handle inputs from its touchpad and door sensor, provide output to a digital display, and control the cooling and icemaker machinery. It, however, may not have “luxurious” resources reserved for debugging.

• Most embedded systems are required to offer high reliability. For example, if a system has a reliability requirement of four nines (i.e., 99.99% availability), it is not tolerable if the downtime is greater than 9 s per day. High reliability is not a trivial objective especially for an embedded system that may be operating in a hostile or an unexpected environment. Unpredictable event patterns from the environment may significantly change an embedded system’s sequence of execution.

• Efficient utilization of the memory space is another challenge. More memory means more monetary cost. Making software is a creative activity; making software that can fit into the available memory space demands even more creativity.

• Power management is critical to prolong the operating time of an embedded system. Being able to switch to a low-power state when inactive is a must-have feature for many embedded systems.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Owing to the uniqueness of embedded software, we need to distinguish two terms: host platform and target platform. The term "host platform" refers to the computing environment (i.e., the processor architecture and, if applicable, the operating system) upon which software is developed and its executable artifact is built. In contrast, the term "target platform" refers to the computing environment upon which software (actually its executable artifact) is intended to run.

For most software systems running on general-purpose computers, the host platform is the same as the target platform, and it is not necessary to distinguish the two. However, for embedded software, its target platform is typically different from its host platform.

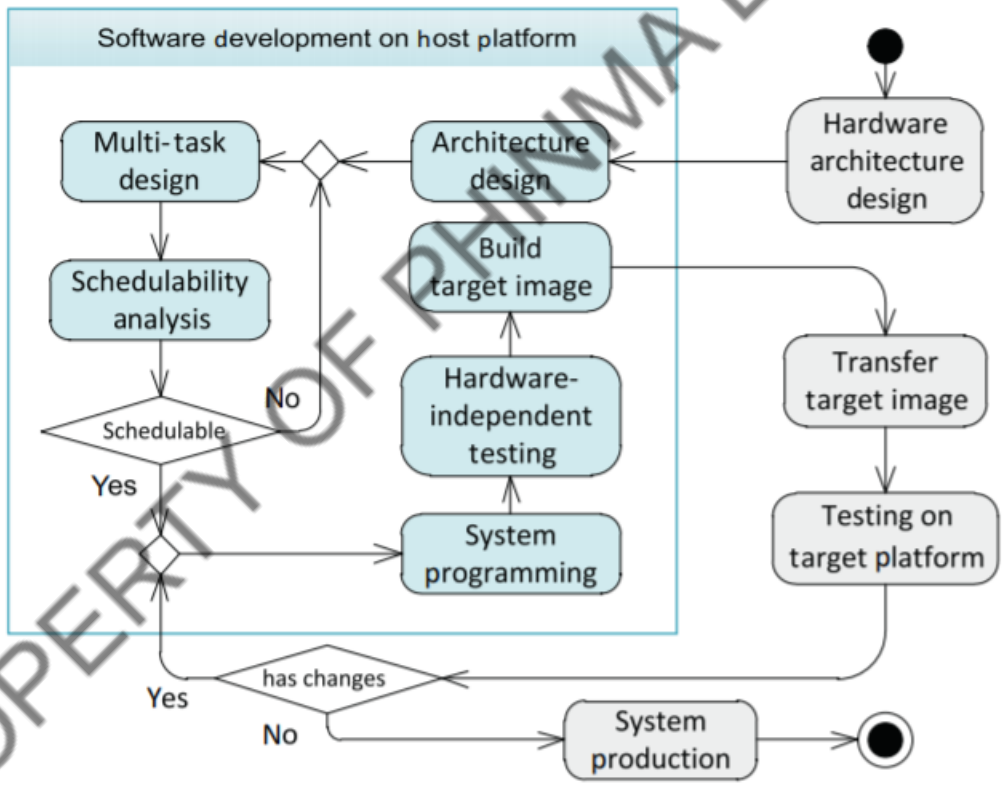


Figure 2.1
Real-time embedded systems development process.

As a road map, Figure 2.1 shows the cross-development process for real-time embedded systems. At a high level, some activities need to be conducted on the host platform, while others need to be performed on the target platform. We next explain each of those activities.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2.2 Hardware Architecture

As far as embedded systems are concerned, hardware architecture is an abstract representation of an electronic or an electromechanical device that contains microprocessors, memory chips, and some peripherals (e.g., clocks, controllers, I/O devices, sensors, and actuators) as needed to fulfill its expected functionality.

Some common design factors related to hardware architecture design are as follows:

• Processing power. How fast a microprocessor will be required? A closely related factor is the processor's power consumption, typically expressed in terms of millions of instructions per second per milliwatt. A more powerful processor generally means higher power consumption, and a higher price too.

• Memory requirements. What types of random-access memory (RAM) and nonvolatile memory (NVM) will be needed? For each type, how much memory will be appropriate on the evaluation board, and how much will be appropriate on the target system? A design with more memory on the evaluation board than what might be needed can often save a project that could have failed otherwise. However, more than necessary will certainly increase the production cost.

• Peripherals. What peripherals are required? Some advanced processor designs, also called microcontrollers or system-on-chip processors, have many built-in peripherals. Such a processor, if chosen, may be able to meet the needs for peripherals already. In general, it is advised to design a debugging interface (such as a serial port) on the evaluation board or the target system. When choices need to be made, performance always has higher priority than cost. A proven statement says that never use a $1.00 chip and expect the performance of a $10.00 chip.

• Reliability. Should the system be fail-proof? May it fail occasionally? What is the tolerable downtime?

• Future upgrade. How will field upgrades be performed?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2.3 Software Development

As shown in Figure 2.1, software development activities are performed on the host platform.

2.3.1 Software Design

Software design includes software architecture design, task design, and schedulability analysis:

• Software architecture design. How are different functional modules interrelated and synchronized?

• Task design. How many tasks should be considered? How should one assign priorities to those tasks? How should one document the task design and the timing constraints associated with the tasks?

• Schedulability analysis. As far as the identified timing constraints are concerned, is the set of tasks schedulable?

2.3.2 System Programming Language C/C++

A wide range of programming languages are in use today. However, for implementing real-time, resource-constrained embedded systems, the language chosen is required to have language-level direct access to low-level hardware. Moreover, different processor types accept different formats of binary code (machine instructions). The source code written in the chosen programming language is not executable until it is transformed into binary code by a computer utility called a compiler. This raises another requirement: a compiler has to be available for the chosen programming language such that the source code can be transformed into efficient binary code that is understandable to the processor selected in hardware design.

In practice, C and C++ are the de facto programming languages for embedded systems. It is estimated that among all the embedded systems, about half of them are implemented in C, and about one third are implemented in C++. Their popularity is partially because C/C++ compilers are available for almost every processor type on the market.

The syntax of C/C++ programming is well beyond the scope of this book. In the rest of this section, we briefly cover a few concepts (semantics) that are relevant to binary code generation.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

```
//-----start of a.h--------
extern int a;
static int sa=5;
//-----end of a.h----------

//-----start of b.h--------
int b(int p);
static int sb=10;
//-----end of b.h----------

//-----start of c.h--------
extern int c;
static int sc=3;
//-----end of c.h----------

//-----start of one.c------
#include "a.h"
#include "b.h"
int c = 2;
int main(void)
{   int temp = 10;
  sb = sb + temp;
  printf("Invoke from %d: %d\n", a, sb);
  b(temp+sb);
  sb = sb + temp;
  printf("Invoke from %d: %d\n", a, sb);
  b(temp+sb);
  return 0;
}
```

```
//-----end of one.c----------

//-----start of two.c--------
#include "c.h"
#include "b.h"
int a = 1;
int b(int p2)
{   static int temp = 2;
  sc++;
  sb = sb + 2;
  temp = p2+temp+sc+sb;
  if (temp < 50) goto done;
  temp = 100;
done:
  printf("Output from %d: %d\n", c, temp);
  return temp;
}
//-----end of two.c----------
```

Listing 2.1- An example C code.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

A C/C++ program (application) can consist of numerous source files, each of which usually contains some #include directives that refer to header files. A compiler processes one source file at a time; it merges those headers with the source file to produce a transitory source file, which is called a translation unit or a compilation unit.

In Listing 2.1, there are two source files and three header files. The source file one.c together with a.h and b.h forms one translation unit, while the source file two.c together with b.h and c.h forms another translation unit.

2.3.2.1 Declarations and definitions

Each source or header file can define or declare many names, such as names for variables, names for functions, statement labels, tags for structures/unions/enumerations, identifiers for constants, and names for namespaces and classes in C++. The following discussion focuses on variable names and function names only.

We distinguish name declarations from name definitions. A name declaration simply states that the name belongs to the current translation unit. More than a name declaration, a name definition also imposes a requirement for storage.

• A variable declaration is a statement specifying a type for a variable name. A variable declaration is a variable definition if it contains no “extern” keyword, or it contains both an “extern” keyword and an initializer. For example, the variable a is defined in two.c and declared in a.h; similarly the variable c is defined in one.c and declared in c.h. Note that the variable temp is defined, independently, in both functions main() and b(). The statement “extern intk=0” is a definition of variable k because it has an initializer.

• A function declaration is a statement containing a function prototype (function name, return type, the types of parameters and their order). A function declaration is a function definition if the function prototype is also followed by a brace-enclosed body, which generates storage in the code space. For example, the function b() in b.h is a function declaration, while the function b() in two.c is a function definition.

A name declared in C/C++ may have attributes. For example, a variable name has a type, a scope, a storage duration, and a linkage; a function name has all those attributes except storage duration.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The type of a variable determines its size and memory address alignment, the values it can take, and the operations that can be performed on it. A function's type specifies the function's parameter list and return type.

We next examine scope, storage duration, and linkage in detail.

2.3.2.2 Scope regions

A name's scope is that portion of a translation unit in which the name is visible. A name in an inner scope can hide a name from an outer scope. C and C++ each support five different kinds of scope regions:

• File/namespace scope. In C, a name has file scope if it is declared in the outermost level of a translation unit. In C++, a name has namespace scope if it has file scope (global scope) or is declared in a namespace definition. In Listing 2.1, names with file scope include a, b, c, sa, sb, sc, and main.

• Function scope. A statement label has function scope. A label can be defined only in the body of a function definition and is in scope everywhere in that body. In Listing 2.1, the label done has a function scope.

• Function prototype scope. A name has function prototype scope if it is declared in the function parameter list of a function declaration without a body. In Listing 2.1, the parameter p of function b() in b.h has function prototype scope.

• Block scope. A name has block scope (called local scope in C++) if it is declared within a function definition or a block nested therein. In Listing 2.1, for the function b() defined in two.c, the parameter p2 and the variable temp have block scope.

• Class scope. A name in C++ has class scope if it is declared within the body of a class (including structure and union) definition. In C, there is no corresponding notion of class scope: each structure or union has a separate namespace for its members.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The storage duration of a variable determines the lifetime of the storage for that variable. Each variable in C/C++ has one of the following three storage durations:

• Static. For a variable with a static storage duration, its storage size and address are determined at compile time (before the program starts running); the lifetime of its storage is the entire program execution time. A variable declared at file/namespace scope has a static storage duration.

• Automatic. A local variable declared at block scope normally has an automatic storage duration. Local variables are stored in a run-time stack. Allocating storage for local variables usually takes just one machine instruction. Each time a function is called, a stack frame (a block of memory in the stack) is allocated for the function's local variables, and the stack frame is deallocated when the function returns. Thus, for a variable with an automatic storage duration, the lifetime of its storage begins upon entry into the block immediately enclosing the object's declaration and ends upon exit from the block.

• Dynamic. A local variable declared at block scope can have a dynamic storage duration if its storage is allocated by calling an allocation function, such as malloc() in C or the operator new in C++. Dynamic memory allocation allows a user to manage memory very economically. The drawback is that it is much slower than automatic allocation because it typically involves tens or hundreds of instructions. For a variable with a dynamic storage duration, the lifetime of its storage lasts until the memory is deallocated explicitly—say, by a free function in C or the operator delete in C++.

2.3.2.4 Linkage

Linkage determines whether name declarations in different scopes can refer to the same name definition. The linkage attribute applies to both variable names and function names.
C and C++ support three levels of linkage:

• No linkage. A name defined in block scope or structure/class scope normally has no linkage. In such a case, it can be referenced by name only within its scope. Outside its scope, declarations of the same name refer to different entities. Function parameters and local variables normally have no linkage.

• Internal linkage. A name defined in file/namespace scope can have internal linkage. In such a case, it can be referenced by name only within the same translation unit. Outside

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

the translation unit, declarations of the same name refer to different entities.

• External linkage. A name defined in file/namespace scope or class scope can have external linkage. In such a case, it can be referenced by name within the same translation unit, and can be referenced from other translation units. Declarations of the same name always refer to the same entity. Global variables and functions normally have external linkage by default.

2.3.2.5 Storage-class specifiers

A variable declaration in C/C++ can be preceded by a storage-class specifier, which is used to change the way of creating the memory storage for the variable. Both storage duration and linkage can be affected by the use of a storage-class specifier.

A storage-class specifier can be one of the following keywords:

• Auto. This specifier can be used only in declarations of local variables with block scope. All local variables in C/C++ are of the "auto" type by default, so the keyword "auto" is very rarely used explicitly. The "auto" specifier indicates a variable with an automatic storage duration. If it is uninitialized, such a variable has a garbage value.

• Register. This specifier can be used only in declarations of variables with block scope. The "register" specifier basically requests the compiler to store the variable in a register; this allows faster access than an "auto" variable, which is stored in the main memory. You are not allowed to take the address of a register variable. "Register" is the only storage-class specifier that can be used for function parameters.

• Extern. This specifier can be used in declarations of both variables and functions.

• A variable with the "extern" specifier has external linkage, which means that it can be referenced from other translation units. This also avoids unnecessary passing of variables as arguments during function calls.

• A variable with the "extern" specifier has a static storage duration, which means that its memory is allocated at compile time and it exists and retains its value as long as the program runs. If it is uninitialized, such a variable is set to 0.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• A local variable (block scope) normally has an automatic storage duration and no linkage. With the “extern” specifier, a local variable will have a static storage duration and external linkage.

• With or without the “extern” specifier, a global variable (file/namespace scope) by default has a static storage duration and external linkage.

• With or without the “extern” specifier, a function by default has external linkage, which means that it can be called from other translation units.

• Static. This specifier can be used in declarations of both variables and functions.

• A variable with the “static” specifier has a static storage duration, which means that its memory is allocated at compile time and it exists and retains its value as long as the program runs. If it is uninitialized, such a variable is set to 0.

• A local variable (block scope) normally has an automatic storage duration. With the “static” specifier, a local variable will have a static storage duration. As a consequence, its value persists between different function calls.2 This will not affect its linkage (no linkage) and scope attributes.

• With the “static” specifier, a global variable (file/namespace scope) has its linkage attribute changed to “internal linkage.” For example, in Listing 2.1, the global variable sb defined in b.h has a “static” specifier. The header b.h is included by both one.c and two.c. Although each of the two translation units uses the same name sb, it actually refers to a distinct entity in each translation unit.

• A function normally has external linkage. With the “static” specifier, a function will have internal linkage, which means that it may be called only within the translation unit in which it is defined.

By default, a local variable without a specifier is treated the same as one with “auto,” and a global variable without a specifier is treated the same as one with “extern”.

Table 2.1 summarizes how a variable’s linkage and storage duration may be affected by a storage class specifier. Note that the only storage class specifiers allowed in a file/namespace scope declaration are “static” and “extern.”

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Table 2.1 A variable's linkage and storage duration depend on the storage class specifier

Storage Class Specifier	Block Scope		File/Namespace Scope		Structure Member/Class Scope	
	Linkage	Storage Duration	Linkage	Storage Duration	Linkage	Storage Duration
None	No linkage	Automatic	Normally external linkage	Static	No linkage	Same as enclosing object
Auto	No linkage	Automatic	NA	NA	NA	NA
Register	No linkage	Automatic	NA	NA	NA	NA
Extern	Normally external linkage	Static	Normally external linkage	Static	NA	NA
Static	No linkage	Static	Internal linkage	Static	External linkage in C++ only	Static in C++ only

NA, not applicable.

When the modifier "static" specifies a variable or a function declaration, it makes the variable (function) local to the unit in which it is declared; in other words, the variable (function) cannot be referenced outside that unit. For instance, a static global variable (function) is limited to the file in which it is declared, and a static local variable inside a function is limited to that function. The lifetime of a static variable, global or local, is until the program terminates. This explains why a static variable can retain its value between function calls.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2.3.3 Test Hardware-Independent Modules

In practice, most modules of an embedded software are hardware independent. Hence, the skills for testing "general-purpose" software systems are equally applicable to the testing of those hardware-independent modules. For instance, test stubs, as used in the top-down testing approach, can be used to simulate the functionality of the software components that directly interact with hardware devices. As a simulator, a test stub of a module implements the interface of the actual module but simply provides canned responses to calls made during testing.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

2.1 What makes it challenging to develop an embedded system?
2.2 How many scope regions are defined in C/C++?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Cross-Platform Development?	
	2 What is Hardware Architecture?	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is an abstract representation of an electronic or an electromechanical device that contains microprocessors, memory chips, and some peripherals (e.g., clocks, controllers, I/O devices, sensors, and actuators) as needed to fulfill its expected functionality.

_____2. This specifier can be used in declarations of both variables and functions.
_____3. This specifier can be used only in declarations of variables with block scope.
_____4. This specifier can be used only in declarations of local variables with block scope.
_____5. , It can be referenced by name within the same translation unit, and can be referenced from other translation units.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FAQs

1. What is the common programming language used in embedded systems?

In practice, C and C++ are the de facto programming languages for embedded systems. It is estimated that among all the embedded systems, about half of them are implemented in C, and about one third are implemented in C++. Their popularity is partially because C/C++ compilers are available for almost every processor type on the market.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #3

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Lesson title: Cross-Platform Development Process (Part 2)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand what are Build Target Images;**
2. **Understand how Transfer Executable File Object to Target works;**
3. **Understand how Integrated Testing on Target works;**
4. **Understand what a system production is.**

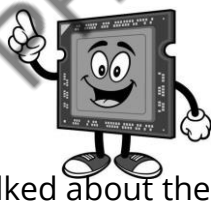
Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"If you spend too much time thinking about a thing, you'll never get it done." So, get up and let those things get done!

A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)



In the previous lesson, we talked about the first part of Cross-Platform Development Process. Now, let's dig deeper into discussion as we will tackle the Cross-Platform Development Process (Part 2) which includes Build Target Images, Transfer Executable File Object to Target, Integrated Testing on Target and System Production.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Build Target Images?	
	2 How Integrated Testing on Target works?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

2.4 Build Target Images

In this section, we will look at the toolchain for building target images, and introduce a standard object file format—ELF. As a case study, we will also examine the image building process for embedded systems that rely on the QNX operating system.

2.4.1 Cross-Development Toolchain

Figure 2.2 shows the code transformation process involving a chain of cross-development tools: cross compiler/assembler, linker, and dynamic linker.

2.4.1.1 Cross compiler/assembler

A compiler is called a native compiler if its output is intended to directly run on the host platform (or the same type of environment) where the compiler runs. A cross compiler is a compiler capable of generating executable code for a target platform that is different from the host platform. This statement applies to a cross assembler as well, except that it processes source files written in assembly languages.

A cross compiler/assembler is useful in a number of situations:

• The target platform (e.g., an embedded system) has extremely limited resources. It is not powerful enough to run a native compiler to generate executable code by itself.
• A source project is intended to run on different target platforms—say, different processor types, different versions of an operating system, or even different operating systems. By use of a cross compiler, a single host platform (development environment) can be set up to compile for each of these targets.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

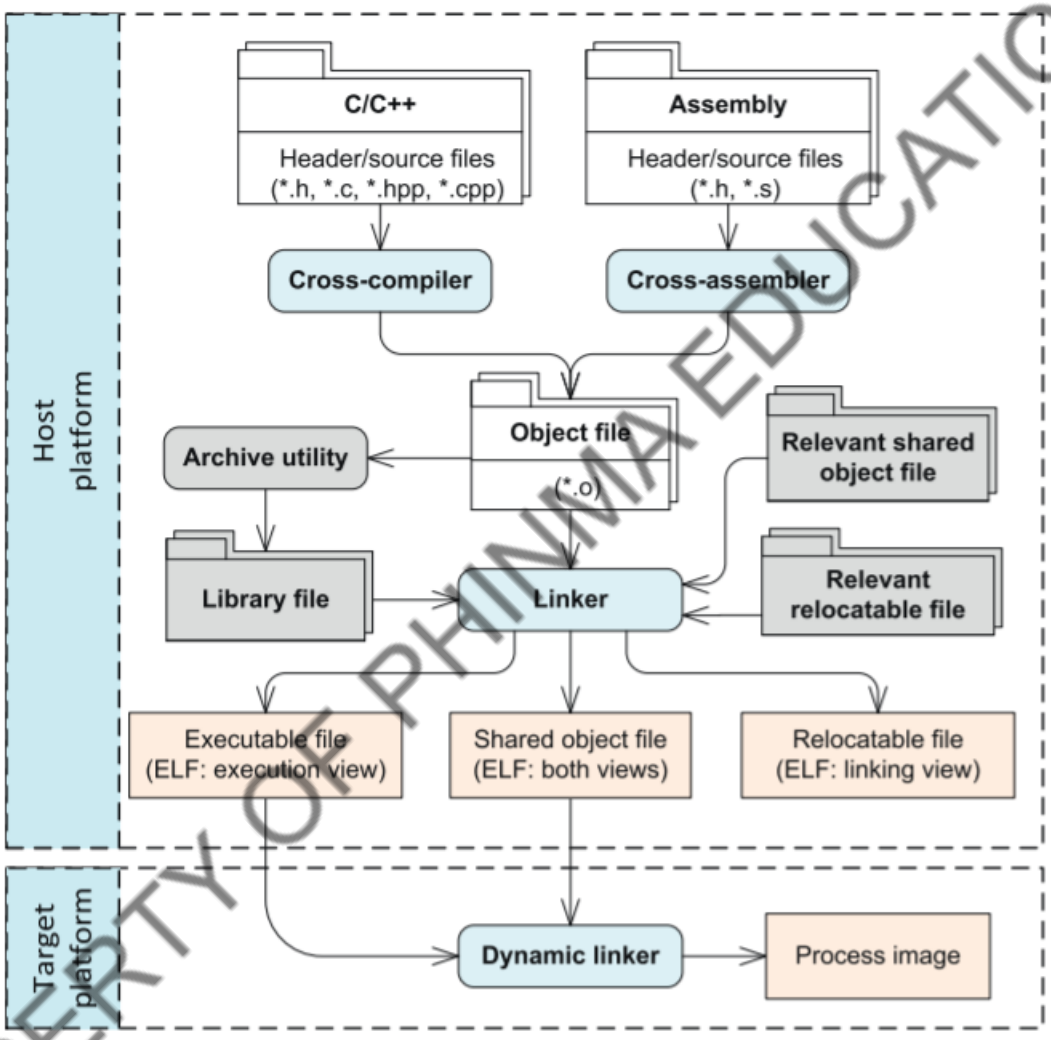


Figure 2.2
Cross-development toolchain.

For each compilation unit, the compiler/assembler generates an object file.3 For instance, the C compiler can produce two object files—one.o and two.o—for the example code given in Listing 2.1.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Each object file contains a symbol table. A symbol table is an array-like data structure consisting of entries about the global symbols (i.e., names of global variables and nonstatic functions) defined in the compilation unit, as well as the external symbols (with "external linkage") referenced in the compilation unit. When the compiler encounters a symbol declaration, it stores that symbol and its attributes in the symbol table of the object file.

For the example code given in Listing 2.1, we have the following:

• In one.c, there are four global symbol definitions: global variables c, sa, and sb, and a function main(). It references two external symbols a and b().

• In two.c, there are four global symbol definitions: global variables a, sb, and sc, and a function b(). It references the external symbol c. The symbol table also has an entry for temp, which is a local variable with a static storage duration.

An object file's symbol table holds information that is needed by a compiler/linker to locate and relocate a program's symbolic definitions and references.

2.4.1.2 Linker

A complete program (application) is typically composed of multiple object files, each of which may cross-reference the definitions for data or functions defined in the other object files. The process of combining multiple object files into a single object file is called static linking, which is performed by a computer program known as a linker or link editor. The compiler automatically invokes the linker as the last step of compiling.

A linker has two major jobs:

1. Symbol resolution. While a linker is processing the application-specific object files, it needs to analyze each object file and determine where symbols with external linkage are defined. External symbol references may also involve application-independent object files—those that define commonly used functions (say, printf) for a wide range of applications (some known as archived library files, some known as .so, or sharable object files). It is worth noting that two variables of the same name can be defined in different scopes. In Listing 2.1, a static variable sb is defined in both one.c and two.c. This will not confuse the linker because the compiler has treated them as distinct symbols. Essentially the compiler uses "namespace" to distinguish variables: a local variable name is tagged by

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

the function to which it belongs and a global variable name is tagged by the file name.

2. Symbol relocation. The final object file produced by a linker contains all the symbol definitions originating from the input object files, as well as those from static library files. As the linker merges the input object files and inserts code from the library files, the symbol offsets are changed. By symbol relocation, the linker, via a relocation table, modifies the binary code of the final object file so that each symbol reference reflects the actual address assigned to that symbol.

A linker can generate three types of object files: relocatable file, executable file, and shared object file:

• Relocatable file. A relocatable file holds code and data suitable for linking with other object files to create an executable file or a shared object file.

• Executable file. An executable file holds code and data suitable for execution. In an operating system, the basic unit of execution is called a process or process image, which is dynamically created from an executable file (say, as a result of an exec or spawn call). For systems supporting virtual memory, each process is put into its own address space.

• Shared object file. A shared object file holds code and data suitable for further linking. It can be combined with other relocatable and shared object files to create another object file. Some shared object files are dynamically linkable, and are intended to be loaded at run time and can be simultaneously shared by multiple process images.

2.4.1.3 Dynamic linker

In order to create a process image for an object file, the object file has to be loaded from its storage location into RAM. This job is performed by a program interpreter, which
• itself is an executable file or a shared object file;
• is self-interpreted (it does not need another program interpreter);
• is able to receive control from the system and provide a running environment for the object being loaded.

A program loader is a program interpreter that simply loads a program into memory and transfers the execution control to it. This approach works for stand-alone executable files. An executable file is stand-alone when it contains one copy for each library routine used in the

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

program. This makes the executable object easier to distribute to diverse target environments, at the cost of larger memory space.

Another type of program, instead of having an embedded copy for each library routine, contains only reference information of sharable library routines. Obviously, such a program requires the presence of library files on the target system. It also demands a dynamic linker—a program interpreter that is more powerful than a simple program loader. A dynamic linker, once it has gained control from the system, will first load the whole program into memory to form an initial process image, then resolve symbolic references dynamically by loading and binding external shared libraries to form a complete process image, and finally transfer control to the process. This procedure is also called dynamic linking.

An advantage of dynamic linking is that multiple applications can share a single copy of a library. This also implies that a deployed system can automatically benefit from bug fixes and upgrades to libraries.

2.4.2 Executable and Linking Format

When tools from different vendors are used in the development process, some standard format ought to be adopted in the linking and dynamic linking process. ELF [2, 14] is a widely adopted object file format. It supports cross-compilation, initializer/finalizer (e.g., the constructor and destructor in C++), dynamic linking, and other advanced system features. An ELF object file has an ELF header, a section header table and/or a program header table, and a series of sections or segments. As shown in Figure 2.3, the ELF header of an object file resides at the beginning; it serves as a "road map" for the rest of the object file. An ELF header has the following fields:

• The first four bytes mark the file as an ELF object file. The "class" byte can take a value of 1 or 2, indicating the support for 32-bit or 64-bit processor architectures, respectively. The "data" byte can take a value of 1 or 2, specifying ELFDATA2LSB or ELFDATA2MSB as the data-encoding format. The encoding ELFDATA2LSB specifies 2's complement values,6 with the least significant byte occupying the lowest address. The encoding ELFDATA2MSB specifies 2's complement values, with the most significant byte occupying the lowest address. The "pad" bytes are unused and are reserved for future changes.
• The "type" field indicates the object file type, which can take a value of 1, 2, or 3, representing relocatable file, executable file, or shared object file, respectively.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• The “machine” field specifies the required architecture for this object file. For example, the value 3 is reserved for Intel 80386 processors, and 40 is reserved for ARM processors.
• The “version” field identifies the version of the object file format, which is fixed to 1 for the current standard.
• The “entry” field gives the memory address of an entry point to which the system first transfers control. A system may have many entry points—say, for reset, interrupts, and software interrupts. However, an executable file can have one and only one entry point.
• The “phoff” field holds the byte offset of the program header table in the object file. It is zero if the file has no program header table.
• The “shoff” field holds the byte offset of the section header table in the object file. It is zero if the file has no section header table.
• The “flags” field holds processor-specific flags associated with the file. It is zero for the 32-bit Intel architecture because it defines no flags.
• The “ehsize” field holds the ELF header’s size in bytes.
• The “phentsize” field holds the size in bytes of one entry in the file’s program header table; all entries are of the same size.
• The “phnum” field holds the number of entries in the program header table.
• The “shentsize” field holds the size in bytes of one entry in the file’s section header table; all entries (section headers) are of the same size.
• The “shnum” field holds the number of entries in the section header table.
• The “shstrndx” field holds an index in the section header table, giving the entry associated with the section-header string table

The ELF standard provides two parallel views of an object file’s contents. On the one hand, an ELF object file can be viewed as a series of named sections. This “linking” view is taken by compilers/assemblers and linkers. Sections are intended for further processing by a linker. On the other hand, an ELF object file can be viewed as a series of named segments. Segments are intended to be mapped into memory to create a process image. This “execution” view is taken by dynamic linkers or program loaders.

2.4.2.1 Linking view

Figure 2.3 shows the linking view of ELF, where it is optional to have a program header table. For example, there is no program header table in a relocatable file, whereas a shared object file may or may not have a program header table.

Table 2.2 lists some default sections with predefined names. Each section contains a single

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

type of information, such as program code (instructions), data, symbol, or relocation information. Typically, an object file may have

• a .text section for executable instructions;
• a .data section for initialized data;
• a .bss (for "block started by symbols") section for uninitialized data;
• a .symtab section (symbol table) for static linking;
• a .strtab section holding a symbol string table;
• a .shstrtab section holding a section-header string table; and
• a few .rel[name] sections each containing the relocation information for a specific section with the name [name].

Each section in an object file occupies one contiguous (possibly empty) sequence of bytes, and has exactly one section header describing it. A section header has the following fields:

• The "name" field specifies the name of the section. Its value is an index in the .shstrtab section (section-header string table), giving the location of a null-terminated string.
• The "type" field categorizes the section's contents. The first column of Table 2.2 lists some of the supported types.
• The "flags" field supports one-bit flags: 0x1 for WRITE (the section contains writable data), 0x2 for ALLOC (the section occupies memory during process execution), and 0x4 for EXECINSTR (the section contains executable machine instructions).
• The "addr" field, if not zero, gives the load address7; that is, the starting address in the NVM at which the section's first byte resides.
• The "offset" field gives the byte offset from the beginning of the file to the first byte in the section.
• The "size" field gives the section's size in bytes.
• The "link" field, for a section related to dynamic linking or relocation, holds a section header index of a string table or symbol table.
• The "info" field, for a .rel or .rela section, holds the section header index of the section to which the relocation applies.
• The "addralign" field holds a value of 0 or positive integral powers of 2, specifying a constraint for section address alignment. If the value of addralign is greater than 1, the value of addr must be congruent to 0, modulo the value of addralign.
• The "entsize" field, if not zero, gives the size in bytes of an entry of a table section.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

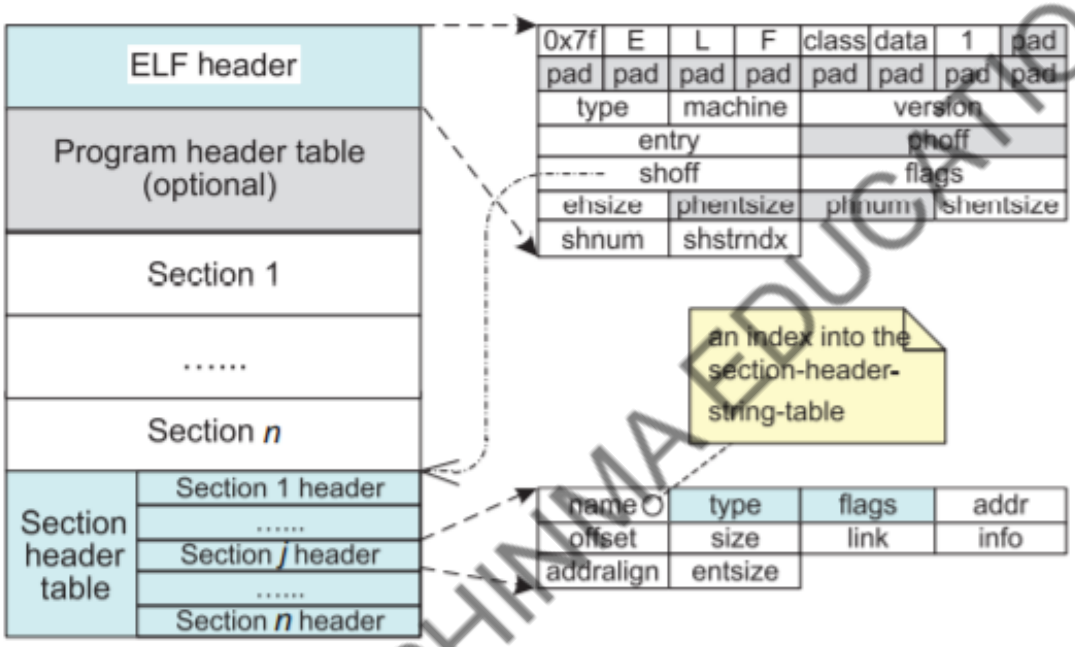


Figure 2.3
The linking view of ELF.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Table 2.2 Some predefined section headers, their types, and their attributes

Type	Name	Meanings	Flags
NOBITS	.bss	Uninitialized data (global, static)	ALLOC + WRITE
PROGBITS (program contents)	.comment	Extra information such as version control	
	.data	Initialized data (global, static)	ALLOC + WRITE
	.data1	More initialized data	ALLOC + WRITE
	.debug	Symbolic debugging information	
	.fini	Process termination code	ALLOC + EXECINSTR
	.got	Global offset table (each entry contains a symbol with an absolute address to be determined in dynamic linking)	
	.init	Process initialization code	ALLOC + EXECINSTR
	.interp	Path name of a program interpreter—say, a dynamic linker (to load the program and to link shared libraries if necessary)	ALLOC + EXECINSTR
	.line	Line number information for symbolic debugging	
	.plt	Procedure linkage table	
	.rodata	Read-only data (constant)	ALLOC
	.rodata1	More read-only data	ALLOC
	.text	Executable instructions	ALLOC + EXECINSTR
DYNAMIC	.dynamic	A table of entries for dynamic linking	ALLOC
DYNSYM	.dynsym	Symbol table for dynamic linking	[ALLOC]
HASH	.hash	Symbol hash table for dynamic linking	[ALLOC]
REL	.rel[name]	A table of relocation entries without explicit addends ([name] can be .text, .data, etc.)	[ALLOC]
RELA	.rela[name]	A table of relocation entries with explicit addends	[ALLOC]
STRTAB	.shstrtab	Section-header string table (section names)	
	.strtab	Symbol string table (for names associated with symbol table entries)	[ALLOC]
SYMTAB	.symtab	Symbol table for static linking	[ALLOC]

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2.4.2.2 Execution view

Figure 2.4 shows the execution view of ELF, where it is optional to have a section header table. In this view, an object file is a set of segments described by a program header table, which is meaningful only for executable and shared object files. As the system creates a process image from an object file, a file segment (such as .text, .data) is logically copied/mapped into virtual memory to form a memory segment.

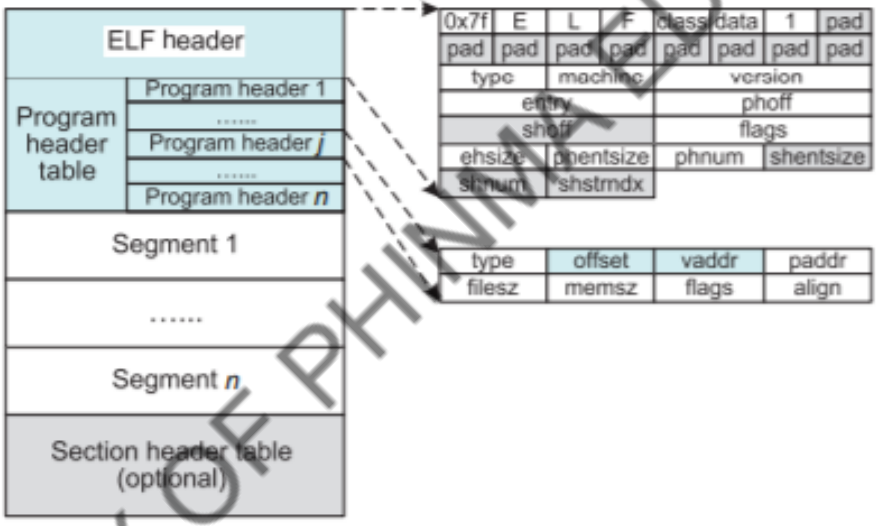


Figure 2.4
The execution view of ELF.

A file segment usually consists of several sections. Each file segment is described by an entry of the program header table, which has the following fields:
• of fset. This gives the offset from the beginning of the file at which the first byte of this file segment resides.
• filesz. This gives the size in bytes of this file segment; it may be zero.
• vaddr. This gives the virtual memory address at which the first byte of the memory segment resides. This is also called the segment's run address, and it should be within the RAM range.
• memsz. This gives the size in bytes of the memory segment; it may be zero.
• paddr. This is reserved for the segment's physical address. When used, this is the starting address in the NVM at which the first byte of this file segment resides. This is also called the segment's load address. For an object capable of in situ execution, its segments can

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

have the same load addresses and run addresses.

• type. This field specifies the segment type, which can be one of the following:
  • NULL. This segment is unused and should be ignored.
  • LOAD. This is a loadable segment. A corresponding memory segment is created in the process image, and the bytes from the file (specified by filesz) are mapped to the beginning of the memory segment. If the specified memory size is larger than the file size (memsz > filesz), the "extra" bytes in the memory segment hold the value 0.
  • DYNAMIC. This is a dynamic segment, specifying information for dynamic linking. This segment starts with a .dynamic section, followed by a few other sections such as a hash table section, symbol table section, a string table section, and a relocation table section.
  • INTERP. This segment is meaningful only for executable files, specifying the location and size of a null-terminated path name to a program interpreter (program loader or dynamic linker). If it is present, it must precede any loadable segment.
  • NOTE. This segment specifies the location and size of auxiliary information.
  • PHDR. This segment specifies the location and size of the program header table itself. It may occur only if the program header table is part of the process image. If it is present, it must precede any loadable segment.
• flags. This gives flags (readable, writable, executable) relevant to the segment.
• align. If align > 1, this gives the value to which the file segment and the corresponding memory segment should be aligned. The value of align should be a positive, integral power of 2, and vaddr should equal of fset, modulo align.

2.4.3 Memory Mapping

By default, especially when an executable file is intended to run on an operating system, each loadable segment of the object file is mapped by the linker to a virtual address location (specified by the vaddr field). For each program to be executed, the operating system typically allocates a protected memory space in RAM, where a process image is formed by loading the program from the file system (via a file descriptor).

An operating system is simply a "big" application. An operating system itself needs to be loaded from somewhere when the system starts. This job is done by a "starter" program, also known as an initial operating system loader. Where is such an operating system loader found when the system is powered on? Moreover, there are many embedded systems running without an operating system. Where should the instructions be loaded when such an embedded system starts? This brings us to memory mapping.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Via a script file, a linker can be directed to map each segment of an executable file to a specific memory location available on the target system. First of all, an operating system loader or an embedded application without an operating system needs to be mapped directly to specific locations on the memory chips of the target system. When the system starts, a booter is able to locate the loader or application image stored on the NVM and, if necessary, copy some segments from the NVM into appropriate RAM locations. The
loader or application gains control afterward and its execution follows.

As an example, Listing 2.2 gives a linker script written for an operating system loader. The TARGET command specifies the format of the input object files. The input format used in the example is 32-bit ELF that is customized to support little-endian ARM processors. The OUTPUT_FORMAT command specifies the format of the output object file. In the example, the output format is the same as the input format.
The ENTRY command specifies a symbol name "_start," which indicates the first executable instruction in an output file (its entry point).

The MEMORY command describes memory regions on the target board, their sizes, and their locations (start addresses of the regions in physical memory). The example has three named memory regions defined: stack, ram, and rom. In particular, the rom block starts at address 0x00300000 and has 16K (0x4000) bytes.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

```
TARGET(elf32-littlearm)
OUTPUT_FORMAT(elf32-littlearm)
ENTRY(_start)

MEMORY
{
  stack : ORIGIN = 0x0030E000,  LENGTH = 0x2000
  ram   : ORIGIN = 0x00304000,  LENGTH = 0x6000
  rom   : ORIGIN = 0x00300000,  LENGTH = 0x4000
}

SECTIONS
{
  .text :
  {
    *(.text)
    *(.rodata*)
    *(.glue_7)
    *(.glue_7t)
  } > rom
  _etext = .;

  .data :
  {
    *(.data)
    *(.sdata)
  } > ram

  .bss :
  {
      *(.bss)
      *(.sbss)
  } > ram
}
```

Listing 2.2
Example linker script for an operating system loader.

The SECTIONS command describes the placement of named output sections, and which input sections go into which output sections. In the example, three output sections are defined: .text, .data, and .bss.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The content of an output section consists of one or more statements of the form filename (section, section, ...). An asterisk can be used to represent all input object files. In the example, the .text section contains four statements:

• *(.text): copy the .text section from each input object file into the .text section of the output object file.
• *(.rodata*): copy all the sections with the prefix .rodata from each input object file into the .text section of the output object file.
• *(.glue_7): copy the .glue_7 section from each input object file into the .text section of the output object file. The glue_7 segment contains glue functions generated by the compiler for the 32-bit ARM mode.
• *(.glue_7t): copy the .glue_7t section from each input object file into the .text section of the output object file. The glue_7t segment contains glue functions for the ARM thumb mode.

The linker does not shuffle sections to fit into the available memory regions. The statement ">region" at the end of a section command assigns the section to a specific memory region (it is reflected in the segment's vaddr field). In the example, the .text section goes to rom, while the .data and .bss sections go to ram.

The statement "_etext = .;" defines a global symbol "_etext" just after the last byte of the .text section.

Note that the .sdata and .data sections both contain initialized data, but the size of a .sdata session is less than a threshold (say, 64 KB), smaller than a .data section (with a size above the threshold). Likewise, a .sbss section is similar to a .bss section, except that it is smaller than a .bss section.

Figure 2.5 illustrates a possible memory mapping, given that the linker follows the script in Listing 2.2 to process the two input object files shown on the left in Figure 2.5.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2.4.4 Case Study: Building a QNX Image

In this section, the term "QNX image" refers to a holistic executable file object containing the user applications and the underlying QNX operating system.

Figure 2.6 shows a typical procedure for building a QNX image for a specific target board.

To develop software for a particular target board, it is critical to start with a board support package (BSP), which is a software component specifically tailored for supporting the hardware design of the target board. A commercial evaluation board typically comes with a sample BSP, which can save developers a great deal of time and effort.

A QNX BSP for a board, if available, comes with an initial program loader for loading the QNX operating system, as well as a startup module that contains all the device drivers for the board.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

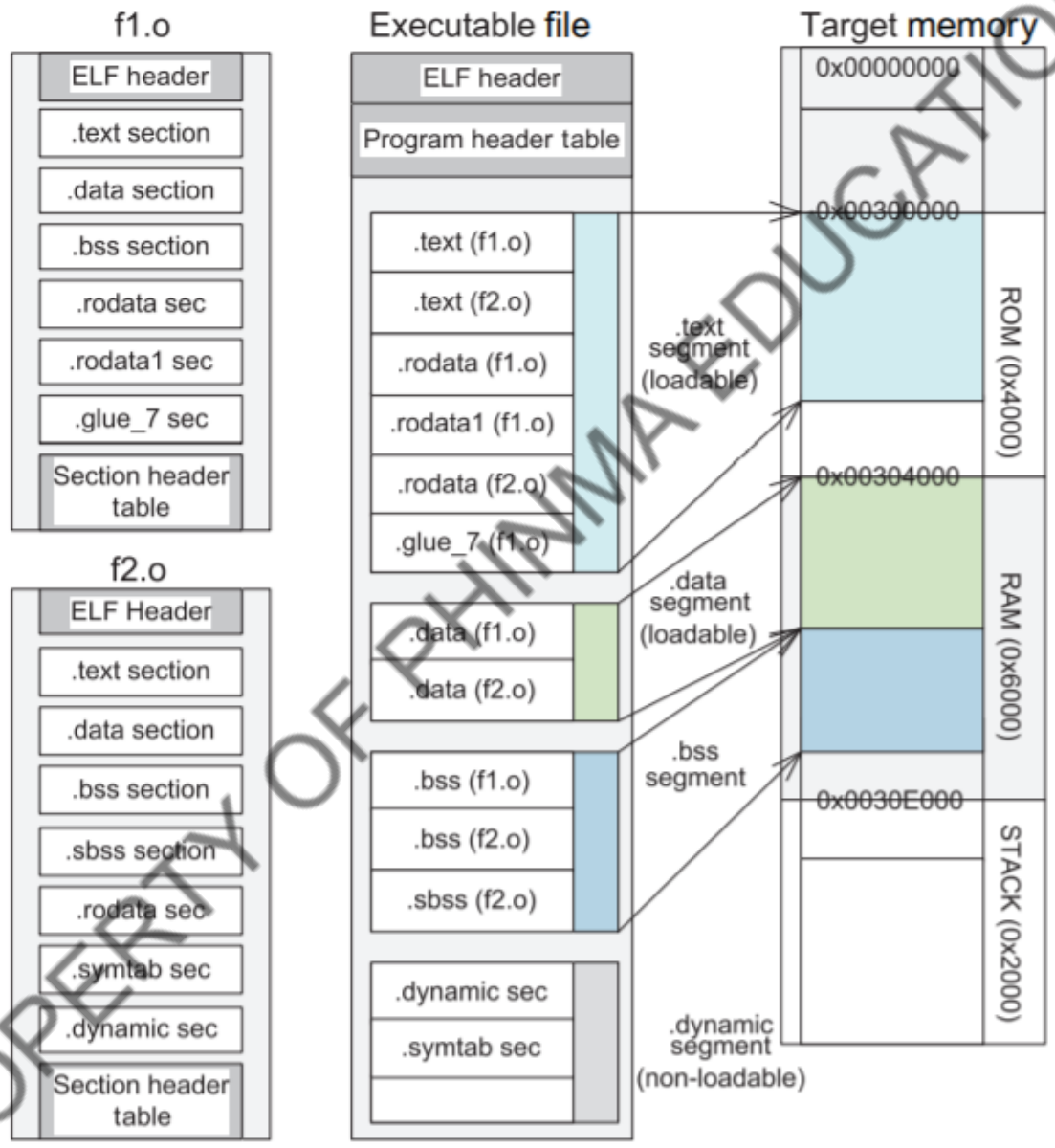


Figure 2.5
Mapping segments into target memory.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

A system builder project is needed to glue all the pieces together [6]. A system builder project can be processed by a utility called mkifs (make image filesystem) to build a bootable QNX image. The behavior of mkifs is governed by a build-script file (.bld), which can be configured such that the QNX image produced contains

- the startup binary generated from the BSP;
- the QNX kernel and relevant shared library objects;
- the application-specific binaries generated from user C/Photon projects; and
- some relevant libraries and utilities.

The final QNX image object for an embedded system has two parts: an initial program loader (generated from the BSP) and the bootable QNX image.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

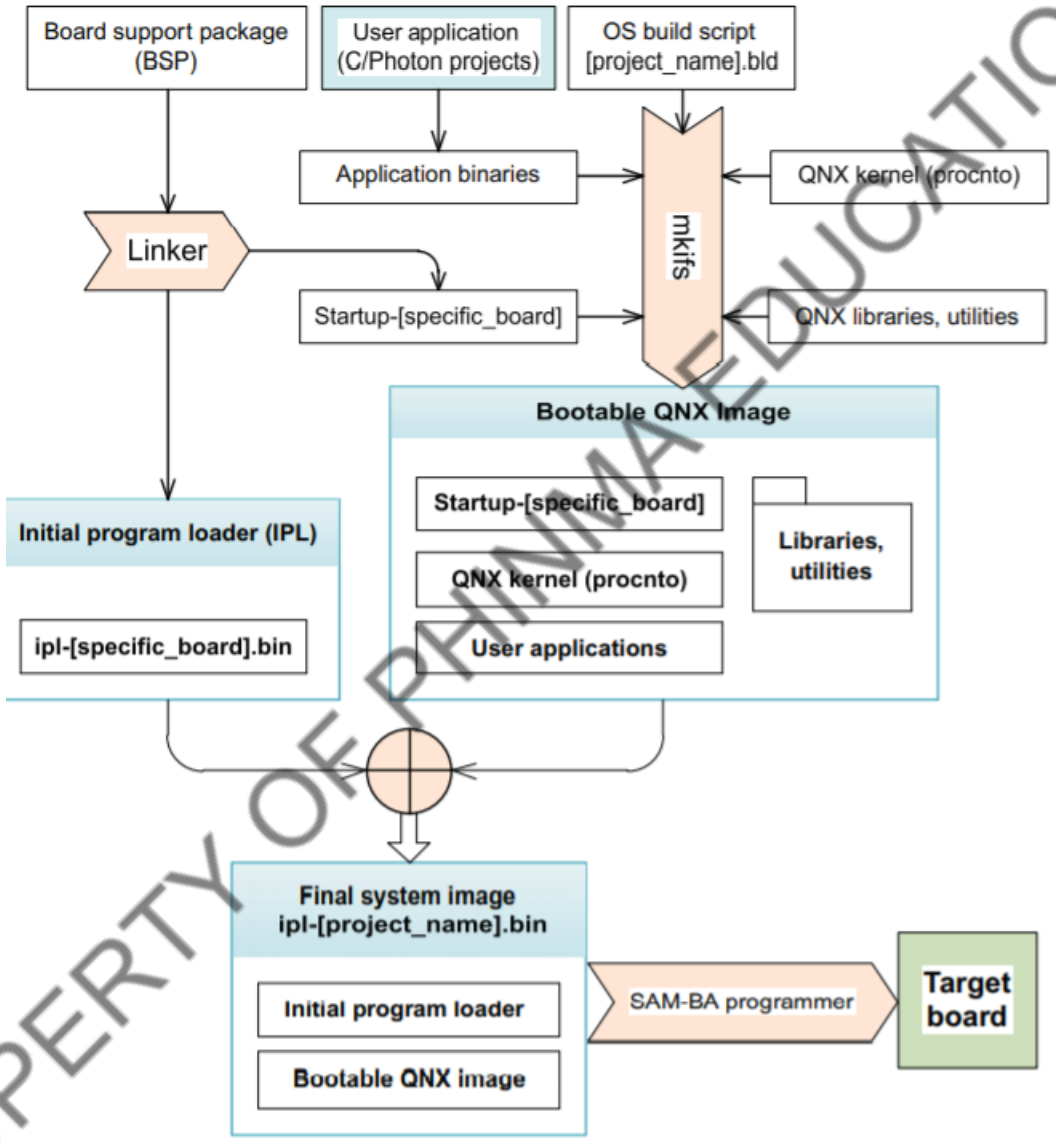


Figure 2.6
QNX operating system image building process.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2.5 Transfer Executable File Object to Target

Normally, a utility known as "programmer" is used to transfer (say, through high-voltage programming) an executable file object to the NVM (ROM or flash) available on the target system. For instance, in Figure 2.6, a programmer called SAM-BA is used to transfer the final QNX image to the target board. Once the executable file object is accessible to the processor of the target board, it is ready to run on the target platform.

2.6 Integrated Testing on Target

The software modules that interact with hardware devices are yet to be tested. Many uncertain factors can make it difficult and challenging to test/debug on the target system. For instance, owing to jitters or some other reasons, it might be extremely hard to reproduce a time-critical input situation. The once useful techniques such as inserting printf() statements8 become futile or even inadvertently interfere with the run-time behavior of the system.

Testing at this step, if necessary, can benefit from devices such as oscilloscopes and in-circuit emulators. The testing activities mainly involve the following aspects:
• Device drivers. They are tightly coupled with the corresponding devices.
• Portability issue. It can be time-consuming if the software has been written to support more than one target platform.
• Shared-resource issues. It is critical and should be thoroughly tested when there are data objects shared between interrupt service routines and user task code.
• Racing conditions. When multiple tasks use mutually exclusive resources, they may race each other to produce nondeterministic results. It is very hard to reproduce and debug a race condition because its occurrence is highly dependent on the relative timing between interfering tasks.
• Timing correctness. This can be tested only on the target system itself.
In order to resolve the problems detected in testing, it might be necessary to change the system design and implementation, and the development process would come back again to the host platform (see Figure 2.1).

2.7 System Production

When the system is ready for production, cost becomes the major concern. While it is advised to use more speed and memory in the development phase, it is now the time to consider reducing the production cost by downgrading the microprocessor and/or memory devices. Of

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

course, the whole system has to be thoroughly tested after each downgrade.
In conclusion, this chapter serves as a road map for the whole book, where all the major activities involved in the process of cross-platform development are explained. Our focus was on the toolchain for the generation of target images and ELF—a widely adopted object file format. While knowing the detail of a standard object file format such as ELF is the business of those vendors who have an interest in producing interoperable development tools, knowing the big picture allows embedded software engineers to understand how an executable image is mapped into the target embedded system for static storage, as well as for run-time loading and execution.

A final note is that the development of many embedded systems demands teamwork among engineers with different skill sets. For instance, a complex embedded system might involve hardware engineers, software engineers, and systems engineers. Hardware engineers deal with hardware components, software engineers deal with software components, and systems engineers ensure that the software and hardware can cooperate smoothly to offer the desired services to the external environment.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

3.1In embedded systems, why should we distinguish run address from load address?

3.2 In a linker script file, how are related sections grouped together into one segment?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Build Target Images?	
	2 How Integrated Testing on Target works?	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is a compiler capable of generating executable code for a target platform that is different from the host platform.

_____2. It holds code and data suitable for linking with other object files to create an executable file or a shared object file.

_____3. It holds code and data suitable for execution. In an operating system, the basic unit of execution is called a process or process image, which is dynamically created from an executable file (say, as a result of an exec or spawn call).

_____4. It holds code and data suitable for further linking. It can be combined with other relocatable and shared object files to create another object file.

_____5. It is a program interpreter that simply loads a program into memory and transfers the execution control to it.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FAQs

1. How does memory mapping works?

An operating system is simply a "big" application. An operating system itself needs to be loaded from somewhere when the system starts. This job is done by a "starter" program, also known as an initial operating system loader. Where is such an operating system loader found when the system is powered on? Moreover, there are many embedded systems running without an operating system. Where should the instructions be loaded when such an embedded system starts? This brings us to memory mapping.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #4

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Lesson title: Microprocessor Primer (Part 1)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand what are Microprocessors;**
2. **Understand the different types of Microporcessors.**

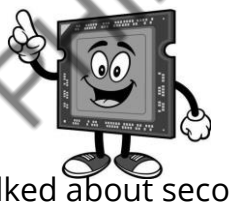
Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"Procrastination is the enemy of success". So, don't let your success to get beaten! Stay focus and let's build the road to success!

A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)



In the previous lesson, we talked about second part of Cross-Platform Development Process which includes Build Target Images, Transfer Executable File Object to Target, Integrated Testing on Target and System Production. Now, we proceed to the first part of Microprocessor Primer. In this lesson, we will discuss the the different workings of a Microprocessor and some commonly used Microprocessors in embedded systems.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Microprocessors?	
	2 What are the Microprocessors used in embedded systems?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

4.1 Introduction to Microprocessors

A microprocessor is a general-purpose central processing unit (CPU) manufactured on a single integrated circuit. To be useful, a microprocessor has to work with other components, such as a memory system for storing instructions and data, and external circuits to communicate with the peripheral environment.

A microprocessor is merely the processing core of an application system. Sometimes, a microprocessor and some commonly used circuits and components (e.g., memory, parallel I/O, serial I/O, clock circuit, etc.) are integrated together on a single chip, which is typically called a microcontroller. A microcontroller is truly a computer on a chip. Using a microcontroller can greatly ease the hardware architecture design, especially when it has all the necessary peripherals for the system to be developed.

The first commercial microprocessor was the Intel 4004, introduced by Intel Corporation in 1971 for electronic calculators. Since then, powerful, low-cost microprocessors have been used in myriads of embedded systems found almost everywhere: household appliances, cars, toys, DVD players, AV receivers, cell phones, and high-definition TVs, to mention only a few

4.1.1 Commonly Used Microprocessors

A few commonly used microprocessors are given in Table 4.1.

Intel 4004 is a 4-bit processor with only 2048 bits of addressable memory. Intel 8080 was introduced in 1974. It is an 8-bit processor with 16 address lines that can access 64 KB memories. The most successful family of processors started with the 16-bit Intel 8086, which sets the basis for the Intel x86 architecture (also referred to as IA-32).

Introduced in 1985, Intel 80386 features three operating modes: real mode, protected mode, and virtual mode. The protected mode allows the processor to address up to 4 GB of memory. The virtual mode makes it possible to run one or more programs in a protected environment. The Pentium processor was the first x86 processor with superscalar architecture. ThePentium processor also features a 64-bit external data bus, which doubles the amount of

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

information it is possible to read or write on each memory access.

Pentium Pro was introduced in 1995. It can not only access a bigger memory space, but also implements a new architecture called microarchitecture—a decoupled, 12-stage superpipelined architecture. In 2001, Intel introduced Itanium, a family of 64-bit

Table 4.1 The characteristics of some commonly used microprocessors

Architecture	Year	Family	Processing Width (bits)[a]	Address Width (bits)	Addressable Memory Space (1 byte = (8 bits)	External Memory Bus: Data (address)
Intel	1971	4004	4	12	$2^{12} = 4\,K(\frac{1}{2}B)$	4
	1974	8080	8	16	$2^{16} = 64\,KB$[b]	8
	1978	8086	16	20	$2^{20} = 1\,MB$	16
	1985	80386	32	32	$2^{32} = 4\,GB$	32
	1993	Pentium	32	32	$2^{32} = 4\,GB$	64
	1995	Pentium Pro	32	36	$2^{36} = 64\,GB$	64
	1999	Pentium III	32	36	$2^{36} = 64\,GB$	64
	2001	Itanium	64	64	2^{64} B	128
Motorola	1979	68000	32 (16)	24	$2^{24} = 16\,MB$	16
Microchip	2000	PIC18F2XK20	16	C[c]: 21	$2^{21} = 2\,MB$	No
				D[d]: 12	$2^{12} = 4\,KB$	
	2000	PIC18F8X20 PIC18F97J60	16	C: 21	$2^{21} = 2\,MB$	16 (20)
				D: 12	$2^{12} = 4\,KB$	
	2007	PIC32MX	32	32	$2^{32} = 4\,GB$[e]	Not yet. 16 (16) through PMP
ARM	2001	ARM9E	32	C: 18	$2^{18} = 256\,KB$	32
				D: 18	$2^{18} = 256\,KB$	32
	2006	Cortex-M3	32	32	$2^{32} = 4\,GB$[e]	32

PMP, parallel master port.
[a]The processing width of a processor is the same as the width of the general-purpose registers and the arithmetic logic unit of the processor. It is also called the word width of the processor.
[b]When the address width is larger than the processing width, memory segmentation is done by the processor internally.
[c]Program code space.
[d]Data (random-access memory) space arranged in banks of size 256 bytes.
[e]Program code space and data space are still separated, but they are mapped into a unified virtual memory space.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

microprocessors that implement the IA-64 architecture. Itanium processors are targeted for enterprise servers and high-performance computing systems.
Motorola 68000 (also called 68K) is a 16-/32-bit microprocessor. Although introduced in 1979, it is still in use in embedded applications. For example, it has been used in low-end printers such as HP's LaserJet introduced in 1984, and Apple's LaserWriter introduced in 1985, in game consoles such as Sega's System 16, Mega Drive (Genesis) console, and Saturn console.

The PIC family of microcontrollers is made by Microchip Technology. PIC microcontrollers are popular in both industry and education because of their low cost, large user base, and wide availability of development tools. PIC microcontrollers feature on-chip program and data memory as well as many peripheral components. Some PIC microcontrollers (e.g., PIC18F8X20 and PIC18F97J60) even have an external memory interface to expand the internal memory space.

ARM processors have been used extensively in consumer electronics, including personal digital assistants, tablets, cell phones, music players, handheld game consoles, and computer peripherals such as hard drives and routers. As of 2009, ARM processors accounted for approximately 90% of all embedded 32-bit reduced instruction set computing (RISC) processors. In 2010, over 6.1 billion ARM-based chips were sold, representing over 95% of the smartphone market, 10% of the mobile computer market, and 35% of the digital TV and set-top box market.

As integrated circuit technology advances, it becomes feasible to manufacture more and more complex processors on a single chip. It is clear that high-end applications demand more address space and computing power than those offered by 32-bit processors, and the transition from 32-bit computing to 64-bit computing has become the trend.

4.1.2 Microprocessor Characteristics

4.1.2.1 Architectures

As far as memory access is concerned, a microprocessor may belong to either von Neumann architecture or Harvard architecture.

The von Neumann architecture is named after the mathematician and early computer scientist John von Neumann. The von Neumann architecture has two features:

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• Data and instructions (executable code) are stored in the same address space. On the one hand, this allows for self-modifying code. On the other hand, this opens security holes. For instance, a program may attempt to run instructions from memory that contains data, or a program might write data into memory containing instructions.

• The processor interfaces with memory through a single set of address/data buses. Since an instruction fetch and a data operation cannot occur at the same time because they share a common bus, this often limits the performance of the system. This is referred to as the von Neumann bottleneck.

The Harvard architecture is named after the Harvard Mark I computer. The Harvard architecture has two features:

• Data and instructions (executable code) are stored in separate address spaces. For instance, the instruction space may be accessed by 20 address lines, while addresses in the data space may only have 16 bits. In addition, program memory is typically read-only memory (ROM). It is impossible for program contents to be modified by the program itself.

• There are two sets of address/data buses between the processor and the memory. Since instruction fetches and data operations are carried on separate buses, it is possible to access program memory and data memory simultaneously.

Modern microprocessor designs (e.g., ARM) incorporate aspects of both Harvard and von Neumann architectures. For instance, a modified Harvard architecture has separate instruction and data caches that are backed by a common address space. While the processor executes from cache, it acts as a pure Harvard architecture. When accessing backing memory, it acts like a von Neumann architecture (where code can be moved around like data).

As far as the instruction set is concerned, a microprocessor may be one of two types: a complex instruction set computing (CISC) processor or an RISC processor.

A CISC processor has multiple addressing modes and runs “complex instructions” where a single instruction may execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store). While this leads to high code density, it often requires manual optimization of assembly code for embedded systems.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

An RISC processor runs compact, uniform instructions where the amount of work any single instruction accomplishes is reduced (e.g., separate instructions for I/O and data processing). Such compact instructions allow effective compiler optimization and facilitate pipelining, typically leading to higher performance than CISC. As an overhead, however, RISC inevitably produces more lines of code (larger memory footprint) than CISC.

Today, CISC processors dominate the personal computer market and are used in a significant fraction of the low- and mid-range servers and workstations. RISC processors have been successfully used across a wide range of platforms, from mobile devices such as cellphones and tablets to some of the world's fastest supercomputers.

Figure 4.1 gives some example microprocessors for each category, where SHARC (for "super Harvard architecture") is a processor family that offers many features required by digital signal processor applications: exceptional core and memory performance and outstanding I/O throughput

	Von Neumann	Harvard
CISC	X86 (8086, Pentium) Motorola 68000	SHARC (DSP)
RISC	ARM7, SPARC, MIPS, PowerPC	ARM9, PIC

Figure 4.1
Microprocessor Architecture Types

4.1.2.2 Processing width

The processing width is an important characteristic of a processor design. For example, the Intel 8086 is a 16-bit processor, where the number 16 is its processing width. The processing width of a processor is exactly the width of its working memory (registers).

The term "word" also refers to the largest processing unit that can be transferred to and from the working memory in a single operation of a processor. In other words, the number of bits in a word (the word size or word width) is the same as the processing width (register width). For instance, PIC18F8720 is a 16-bit processor; its word size is 16 bits and a word is composed of 2 bytes. ARM926EJ-S is a 32-bit processor; its word size is 32 bits and a word is composed of 4 bytes.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Modern processors usually have a word size of 16, 32, or 64 bits.

4.1.2.3 I/O addressing

A microprocessor typically accesses I/O devices in two ways.

I/O devices can be placed in a microprocessor's memory address space. This approach is called memory-mapped I/O, where I/O devices and memory components are indistinguishable to the processor. Memory-mapped I/O simplifies coding and testing. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the Intel 80386 processor can use the MOV instruction to transfer data between any register and a memory-mapped I/O port. The Motorola 68000 uses memory-mapped I/O. PIC family processors also have all peripherals memory mapped (through ports and port registers).

Some processors may feature a special signal pin (e.g., M/IO in the Intel 8086 and Pentium processors) to indicate whether a memory device or an I/O device is accessed. In such a case, the designer can choose whether to map I/O devices into the memory space or use a separate I/O address space. In the latter case, all I/O devices are treated differently from normal memory locations. Typically special I/O instructions are needed to access I/O ports.
4.1.2.4 Reset vector

The reset vector of a processor is the default location where, upon a reset, the processor will go to find the first instruction to execute. In other words, the reset vector is a pointer or address where the processor should always begin its execution. This first instruction typically branches to the system initialization code.

Most microprocessors have reset vectors fixed at the two ends of their address spaces. Here are a few examples:

• The Intel 8086 processor has its reset vector at FFFF0h, the high end of its address space.
• PIC18 processors have the reset vector located at the low end, 0000h.
• The Motorola 68000 processor has a 1024-byte vector table beginning at 000000h. The first entry is the reset vector, which is at 000000h. This vector table also contains pointers to routines used by the processor, operating system, and users.
• ARM family processors have the address 0x00000000 reserved for the vector table (including reset vector, undefined instruction vector, software interrupt vector, interrupt request vector, fast interrupt vector), and the reset vector is at 0x00000000. On some

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

processors (e.g., ARM926EJ-S) the vector table can be optionally located at a higher address in memory, 0xffff0000.

4.1.2.5 Endianness

The term "endian" or "endianness" refers to the ordering of individually addressable units (e.g., bytes) within a larger data item (e.g., word) as stored in external memory (or, sometimes, as sent on a serial connection).

As far as microprocessors are concerned, a big-endian processor stores the most significant byte (MSB) first (i.e., at the lowest byte address), while a little-endian processor stores the least significant byte (LSB) first (see Figure 4.2). When we write the hex value 0x0a0b0c0d from left to right, we are implicitly writing in big-endian style.

Different processors order their multibyte data (i.e., 16-, 32-, or 64-bit words) in different ways. For example, the Intel X86 family and PIC processors use little-endian mode, and the Motorola 68000 family uses big-endian mode. Some processors, such as ARM processors, can be set up to be in either little-endian or big-endian mode.

Endianness is typically transparent to a user of a computer. However, difficulty arises when different types of computers attempt to communicate with one another over a network. Internet Protocol (IP) defines big endian as the standard network byte order used by IP and many higher-level protocols over IP for all numeric values. When communicating over a

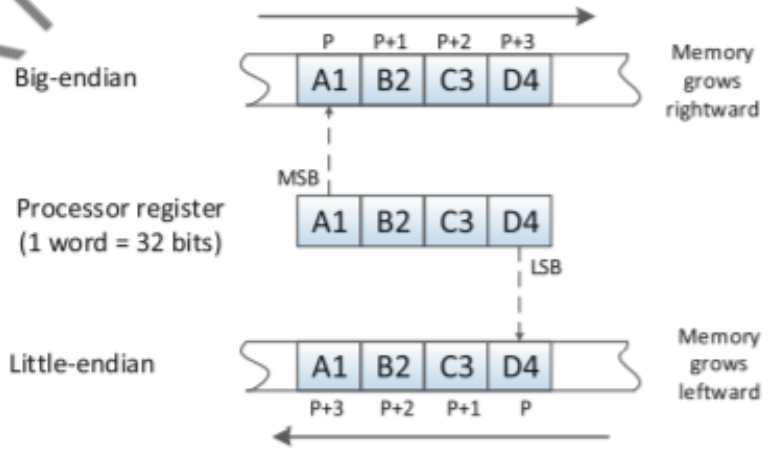


Figure 3.2
Microprocessor endianness

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

network composed of both big-endian and little-endian machines, high-level software should format packets of data in network byte order. The Berkeley sockets application programming interface defines a set of functions2 for little-endian host machines to convert 16-bit and 32-bit integers to and from network byte order.

A software system is called “endian clean” when it always reads and writes a multibyte data item as a whole rather than byte-by-byte. An “endian-clean” software system can be recompiled with no changes for big-endian or little-endian machines. It is encouraged to write “endian-clean” software, whenever possible, for networked embedded systems.

4.2 Microchip PIC18F8720

Microchip PIC18F8720 is a family of high-performance RISC microprocessors. PIC18F8720 features a hardware stack for storing return addresses and a wide range of hardware interfaces including 10-bit A/D converters with 16 input channels, five timers, an external memory interface, and nine general-purpose I/O ports (one of which can be reconfigured as an 8-bit parallel slave port for direct processor-to-processor communications). Figure 4.3 shows the pin diagram of an 80-pin PIC18F8720 microprocessor, where many pins are physically multiplexed. For instance, the external memory interface (A19:A16, AD15:AD0) is multiplexed with I/O ports D, E, and H.

The PIC18F8720 device runs from a clock (the OSC1 pin) that is four times faster than its instruction cycle. The four clock pulses are a quarter of the instruction cycle in length and are referred to as Q1, Q2, Q3, and Q4. Generally speaking, instruction fetching and execution are pipelined: while a 2-byte instruction is executed in an instruction cycle, the next 2-byte instruction can be fetched during the same cycle.

4.2.1 Memory Organization

PIC18F8720 devices have separate spaces for data memory and program memory, which is illustrated in Figure 4.4.

Data space in electrically erasable programmable ROM (EEPROM). PIC18F8720 devices have 1024 bytes of data EEPROM, which is a byte-addressable memory space that has been optimized for the storage of frequently changing values (e.g., program variables that are updated often). Variables that change infrequently (such as constants, IDs, etc.) should be stored in flash program memory.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

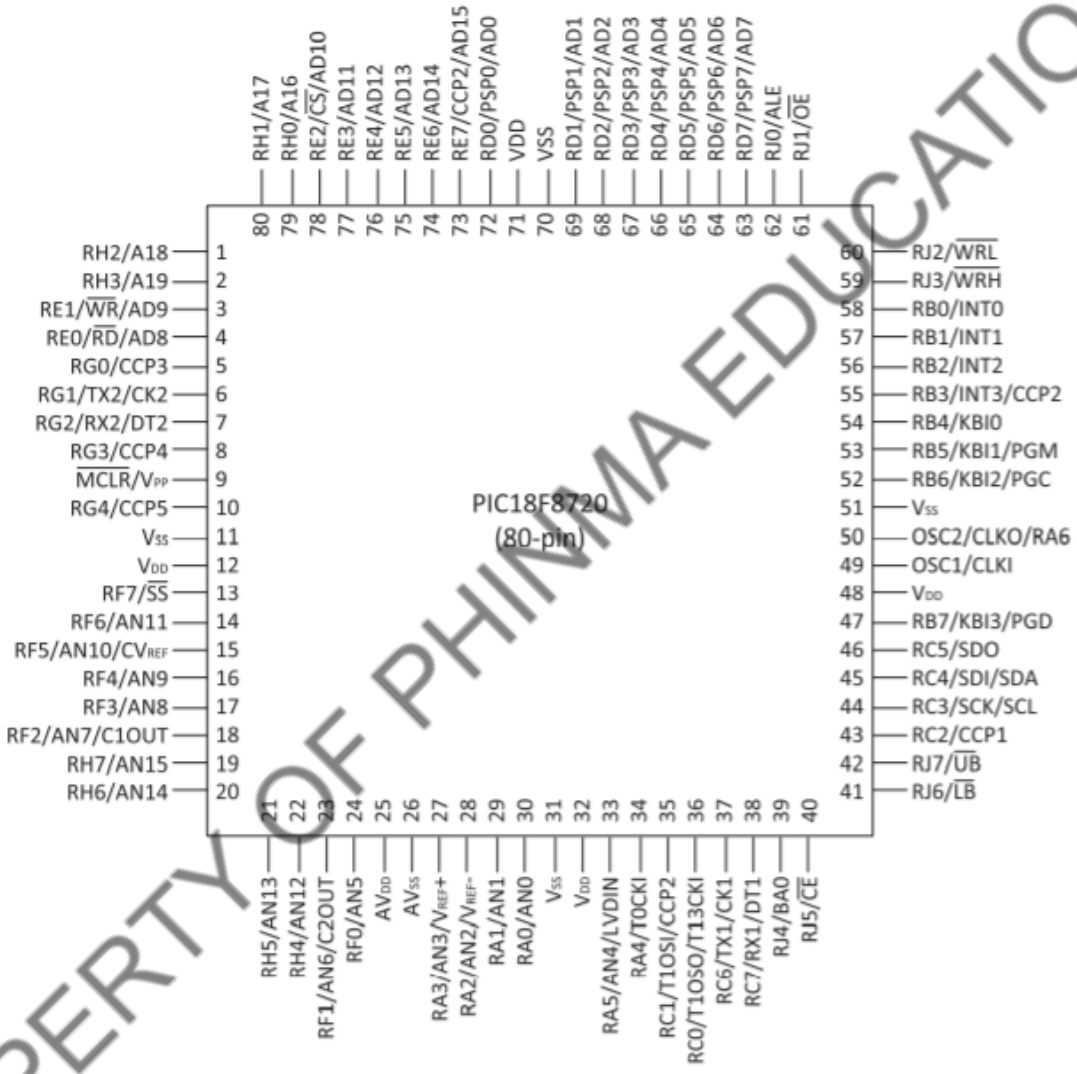


Figure 4.3
Microchip 16-bit 80-pin PIC microprocessor.

When interfacing with the data EEPROM block, the register EEDATA holds the 8-bit data for read/write, while the register EEADR and the two least significant bits (LSbs) of EEADRH hold the address of the EEPROM location being accessed.

Data space in random-access memory (RAM). As shown in Figure 4.4, PIC18F8720 devices also have a data RAM space that contains both special-function registers (SFR) and

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

general-purpose registers (GPR). The SFRs are used for control and status of the microcontroller and peripheral functions, while GPRs are used for storing application data such as intermediate computational values, local variables of subroutines, and task contexts.

Each register in the RAM space has a 12-bit address. To enable rapid access to those registers, a banking scheme is implemented where the whole space is partitioned into 16 banks of

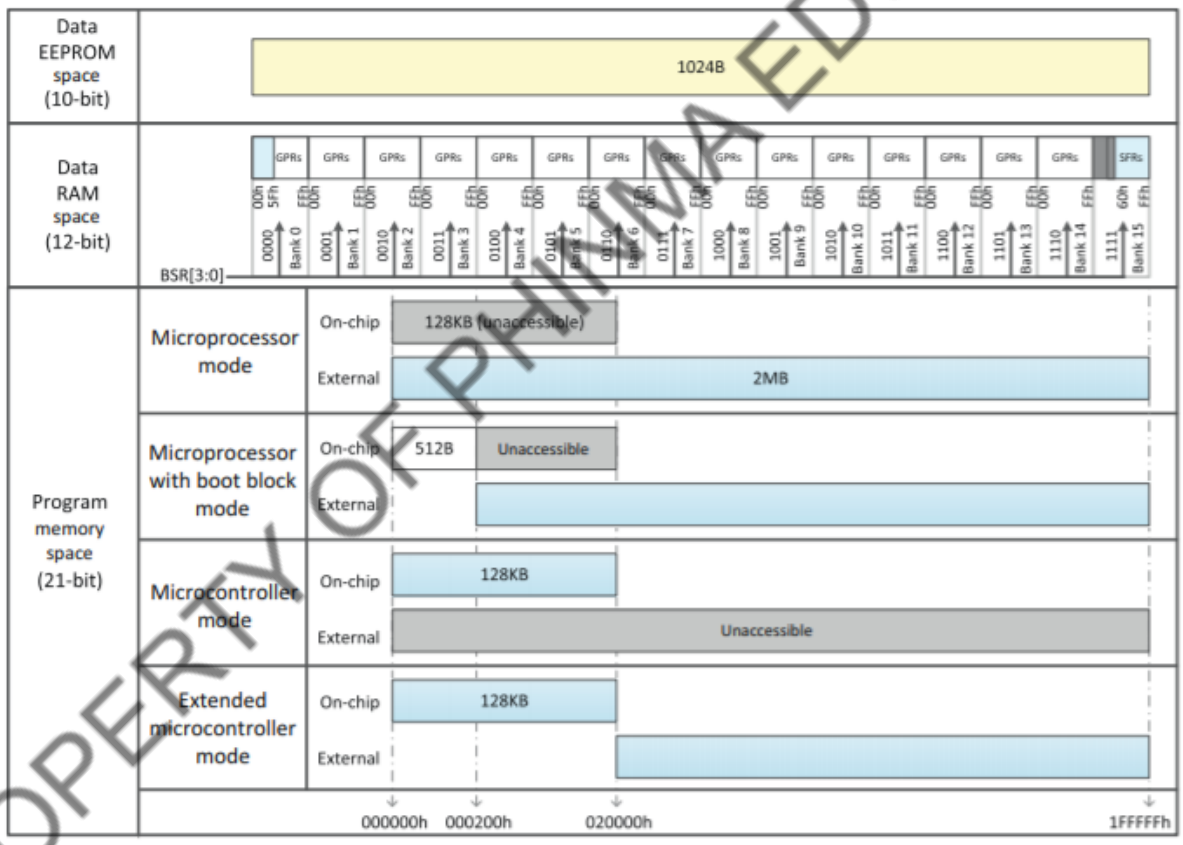


Figure 4.4
PIC18F8720 memory organization

256 bytes each. To directly access a register, the lower 8 bits of its address are embedded in the instruction, and the upper 4 bits of its address, which specify the bank to be accessed, are taken from the lower 4 bits of the bank select register (BSR[3:0]).

GPRs start at the first location of bank 0 and grow upward up to the last location of bank 14.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Bank 15 contains 160 SFRs with addresses ranging from F60h to FFFh; its lower portion (from F00h to F5Fh) is unused.

The first segment (96 bytes) of bank 0 and the SFR segment (160 bytes) of bank 15 also form a so-called access bank, which allows the commonly used registers (SFRs and some GPRs) to be accessed in a single instruction cycle. A reserved bit in the instruction word is used to specify if the operation is to occur in the bank specified by the bank select register or in the access bank.

Program memory space. PIC18F8720 has 128 KB of on-chip flash memory. Moreover, it implements a 21-bit program counter, which is capable of addressing a 2 MB program memory space through the external memory interface. Depending on the two LSbs of the CONFIG3L register4, PIC18F8720 may operate in one of four modes:

(i) Microcontroller mode. This is the default mode. Under this mode, the processor can access only the on-chip flash memory. Attempts to read above the physical limit of the on-chip flash memory return 0's.

(ii) Microprocessor: Under this mode, the processor can access only the external program memory; the contents of the on-chip flash memory are ignored. The 21-bit program counter permits access to a 2 MB linear program memory space.

(iii) Microprocessor with boot block. Under this mode, the processor accesses the boot block (from addresses 000000h to 0001FFh)5 of the on-chip flash memory. Above this, external program memory is accessed all the way up to the 2 MB limit. Program execution automatically switches between the two memories, as required.

(iv) Extended microcontroller mode. Under this mode, the processor can access its entire on-chip flash memory; above this, the device accesses external program memory up to the 2 MB program space limit. The execution automatically switches between the two memories, as required.

The program memory space is typically used for storing instructions. The address of the instruction to fetch for execution is specified by the 21-bit program counter, where the low byte, the high byte, and the upper 5 bits are contained in registers PCL, PCH, and PCU, respectively.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The program memory is addressed by bytes. However, since PIC18F8720 uses a 16-bit instruction set, the LSb of the PCL register is fixed to a value of 0 to prevent the program counter from becoming misaligned with word instructions. In other words, the program counter increments by 2 to address sequential instructions in the program memory.

The program memory space can also be used for storing data. A block containing data is not required to be word aligned. There are two instructions that allow a processor to move bytes between the program memory space and the data RAM: (a) table read TBLRD, which retrieves data from program memory (or data EEPROM) and places it into the data RAM space, and

(b) table write TBLWT, which stores data from the data RAM space in program memory (or data EEPROM). Table read and table write operations move data between these memory spaces through an 8-bit table latch register (TABLAT).

A table pointer TBLPTR is used in reads, writes, and erases of the program memory. TBLPTR comprises three SFRs—TBLPTRU, TBLPTRH, and TBLPTRL—which jointly form a 21-bit-wide address pointer.

To read a byte from the flash program memory, simply set TBLPTR to point to its address in the program space, then execute the TBLRD instruction, which will place the byte into TABLAT.

Writing to the flash program memory is a little complicated because the minimum programming block is 8 bytes (four words). In particular, prior to a flash programming operation, the TBLWT instruction has to be executed eight times, with each essentially writing 1 byte contained in TABLAT to one of the eight holding registers. At the end of updating the holding registers, instruction execution is halted and a long write cycle is started to write the contents of the eight holding registers to the flash program memory.

In all modes, a PIC18F8720 processor has complete access to the data RAM and data EEPROM spaces. In addition, the processor, except for the microcontroller mode, can access external memory devices (such as flash memory, erasable programmable ROM (EPROM), and static RAM (SRAM)) as program or data memory through the external memory interface. Depending on the values of the two LSbs of the MEMCON register, there are three operating modes for the external memory interface: word write, byte select, and byte write.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

4.2.2 Word Write Mode

This mode allows instruction fetches and table reads from, and table writes to, all forms of 16-bit (word-wide) external memories (EPROM, flash memory, or SRAM). An example configuration of word write mode is shown in Figure 4.5, where, in contrast to Figure 4.3, only those pins that are relevant to the external memory interface are shown.

Whenever the microprocessor accesses external memory, the chip enable signal $\overline{CE}$ is active (asserted low); it is thus connected to the $\overline{CE}$ pin of the flash memory device.

To allow the microprocessor to fetch data from external memory, the corresponding output enable signals $\overline{OE}$ (active low) are connected. $\overline{OE}$ is inactive during a cycle where a table write is executed. During a cycle where an instruction is fetched or a table read is executed, $\overline{OE}$ is asserted at the beginning of Q3 and deasserted at the end of Q4. Data (16-bit word) are fetched from external memory at the low-to-high transition edge of $\overline{OE}$.

The write high control signal $\overline{WRH}$ is active (asserted low) whenever the microprocessor writes to an odd address (or the high byte of a word). It is connected to $\overline{WR}$ of the memory device so

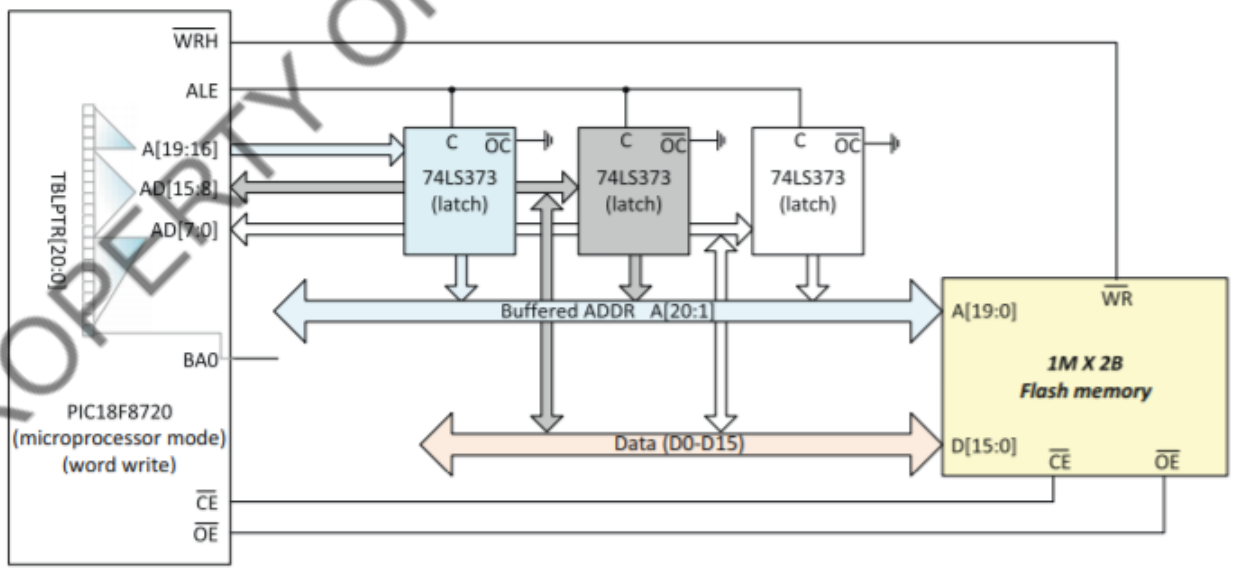


Figure 4.5
PIC18F8720 word write mode.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

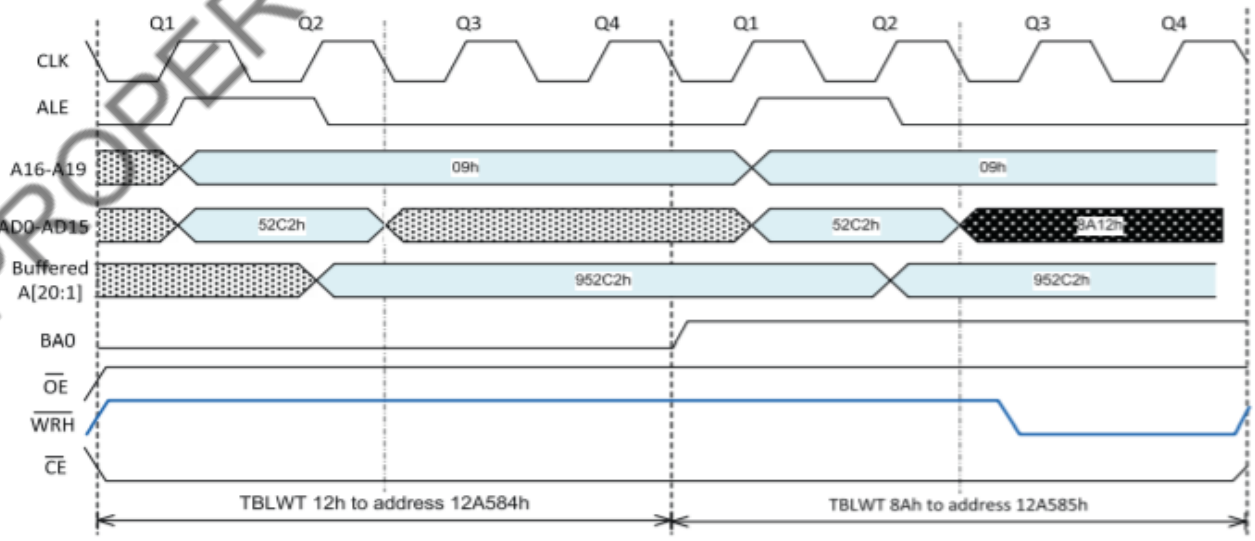
that the device is ready for writing whenever the microprocessor attempts to write the high byte of a word.

The external memory interface has 20 address lines: A[19:16] and AD[15:0]. Internally, the 21-bit table pointer TBLPTR (similarly the program counter) is mapped to the external memory interface as follows:
• the least significant bit TBLPTR[0] is copied to BA0;
• TBLPTR[20:17] appear on address pins A[19:16];
• TBLPTR[16:9] appear on pins AD[15:8]; and
• TBLPTR[8:1] appear on pins AD[7:0].

Consequently, A[19:16] together with AD[15:0] allow memory access based on a word boundary. If necessary, BA0 can be used to access memory by byte.

Because the pins AD[15:0] are multiplexed for both data and addresses, the control pin ALE (for "address latch enable") is used to enable the address latch devices (74LS373), so that the address on the external address bus is still valid while data are being transferred. Note that to be consistent with the table pointer, A[20:1] are used to name the buffered address lines.

Figure 4.6 gives a timing diagram illustrating the signal interactions for two consecutive table write operations while the microprocessor accesses the external memory in word write mode. First, during Q1 of an instruction cycle, ALE is enabled while the address information is placed on pins AD[15:0] and A[19:16]. On the falling edge of ALE, the address is latched and available on the external bus.



Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Second, this mode makes a distinction between TBLWT instruction cycles for even or odd addresses. During a TBLWT cycle to an even address, that is, TBLPTR[0] or BA0 is 0, the TABLAT data are transferred to a holding latch and AD[15:0] is tristated for the data portion of the bus cycle. The WRH signal is not activated. During a TBLWT cycle to an odd address i.e., TBLPTR[0] or BA0 is 1, the TABLAT data are presented on the upper byte of the AD[15:0] bus. At the same time, the contents of the holding latch are presented on the lower byte of the AD[15:0] bus. The WRH signal is activated and the 16-bit data are written to the corresponding word location.

Last but not least, it is worth noting that the address 12A584h used by the table pointer becomes 952C2h; this buffered address does not reflect the LSb of 12A584h. As an exercise, check that 952C2h×2 XOR BA0 equals 12A584h in the first cycle, and equals 12A585h in the second cycle. This is because the LSB of a word location is accessed in the first cycle (BA0 = 0), while the MSB of the word location is accessed in the second cycle (BA0 = 1).

4.2.3 Byte Select Mode

This mode allows instruction fetches and table reads from, and table writes to, 16-bit external memories with byte selection capability, as well as 8-bit external memories. In this mode, the write high signal WRH is asserted whenever the microprocessor writes to a memory location, regardless of an odd or even address. An example configuration of byte select mode is shown in Figure 4.7, where WRH is connected to the WR pin of each memory chip.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

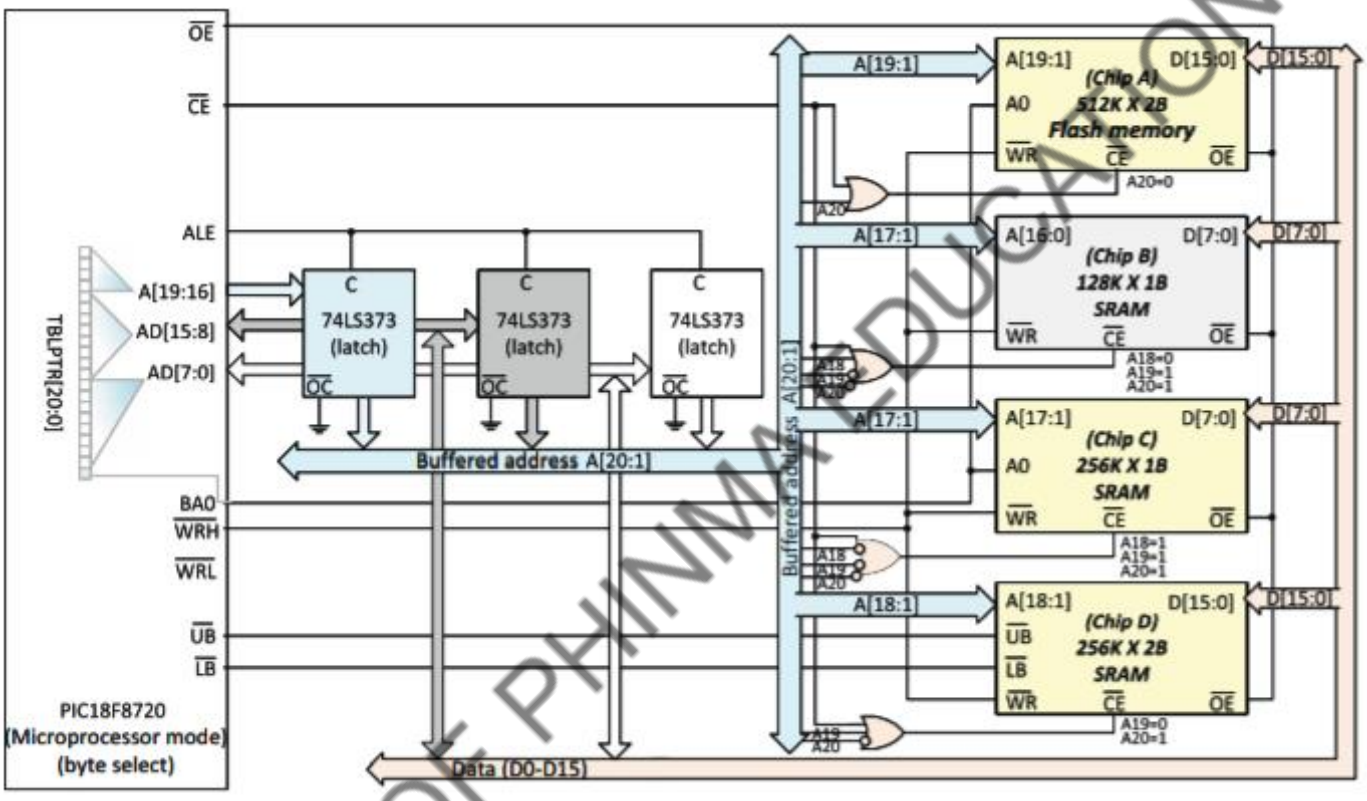


Figure 4.7
PIC18F8720 byte select mode.

First, notice that there are four memory chips: chip A and chip D are word-wide memory devices, and chip B and chip C are byte-wide memory devices. At any given time, only one of the four devices can be enabled for reading or writing. This is achieved by the OR gates. Specifically, the chip enable signal $\overline{CE}$ together with some of the address lines is used to select memory chips:

- Chip A is selected (active) only when $\overline{CE}$ is asserted and A[20]=0.
- Chip B is selected only when $\overline{CE}$ is asserted, A[20]=1, A[19]=1, and A[18]=0.
- Chip C is selected only when $\overline{CE}$ is asserted, A[20]=1, A[19]=1, and A[18]=1.
- Chip D is selected only when $\overline{CE}$ is asserted, A[20]=1, and A[19]=0.

Second, BA0 is connected to the A0 pin of chip A. While A[19:1] allows the microprocessor to address a word location on chip A, A0 will allow the microprocessor to individually access the MSB or LSB of the word. The A[0] pin of chip C is also connected to BA0. This ensures that chip C gets a contiguous block of addresses. Chip B has almost the same wiring as chip C except that its A[0] is connected to bus line A[1] rather than BA0. Because of this, the block

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

of addresses allocated to chip B is not contiguous: each location on chip B gets one odd address and one even address! In other words, the microprocessor can use two different addresses to access each location on chip B.

Third, chip D is a word-wide SRAM, which, according to the JEDEC memory standards, uses $\overline{UB}$ or $\overline{LB}$ to indicate whether the upper byte or the lower byte is to be selected. Thus, pins $\overline{UB}$ and $\overline{LB}$ of chip D are connected to the corresponding microprocessor pins, respectively. $\overline{UB}$ is asserted when the microprocessor accesses an odd address, and $\overline{LB}$ is asserted when the microprocessor accesses an even address.

The memory mappings of the memory devices are given in Table 4.2. These four chips together form a space with 2 MB addresses, but remember that although chip B has 128 KB, it is allocated with 256K addresses: two for each byte location.

Figure 4.8 gives a timing diagram illustrating the signal interactions for two consecutive table write operations while the microprocessor is in byte select mode.

It is worth noting that $\overline{WRH}$ is asserted in each write cycle, and the TABLAT data are copied on both the MSB and the LSB of the data bus (e.g., 1212h) This is not a problem for word-wide memory devices (say, chip A or chip D), because either BA0 or $\overline{UB}$/$\overline{LB}$ (they originate from the LSb of the TBLPTR register) can be used to select the right byte to be written.

Also note that the microprocessor writes to an even address in the first cycle, so $\overline{BA0}$ = 0 and $\overline{LB}$ is asserted. It then writes to an odd address in the second cycle, so BA0 = 1 and $\overline{UB}$ is asserted. As an exercise, check on which chip the address 12A584h appears.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Table 4.2 The memory mapping of Figure 4.7

Chip	Memory width	Address Bus: A[20] —— A[1] $\overline{CE}$	Chip Address	A0	Address Range	Size
A	Word	0	++++++++++++++++++	+[a]	0 0000 0000 0000 0000 0000~ 0 1111 1111 1111 1111 1111	1 MB
B	Byte	110	++++++++++++++++		1 1000 0000 0000 0000 000x~ 1 1011 1111 1111 1111 111x[b]	128 KB
C	Byte	111	++++++++++++++++	+	1 1100 0000 0000 0000 0000~ 1 1111 1111 1111 1111 1111	256 KB
D	Word	10	++++++++++++++++++	+	1 0000 0000 0000 0000 0000~ 1 0111 1111 1111 1111 1111	512 KB

[a] A bit marked by + indicates that it is used by the device; different value combinations of those "+" bits refer to different locations on the device.
[b] A bit marked by x indicates that it is unused by the device. When there are unused bits, each location of the device may be accessed by more than one address. In this case, the last bit can be 0 or 1, implying that each location on chip B virtually has two addresses.

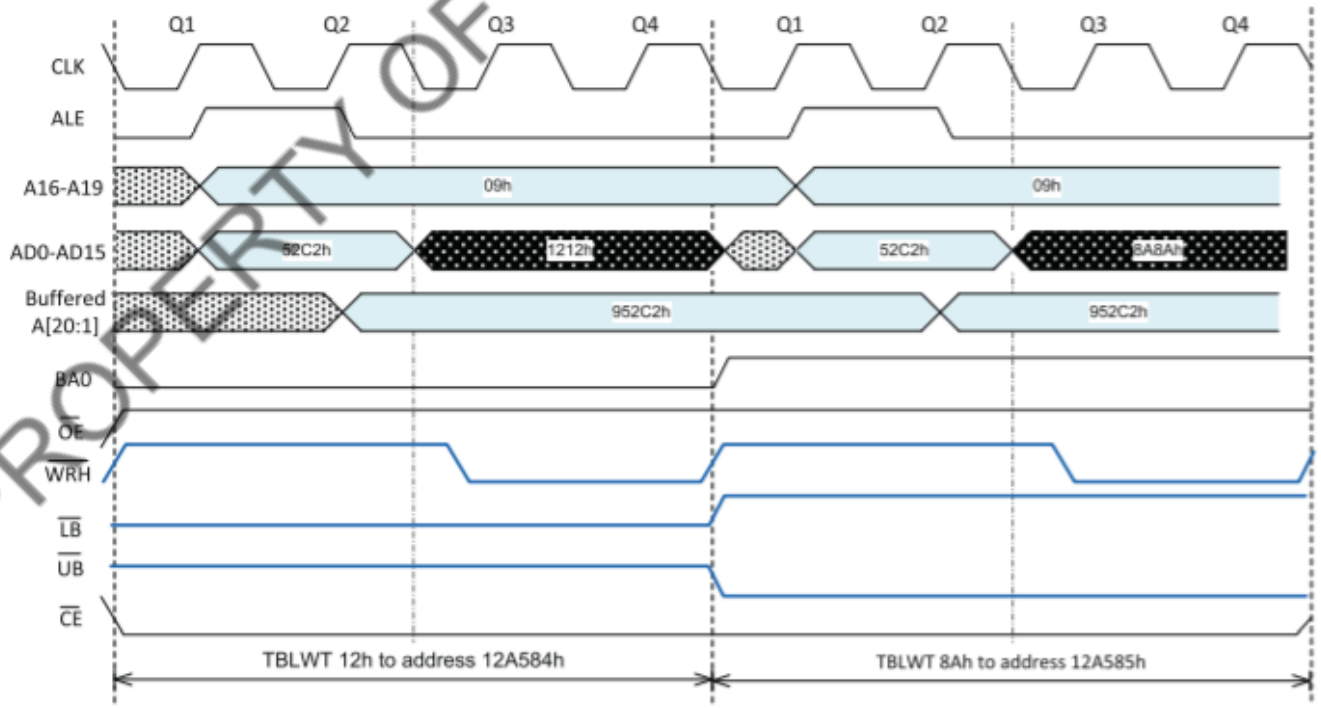


Figure 4.8 - PIC18F8720 byte select mode timing diagram.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3.2.4 Byte Write Mode

This mode allows table reads from, and table writes to, 8-bit-wide external memories. In this mode, during a TBLWT instruction cycle, the TABLAT data are presented on both the upper and the lower bytes of the AD[15:0] bus. The appropriate $\overline{WRH}$ or $\overline{WRL}$ control line is asserted on the LSb of TBLPTR.

An example configuration of byte write mode is shown in Figure 4.9. Like the other modes, AD<15:0> from the microprocessor are mapped through the latch to A<16:1> of the buffered address bus, and A<19:16> are mapped directly to A<20:17>.

Let us first look at the wiring of chip A and chip B. They are of the same type, of the same size (512 KB), and even have the same chip selection signal (A[20]=0). They differ in two ways: (a) the data pins of chip A are connected with the upper byte (D[15:8]) of the data bus, while the data pins of chip B are connected with the lower byte (D[7:0]) of the data bus; (b) the write enable pin $\overline{WR}$ of chip A is connected to $\overline{WRH}$, while the write enable pin $\overline{WR}$ of chip B is connected to $\overline{WRL}$. This implies that data are written to chip A when the address is odd ($\overline{WRH}$ is asserted as the LSb of TBLPTR is 1), written to chip B when the address is even ($\overline{WRL}$ is asserted as the LSb of TBLPTR is 0).

The wiring of chip C is almost the same as that of chip B, except that it is selected when A[20] is 1. This ensures that chip C will get a different block of addresses. Note that chip C

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

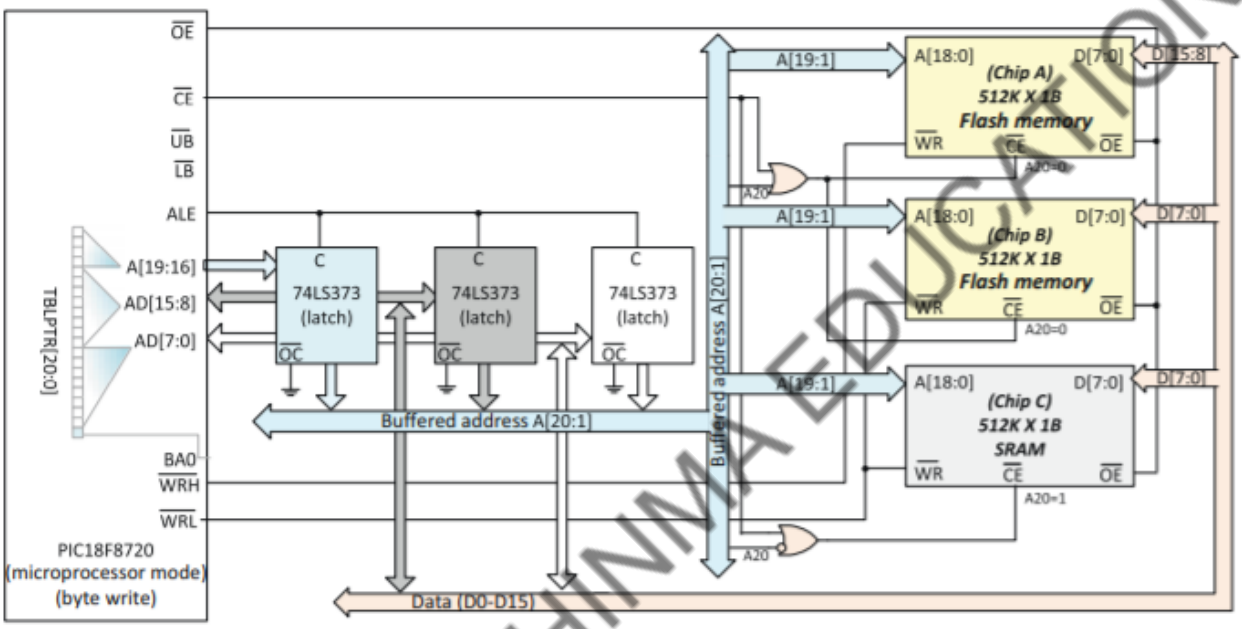


Figure 4.9
PIC18F8720 byte write mode.

permits table operations to only even addresses. Particularly, a table write to an even address of chip C will copy the contents of TABLAT to both bytes of the AD<15:0> bus and activate the WRL signal. The lower byte will then be written to chip C. A table write to an odd address of chip C will also copy the contents of TABLAT to both bytes of the AD<15:0> bus. However, this time, the WRH signal is activated instead. In this case, nothing will be written to chip C. This is similar to the case for chip B in Figure 4.7: although the usage of the memory chips is 100%, a noncontiguous block of addresses is assigned to them. One difference is that each location on chip C gets an even address, while each location on chip B in Figure 4.7 gets an even address and an odd address. The reason is that in byte select mode, WRH is asserted regardless of whether the address is even or odd.

Figure 4.10 gives a timing diagram illustrating the signal interactions for two consecutive table write operations while the microprocessor is in byte write mode. Note that WRL is asserted in the first cycle because an even address is accessed, while WRH is asserted in the second cycle because an odd address is accessed. In each write cycle, the TABLAT data are copied on both the MSB and the LSB of the data bus (e.g., 1212h).

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

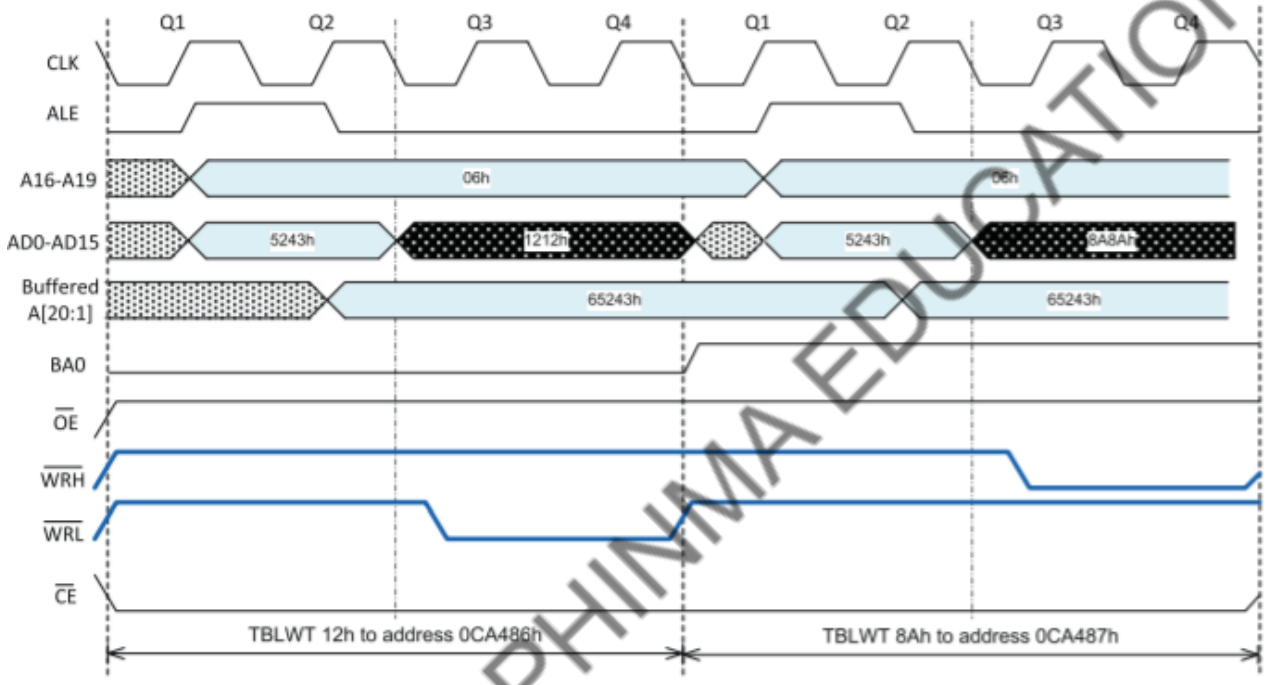


Figure 4.10
PIC18F8720 byte write mode timing diagram.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

4. 1 State an example of an embedded system that uses microchip PIC18F8720 as a microprocessor. Explain the function of the system.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Microprocessors?	
	2 What are the Microprocessors used in embedded systems?	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is a general-purpose central processing unit (CPU) manufactured on a single integrated circuit.
_____2. This 8-bit Intel processor was introduced in 1974.
_____3.-5. What are the three operating mode of Intel 80386?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FAQs

1. What is the first Microprocessor?

The first commercial microprocessor was the Intel 4004, introduced by Intel Corporation in 1971 for electronic calculators. Since then, powerful, low-cost microprocessors have been used in myriads of embedded systems found almost everywhere: household appliances, cars, toys, DVD players, AV receivers, cell phones, and high-definition TVs, to mention only a few.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #5

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

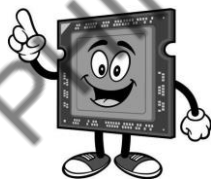
Lesson title: Microprocessor Primer (Part 2)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Know how Intel 8086 works;**
2. **Understand the operations of Intel Pentium**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"Lost time is never found again."... So what are you waiting for? Let's start this thing up and never waste time!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about the first part of Microprocessor Primer which includes the different workings of a Microprocessor and some commonly used Microprocessors in embedded systems. Now, we move on to the second part which particularly focusing on Intel 8086 and Intel Pentium.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Intel 8086?	
	2 Is Intel Pentium same as Intel 8086? Why?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

5.1 Intel 8086

Intel 8086 is a 16-bit microprocessor that represents the first design of the x86-family architecture. The Intel 8086 has a von Neumann architecture and belongs to the CISC category. Its pin diagram is shown in Figure 5.1

Intel 8086 supports two hardware modes (hard wired into the circuit and unchangeable by software): maximum mode and minimum mode. In maximum mode pin 33 (MN/$\overline{MX}$) is wired to ground, and in minimum mode pin 33 is wired to voltage. Changing the state of pin 33 also changes the function of some other pins as indicated in Figure 5.1. The minimum mode is intended for small single processor systems, while the maximum mode is for medium-sized or large systems that use more than one processor (e.g., 8087 coprocessor in an IBM PC).

CLK provides the basic timing for the processor and bus controller. Each bus cycle (read, write, or interrupt acknowledge) consists of at least four clock cycles, which are referred to as T1, T2, T3, and T4. The 8086 processor has its address lines and data lines multiplexed (AD0-AD15). This means that the same pins are used to carry both address and data information at different times: the address is emitted from the processor during T1 and data transfer occurs on the bus during T3 and T4. In the minimum mode, the address latch enable signal ALE can be used to enable chips for latching addresses.

The bus high enable signal $\overline{BHE}$ is asserted (active low) during T1 of a bus cycle when a byte is to be transferred on the high portion (the most significant half) of the data bus, that is, pins D15:D8.

The interrupt request signal INTR is a level-triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. It can be internally masked by software resetting the interrupt enable bit. The nonmaskable interrupt signal NMI is an edge-triggered input, which has higher priority than the maskable interrupt request signal INTR. The interrupt acknowledge signal INTA is active (low) during T2 and T3 of each interrupt acknowledge cycle.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

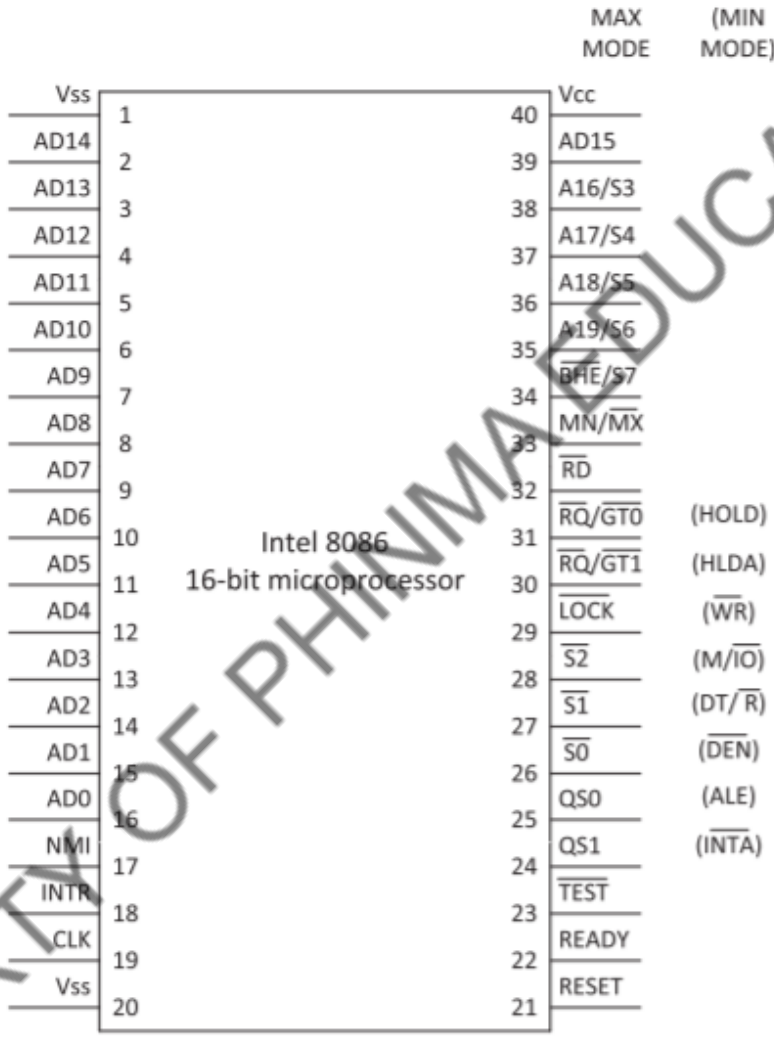


Figure 5.1
Intel 8086 16-bit microprocessor

5.1.1 Memory Organization

The 8086 processor has a 20-bit address bus, which gives a physical address space of up to 1 MB (220), addressed as 00000h to FFFFFh. However, the maximum linear address space was limited to 64 KB, simply because the internal registers are only 16 bits wide. A technique called “internal segmentation” has to be used for applications that are above the 64 KB boundary. There are four 16-bit segment registers (CS for “code segment,” DS for “data

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

segment," ES for "extra data segment," and SS for "stack segment"). All memory references are of the form [segment:offset], relative to the base address contained in the corresponding segment register.

In particular, given a [segment:offset] pair, a 20-bit external (or physical) address is produced by segment × 24 + offset, where segment × 24 is called the segment address, which has its 16 most significant bits from the 16-bit segment register, and its four LSbs are all zeros. A 16-bit offset is always added to the 20-bit segment address to yield an external address. As an example, consider the pair [000Ah:000Ch] (i.e., 10:12). The segment value of 000Ah (10) would give a segment address at 000A0h (160) in the linear address space. The address offset 000Ch can then be added to the segment address: 000A0h + 000Ch = 000ACh (160 + 12 = 172). Such address translations are carried out automatically by the segmentation unit of the processor.

Owing to the 4-bit shift, each segment virtually begins at a multiple of 16 bytes, from the beginning of the 1 MB address space. Particularly, the last segment, FFFFh, begins at linear address FFFF0h, 16 bytes before the end of the 20-bit address space. Since all segments are 64 KB long (with wraparound at the high end) and they increment by 16 bytes, there are huge overlaps among segments, and each location in the linear memory address space can be accessed by 216/24 = 212 = 4096 different [segment:offset] pairs. For example, the memory location 00100h can be referred to by [0000h:0100h], [0001h:00F0h], [0002h:00E0h], [0003h:00D0h], etc. Although complicated, this scheme has its advantages. For instance, a small program (less than 64 KB) can be loaded starting at a fixed offset in its own segment, avoiding the need for relocation, with at most 15 bytes of alignment waste.

Locations from address FFFF0h to address FFFFFh are reserved. Following reset, the CPU will always begin execution at location FFFF0h, which is a jump to the initial program loading routine. The block from 00000h to 003FFh is reserved for the interrupt table, where each of the 256 possible interrupt types has its service routine pointed to by a 4-byte pointer element consisting of a 16-bit segment address and a 16-bit offset address.

The memory is physically organized as a high bank (D15-D8) and a low bank (D7-D0) of 512 KB addressed in parallel by the processor's address lines A19-A1. The processor provides two enable signals, $\overline{BHE}$ and A0, to selectively allow reading from or writing into either an odd byte location or an even byte location, or both. Byte data with an even address (A0 is low) are transferred on the D7-D0 bus lines, and byte data with an odd address (A0 is high) are transferred on the D15-D8 bus lines

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

5.1.2 Separate I/O Address Space

An I/O device may be enabled, and the ports or registers on the device may be accessed, merely by the address lines (A19-A0). In such a case, the I/O devices are placed in the 8086 memory address space. As long as an I/O device responds like a memory chip, they are indistinguishable to the processor.

The 8086 processor also allows I/O devices to be placed in a separate I/O address space. This is achieved by the $\overline{S2}$ signal in maximum mode, and by the M/$\overline{IO}$ signal in minimum mode. M/$\overline{IO}$ becomes valid in the T4 preceding a bus cycle and remains valid until the final T4 of the cycle. The I/O address space is accessed by AD15-AD0 (the address lines A19-A16 are zero in I/O operations), which can maximally accommodate 64 KB registers or 32K word registers on I/O devices.

Figure 5.2 shows the wiring of a universal asynchronous receiver/transmitter (UART) which is placed in the I/O address space. The 8086 processor operates in minimum mode, because the MN/$\overline{MX}$ pin is driven high.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

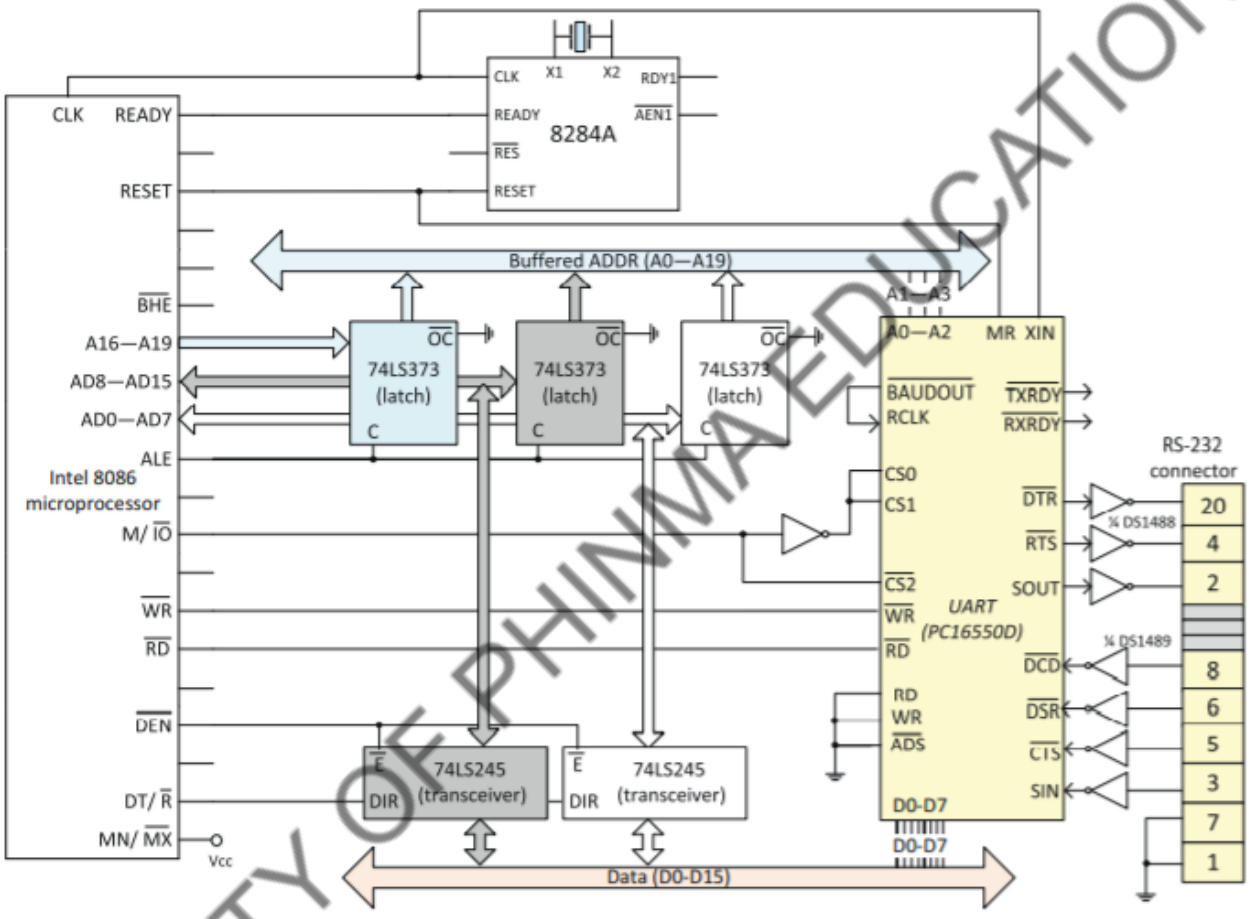


Figure 5.2
Intel 8086 memory I/O mapping.

5.1.2.1 Timing clock

The processor is connected to an external clock generator, 8284A. A crystal is attached to pins X1 and X2 of 8284A; CLK has an output frequency which is one third of the crystal frequency.

The processor gets its clock by connecting its CLK input to the 8284A's CLK output. 8284A also has an input pin $\overline{RES}$, which is used to generate a reset signal through the output pin RESET to reset the 8086 processor. The input pin RDY1 is a bus-ready signal; it is typically driven by memory or I/O devices, acknowledging that data have been received or are available on the data bus. RDY1 is validated by $\overline{AEN1}$, which is connected to the ground when RDY1 is in

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

use. The input signal RDY1 is synchronized by the 8284A clock generator to generate an output signal READY, which in turn is connected to the READY input of the 8086 processor, indicating the completion of data transfer.

5.1.2.2 External bus

Since AD15-AD0 are multiplexed to carry both data and address information, three 8-bit 74LS373 chips are used to implement an address latch. The address latch enable signal ALE is active (high) during T1 of any bus cycle. ALE is used to enable the three 74LS373 chips so that the address information is captured by the address latch during T1 and remains available on the buffered bus (A19-A0) until the next time ALE transits to high.

AD15-AD0 and the external data bus D15-D0 are connected through a bus transmitter/receiver implemented by two 8-bit 74LS245 chips. The direction input DIR controls transmission of data: from AD15-AD0 to D15-D0 when it is high (say, in a write operation), and from D15-D0 to AD15-AD0 when it is low (say, in a read operation). DIR is driven by the data transmit/receive signal $DT/\overline{R}$, which is valid in the T4 preceding a bus cycle and remains valid until the final T4 of the bus cycle.

The 74LS245 chip has an enable input $\overline{E}$; the two buses are isolated when $\overline{E}$ is high. $\overline{E}$ is driven by the data enable output signal $\overline{DEN}$, which, for a read or interrupt acknowledge cycle, is active from the middle of T2 until the middle of T4, while for a write cycle it is active from the beginning of T2 until the middle of T4.

5.1.2.3 I/O device: UART

The processor is wired with a PC16550D UART chip for serial transmission of digital information (bits) through a single wire. The UART performs serial-to-parallel conversion on data characters received from a peripheral device or a modem, and parallel-to-serial conversion on data characters received from the processor.

PC16550D is mainly composed of a programmable baud rate generator, a transmitter, a receiver, and several registers (e.g., 16-bit divisor register) that control UART operations, including transmission and reception of data. Note that the UART chip has three address signals (A0, A1, A2) and eight data signals (D7-D0). Refer to its data sheet [9] for other pin descriptions.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

In Figure 5.2, pins $\overline{RD}$ and $\overline{WR}$ of the UART are connected to the corresponding pins of the processor. This allows the processor to transmit data to, and receive data from, the UART. Also, the UART chip is selected only when the M/$\overline{IO}$ signal is low; this indicates that the registers of the UART are mapped into the I/O address space rather than the memory space.

5.1.3 Memory Address Space

Figure 5.3 shows a configuration where the Intel 2142 memory chips are mapped into the memory address space. This is because the memory chips are readable or writable only when M/$\overline{IO}$ is high. Specifically, the write enable signal $\overline{WE}$ of the memory chips is from $\overline{MEMW}$, which is asserted when M/$\overline{IO}$ is high and $\overline{WR}$ is low, where $\overline{WR}$ indicates that the processor is performing a write cycle. Similarly, the output disable signal $\overline{OD}$ of the memory chips is from $\overline{MEMR}$, which is asserted low when M/$\overline{IO}$ is high and $\overline{RD}$ is low, where $\overline{RD}$ indicates that the processor is performing a read cycle.

A 74LS138 1-of-8 decoder/demultiplexer is used to produce chip selection signals. Its truth table is given in Table 5.3. Since the output signal $\overline{O0}$ is used in Figure 5.3, the memory chips are enabled only when [A18A17A16] = [000].

The 8086 processor is wired with a high memory bank and a low memory bank, each composed of two Intel 2142 chips. A 2142 chip is a 1024 × 4-bit SRAM. The chip select signal $\overline{CS}$ of the high bank is asserted (active low) whenever $\overline{O0}$ and $\overline{BHE}$ are low. In such a case, data are written to or read from the high bank through bus lines D8-D15. The chip select signal $\overline{CS}$ of the low bank is asserted (active low) whenever $\overline{O0}$ and A0 are low. In such a case, data are written to or read from the low bank through bus lines D0-D7. In other words, byte data with an even address (A0 = 0) are read from or written to the low memory bank through D0-D7, while byte data with an odd address ($\overline{BHE}$ is asserted as A0 = 1) are read from or written to the high memory bank through D8-D15.

A0 is also called a bank selection signal since it is reserved for that purpose. Care must be taken to ensure that each register within an 8-bit peripheral device located on the lower portion of the data bus be addressed as even. Now, you should understand why in Figure 5.2 it is address lines A1-A3, rather than A0-A2, that are connected to pins A0-A2 of the UART. If A0 of the address bus were used, some of the UART registers would become inaccessible because the UART is wired only to the low portion of the data bus.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Table 5.3 - 74LS138 truth table

A	B	C	$\overline{O_0}$	$\overline{O_1}$	$\overline{O_2}$	$\overline{O_3}$	$\overline{O_4}$	$\overline{O_5}$	$\overline{O_6}$	$\overline{O_7}$
0	0	0	L	H	H	H	H	H	H	H
0	0	1	H	L	H	H	H	H	H	H
0	1	0	H	H	L	H	H	H	H	H
0	1	1	H	H	H	L	H	H	H	H
1	0	0	H	H	H	H	L	H	H	H
1	0	1	H	H	H	H	H	L	H	H
1	1	0	H	H	H	H	H	H	L	H
1	1	1	H	H	H	H	H	H	H	L

5.1.4 Wait States

Normally, a bus cycle (e.g., accessing a memory or I/O port) on an 8086 processor is composed of four clock cycles—T1 to T4. Wait states, called Tw, can be inserted in a bus cycle to help the processor to interface with slow memory or I/O devices. The READY input signal on the 8086 is used to request the processor to stretch out its read or write cycle, to accommodate slow devices. In particular, the 8086 processor samples the READY line at the rising edge of T3. If READY is low, a Tw state is inserted after T3. A "wait" state is of the same duration as a clock cycle. During the Tw state, the READY is sampled again at the next rising edge of the clock, and another Tw is inserted if READY is still low. Whenever READY is sampled high at the rising edge of T3 or Tw, the T4 clock cycle follows.

In Figure 5.3, the processor's READY pin is driven by the READY output of the 8284A clock generator, which, in turn, is synchronized with its RDY1 input. A memory or I/O device can initiate WAIT state generation by bringing RDY1 low.

74LS164 used in Figure 5.3 is an 8-bit shift register, where the two serial inputs A and B are connected to voltage (steady high). When the clear signal $\overline{CLR}$ is asserted, all the outputs (QA, ..., QH) become low. When $\overline{CLR}$ is not active, QA becomes high on the low-to-high transition of the clock input. The outputs QB, ..., QH, respectively, take the level of QA, ..., QG before the most recent low-to-high transition of the clock, indicating a 1-bit shift every clock.

Now, let us examine the time diagram in Figure 5.4 to see how the 74LS164 configuration given in Figure 5.3 works.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

First, note that the bus cycle begins in T1 with the assertion of the address latch enable signal ALE. The trailing edge of this signal is used to latch the address information, which is valid on the buffered address bus at this time. At T2 the address is removed from the local bus, which goes to a high impedance state.

The 74LS164 shift register is cleared whenever $\overline{\text{INTA}}$, $\overline{\text{RD}}$, and $\overline{\text{WR}}$ are all high. In Figure 5.4 this is indicated by the two shaded areas bounded by the level changes of $\overline{\text{RD}}$.

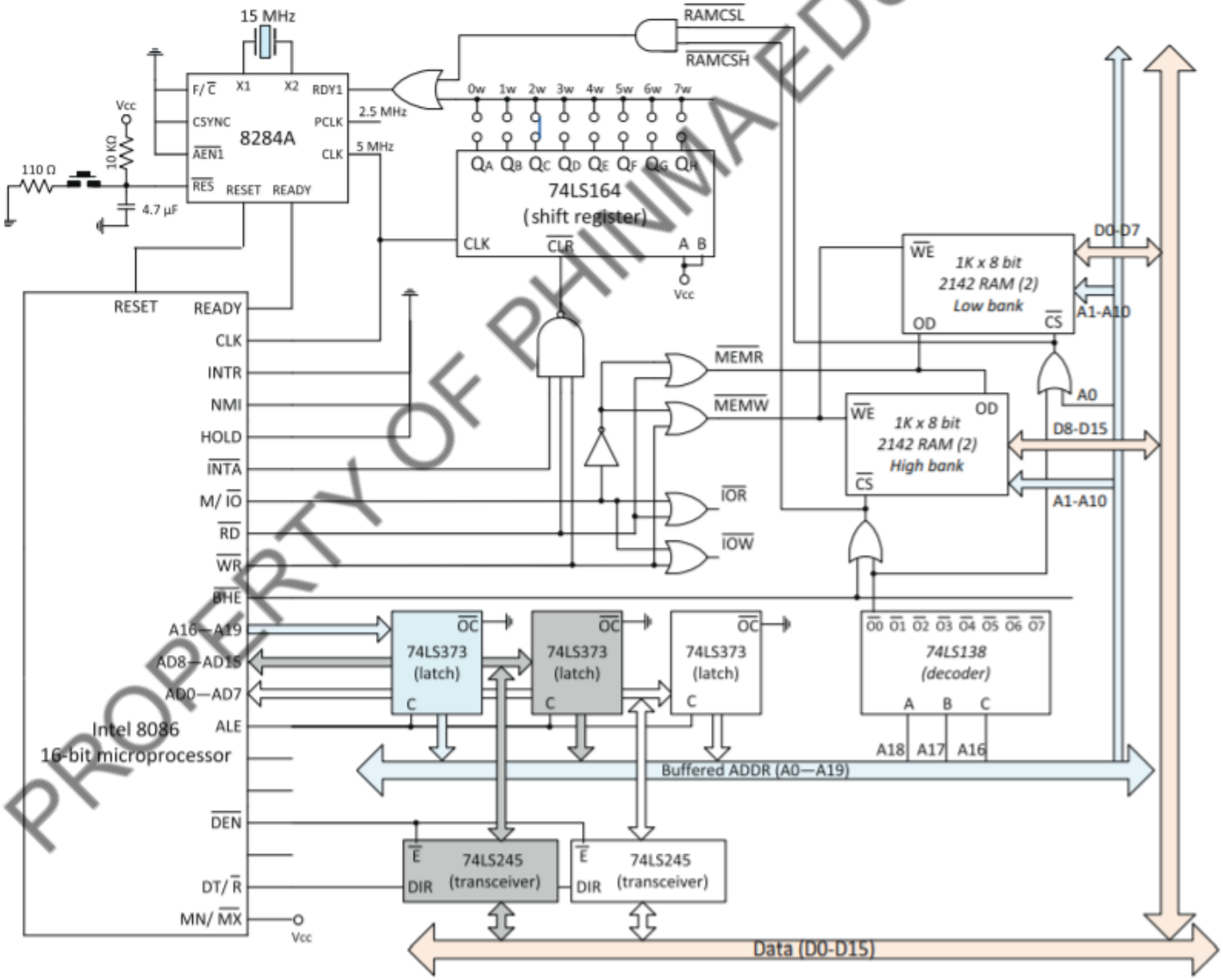


Figure 5.3 - Intel 8086 memory mapping.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The read control signal $\overline{RD}$ is asserted at T2. Whenever $\overline{INTA}$, $\overline{RD}$, or $\overline{WR}$ is asserted, the $\overline{CLR}$ input of the 74LS164 shift register is driven high, and the 74LS164 shift register becomes ready to drive its outputs. The 74LS164 shift register drives QA high when it comes to the rising edge of the next clock, which is the T2 cycle. QA remains high until the 74LS164 shift register is cleared. One clock cycle later, QB is driven high, taking the level of QA; one more clock cycle later, QC is driven high, taking the level of QB.

To ensure that the 8086 timing requirements are met, the device needs to bring RDY1 low prior to the rising edge of the 8086's T2 clock. As indicated in Figure 5.4, both RAMCSH and RAMCSL, which are simply the select signals of the memory chips, are low in clock cycle T1.

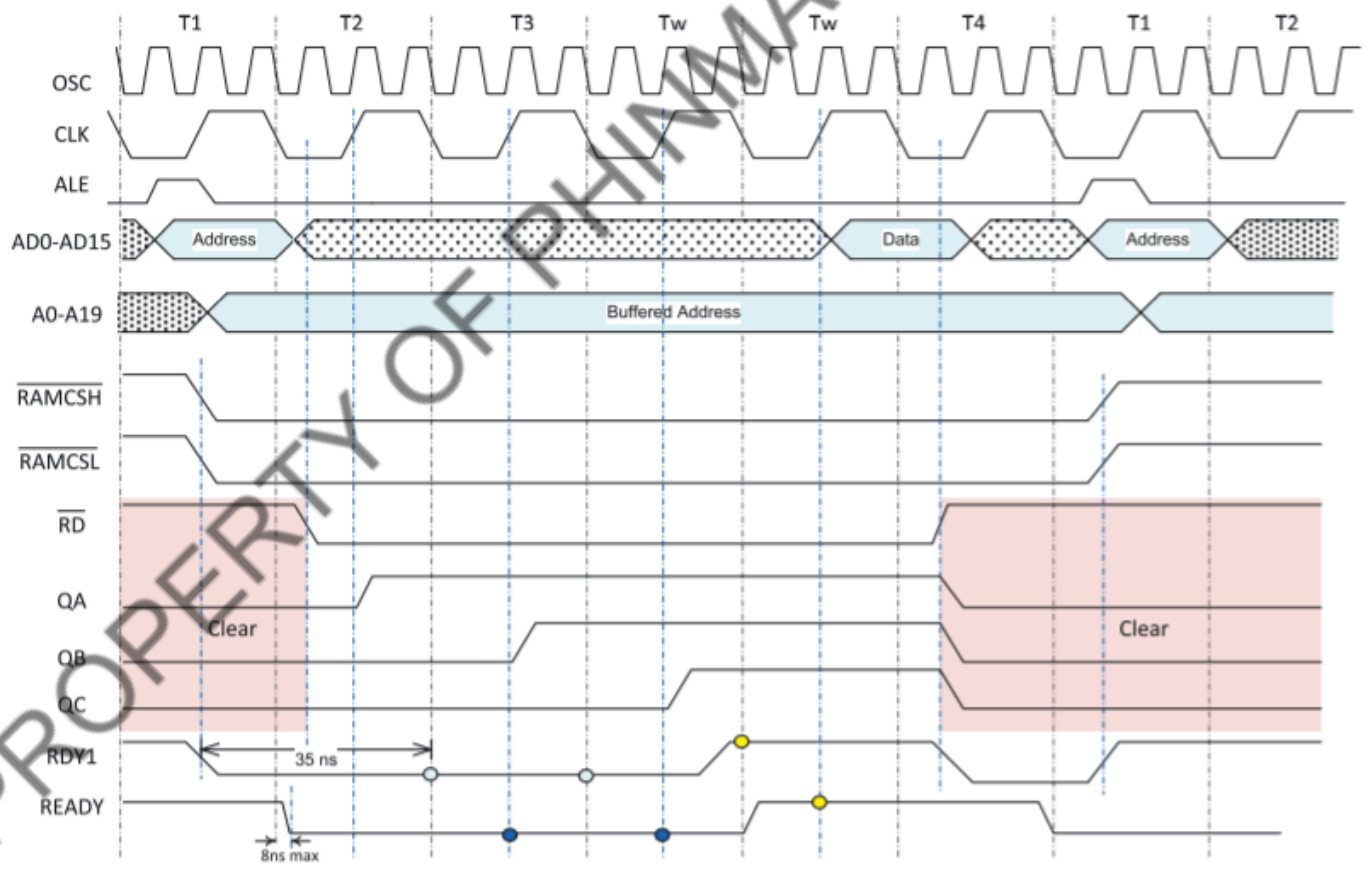


Figure 5.4
Intel 8086 timing diagram.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Since QC of the 74LS164 shift register is also low in cycle T1, RDY1 is driven low, which indicates the "not ready" status of the memory devices.

The 8284A clock generator drives the 8086's READY signal low at the falling edge of T1. The 8086 processor samples the READY line at the rising edge of T3. In our case, it finds that READY is low, and it inserts a Tw clock cycle after T3. The 8086 processor samples the READY line again at the rising edge of Tw, and inserts another Tw clock cycle after the first Tw.

Another timing constraint is that whenever a new Tw cycle is no longer needed, the memory device has to bring RDY1 high early in T3 or Tw so that the 8284A clock generator can bring READY high before the rising edge of T3 or Tw. In order to bring RDY1 high, the 74LS164 shift register has to drive QC high because only QC is wired to the OR gate. QC is driven high at the rising edge of the first Tw, which is two clock cycles later than when QA changed to high. The 8284A clock generator then starts to drive the 8086's READY signal high at the falling edge of the first Tw. This happens before the 8086 processor samples the READY line at the rising edge of the second Tw. As a result, no more Tw cycle is added before the T4 clock cycle.

5.2 Intel Pentium

The Intel Pentium series is the fifth generation of the x86 microprocessor family. The Pentium series is positioned as Intel's mid-range processor series, in between the Celeron and Core series.

The Pentium processor is a 32-bit processor with a 64-bit data bus. Its pin diagram is given in Figure 5.5, where the pins are grouped according to their functions.

The Pentium processor can initiate the following bus cycles:
- Single-transfer cycle: the transfer of one data item (up to 8 bytes or 64 bits).
- Interrupt acknowledge cycle: The Pentium processor has 256 possible interrupt vectors. It generates interrupt acknowledge cycles in response to maskable interrupt requests.
- Special bus cycle: It is generated in response to certain instructions (e.g., the halt instruction is executed).
- Read cycle: It is generated by the processor to read data (or code) from memory or I/O devices.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

- Write cycle: It is generated by the processor to write data to memory or I/O devices.
- Burst-transfer cycle: the transfer of four data items (up to 32 bytes or 256 bits).
  - Burst read cycle: 32 bytes are read consecutively from memory into the internal cache in one bus cycle.
  - Burst writeback cycle: Modified lines in the processor's data cache are written back to memory in blocks of 32 bytes.

The pins that are pertinent to the bus operations are described below (interested readers can refer to the data sheet [3] to get more information):

- A31-A3: Address lines. As outputs, A31-A3 along with $\overline{BE7}$-$\overline{BE0}$ define the physical address space (memory or I/O).

- $\overline{ADS}$: Output signal for address status. When asserted, it indicates that a new valid bus cycle is currently being driven by the processor.

- AHOLD: Input signal for address hold. An external system can initiate an inquire cycle to a Pentium processor to check whether a particular address location is cached in the processor's internal cache, and if so, what state it is in. To prepare for an inquire cycle, an external system first needs to assert AHOLD to force the Pentium processor to float its address bus, then wait two clocks for the processor to finish housekeeping (so that data can be returned or driven for previously issued bus cycles).

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

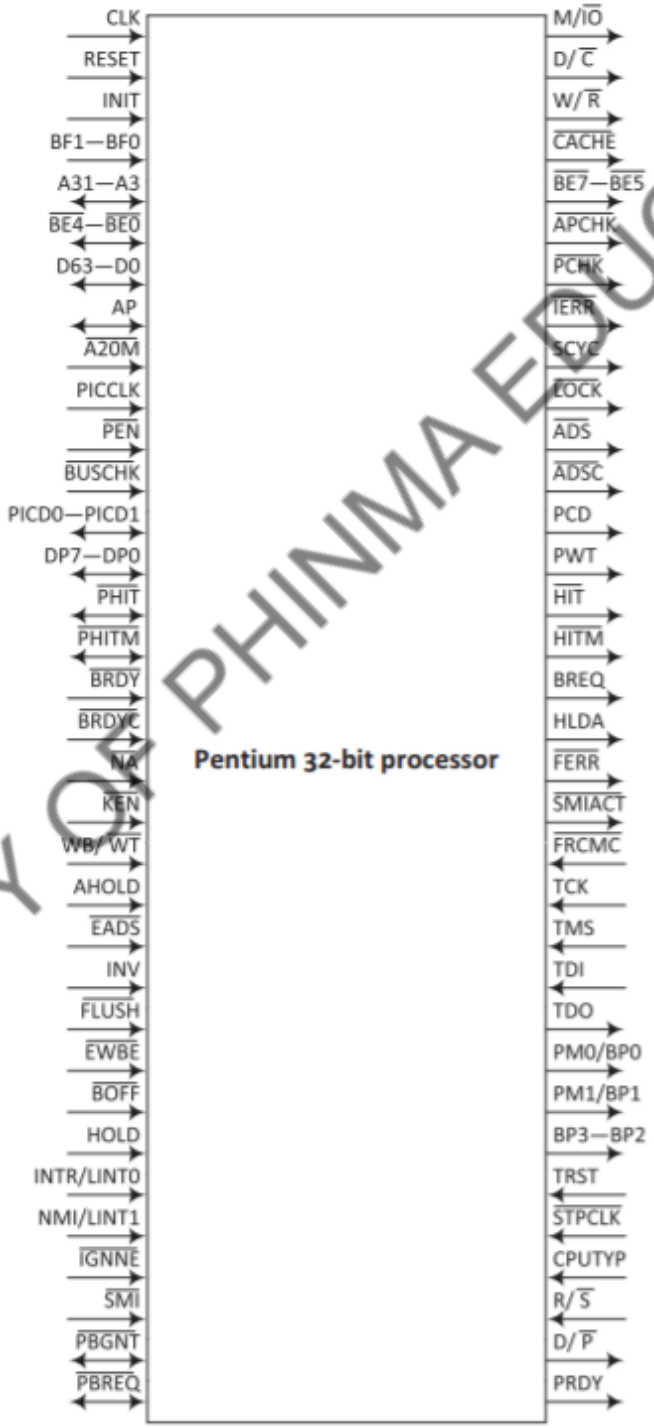


Figure 5.5 - Intel Pentium 32-bit processor

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• $\overline{BE7}$-$\overline{BE0}$: Byte enable signals. $\overline{BE7}$-$\overline{BE0}$ are driven in the same clock as address lines A31-A3, used to determine which bytes are transferred (read or written) by the processor for the current bus cycle.
• $\overline{BOFF}$: Input signal for backoff. It is used to abort all outstanding bus cycles that have not yet finished. When it is asserted low, the processor will float all pins in the next clock and maintain its bus in this state until $\overline{BOFF}$ is deasserted, at which time the processor will restart the aborted bus cycle(s).
• $\overline{BRDY}$: Input signal for burst ready. It indicates that the external system has presented valid data on the data pins in response to a read request or that the external system has accepted the processor's data in response to a write request.
• $\overline{CACHE}$: Output signal indicating the internal cacheability of a bus cycle.
• $\overline{KEN}$: Input signal for cache enable. When the processor generates a read cycle that can be cached ($\overline{CACHE}$ asserted) and $\overline{KEN}$ is active, the cycle will be transformed into a burst read cycle.
• D/$\overline{C}$: Output signal for data/code indication. It is driven valid in the same clock as the $\overline{ADS}$ signal is asserted. The current cycle is for data when D/$\overline{C}$ is high, and is for code or special bus cycles when D/$\overline{C}$ is low.
• D63-D0: Data lines. Lines D7-D0 define the LSB of the data bus; lines D63-D56 define the $\overline{MSB}$ of the data bus.
• EADS: Input signal for external address status. It indicates that an external system has initiated an inquire cycle to the Pentium processor, and a valid inquire address has been driven onto the processor's address pins by the external system.
• $\overline{HITM}$: Output signal for hit to a modified line. It is asserted to indicate to the external system that the cache of the processor contains the most current copy of the data and any device needing to read that data should wait for the processor to write it back. $\overline{HITM}$ remains asserted until two clocks after the last $\overline{BRDY}$ of the writeback cycle is asserted.
• HOLD: Input signal for bus hold request. When it is asserted by another local bus master, the processor will float most of its output pins. After completing all the outstanding bus cycles, the processor will also assert HLDA (bus hold acknowledge) to relinquish the bus to the requester. The processor will maintain its bus in this "bus-hold" state until HOLD is deasserted. In such a case, the processor will deassert HLDA and resume driving the bus.
• $\overline{NA}$: Input signal for next address. When $\overline{NA}$ is asserted, it indicates that the external memory system is ready to accept a new bus cycle although all data transfers for the current cycle have not yet finished. The processor will issue $\overline{ADS}$ for a pending cycle two clocks after $\overline{NA}$ is asserted.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• RESET: It forces the Pentium processor to restart execution at a known state.
Microprocessor

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

5.2.1 Bus State Transition

We are now ready to examine the state transitions of Pentium bus cycles. A bus cycle begins with the processor driving an address and status and asserting ADS, and ends when the last BRDY is returned.

The Pentium processor supports up to two outstanding bus cycles. Each bus cycle is composed of several states (clock cycles). Below is a list of valid bus states for the Pentium processor:

- Ti: This is the bus idle state. An asserted BOFF or RESET will always force the bus back to this state. HLDA will be driven only in this state.
- T1: there is only one outstanding bus cycle, and this is the first clock of that bus cycle. Valid address and status are driven out and ADS is asserted.
- T2: There is only one outstanding bus cycle, and this is the second and subsequent clock of the bus cycle. In state T2, data are driven out (if the cycle is a write cycle), or data are expected (if the cycle is a read cycle), and the BRDY pin is sampled. A bus cycle may have multiple T2 states.
- T12: There are two outstanding bus cycles. While the second (newer one) bus cycle is in its T1 state the first bus cycle is in its T2 state. This implies that (a) for the second outstanding bus cycle, the processor drives the address and status and asserts ADS, and (b) for the first outstanding cycle, data are transferred and BRDY is sampled.
- T2P: There are two outstanding bus cycles, and both are in their respective T2 states. In T2P, data are being transferred and BRDY is sampled for the first outstanding cycle.
- TD: There is one outstanding bus cycle. Write cycles can be pipelined into read cycles and read cycles can be pipelined into write cycles, but one dead clock is required between consecutive read and write cycles to allow bus turnover. The processor enters TD if (a) a dead clock is needed, and (b) in the previous clock there were two outstanding cycles, and BRDY was sampled active (low) for the first bus cycle.

The possible bus state transitions are described in detail in the UML state diagram given in Figure 5.6.

First, the notion of request pending, ReqPend, is defined as a Boolean expression:

$$\text{ReqPend} \triangleq P \wedge \neg \text{HOLD} \wedge \overline{\text{BOFF}} \wedge (\neg \text{AHOLD} \vee \neg \overline{\text{HITM}}),$$

where the proposition P states that the processor has generated a new bus cycle internally. Note that the term BOFF says that BOFF is not asserted (i.e., BOFF=1).

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

A composite state named “Active” is used to encapsulate the nonidle states. The bus stays within the idle state Ti if there is no request pending. Whenever the processor has generated a new bus cycle (ReqPend is true), in the next clock (i.e., as the current clock cycle expires) the bus transits from Ti to T1. The bus will transit back to Ti upon the next clock if:

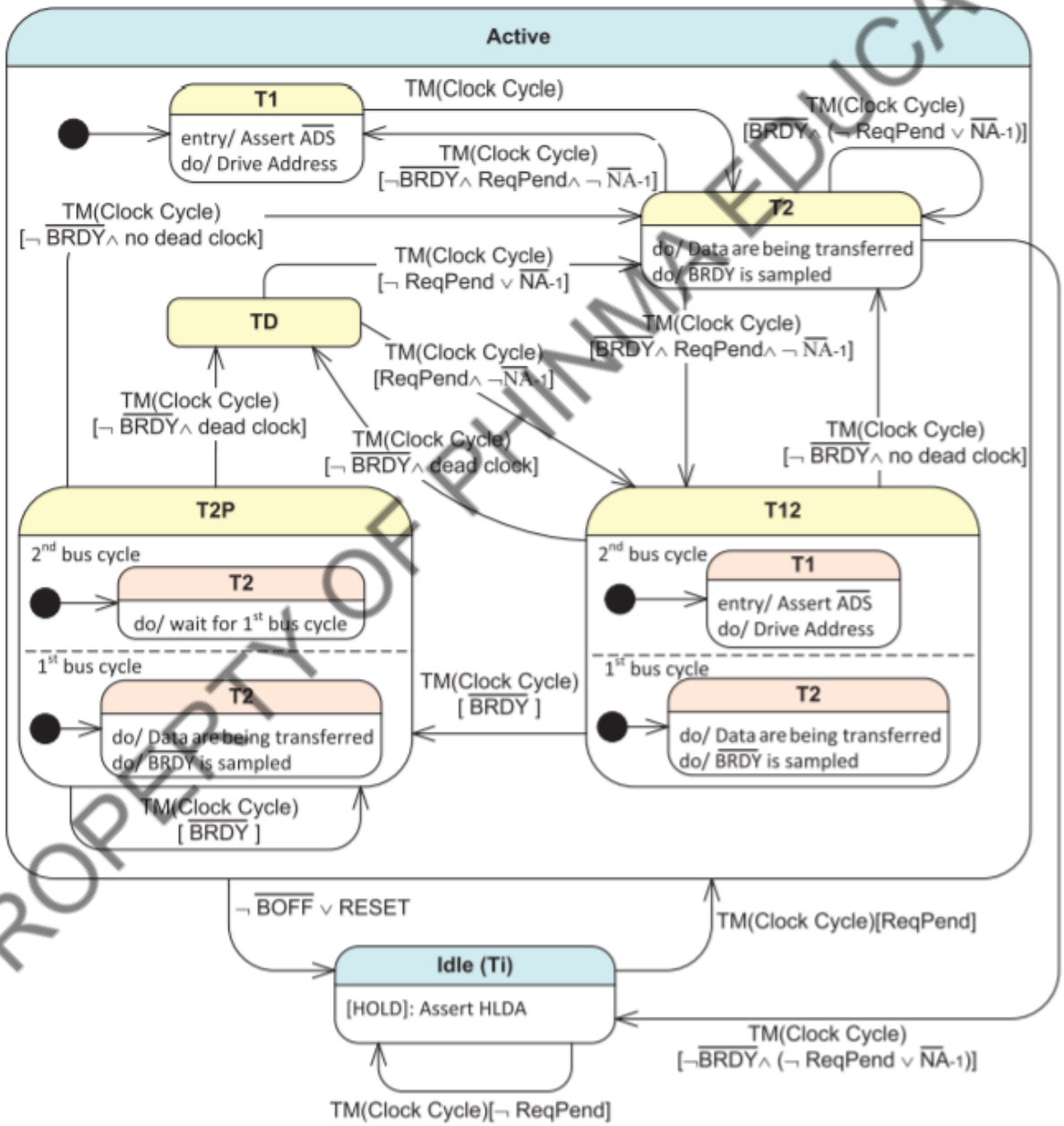


Figure 5.6 - Pentium bus control state machine.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• in the current clock, $\overline{BOFF}$ is asserted (low) or RESET is asserted (high); or
• the current clock is T2, the data transfer has finished ($\overline{BRDY}$ was sampled low), and there is no request pending or $\overline{NA}$−1 is high (not asserted). Note here that $\overline{NA}$−1 represents the logical level of $\overline{NA}$ in the clock before this T2.
In a well-designed state diagram, if a state has multiple outgoing transitions triggered by the same event, their guard conditions—say, C1, C2, … , Ck—ought to be able to form a complete Boolean expression; that is, C1 ∨ C2 ∨⋯∨ Ck = 1. The state diagram in Figure 5.6 is well designed for the following reasons:

• The guard conditions of the four transitions leaving state T2 are complete. That is,

$$[\overline{BRDY} \wedge (\neg ReqPend \vee \overline{NA}-1)] \vee [\overline{BRDY} \wedge ReqPend \wedge \neg\overline{NA}-1)] \vee [\neg\overline{BRDY} \wedge$$

$$(\neg ReqPend \vee \overline{NA}-1)] \vee [\neg\overline{BRDY} \wedge ReqPend \wedge \neg\overline{NA}-1)] = 1.$$

• The guard conditions of the three transitions leaving state T12 are complete.
• The guard conditions of the four transitions leaving state T2P are complete.
• The guard conditions of the two transitions leaving state TD are complete.

Also note that both T12 and T2P contain two parallel sessions, owing to the existence of two outstanding bus cycles. In T2P, the second bus cycle is waiting for the first bus cycle to finish its data transfer.

Let us use some timing diagrams to examine how the bus changes its states as the timing clock goes on.

Figure 5.7 shows a read cycle followed by an idle state, then a write cycle. Notice that $\overline{ADS}$ and W/R are asserted in clock T1; address lines are also driven in clock T1. $\overline{BRDY}$ is sampled in clock T2 to check whether data transfer has finished or not. The transition from T2 to Ti happens because the guard condition $\neg\overline{BRDY} \wedge (\neg ReqPend \vee \overline{NA}-1)$ is true.

Figure 5.8 shows a read cycle followed by an idle state, then a write cycle. Notice that the read cycle takes two T2 clocks and the write cycle takes three T2 clocks. This can happen when the processor interfaces with a slow memory device.

Figure 5.9 shows a burst read cycle. Notice that $\overline{CACHE}$ is driven active (low) in clock T1. The read cycle is transformed into a multiple-transfer burst cycle by $\overline{KEN}$ being asserted active on

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

the clock on which the first active $\overline{BRDY}$ is sampled. Consequently, the processor is able to read one data item (i.e., 8 bytes or 64 bits) in each T2 clock. It may take more than one T2 clock to read one data item when the processor interfaces with slow memory.

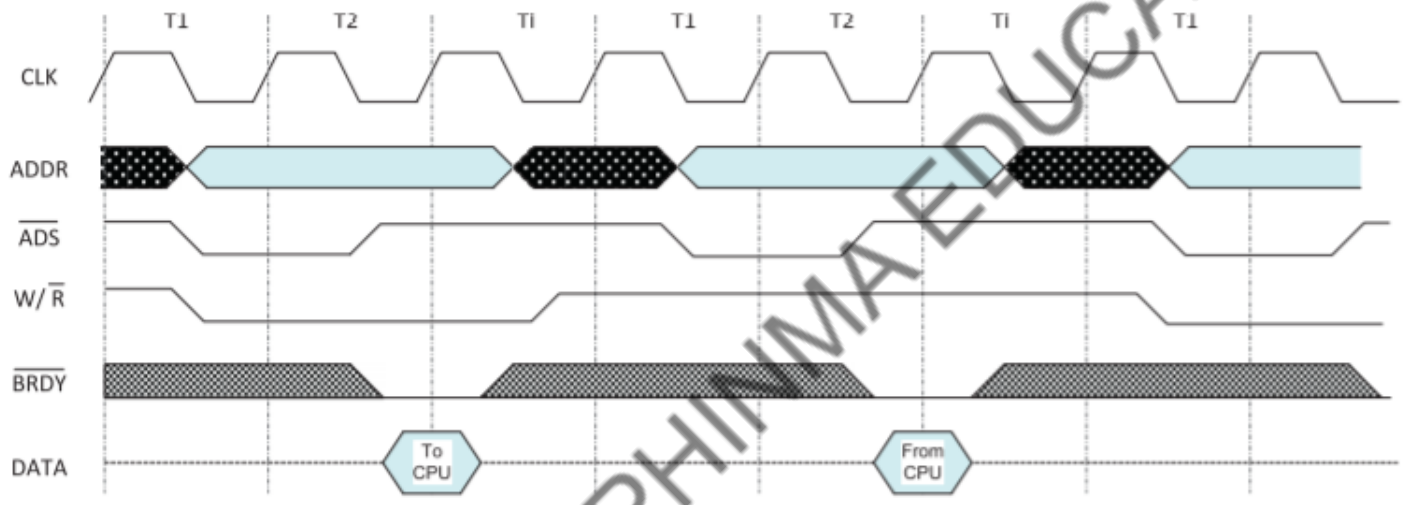


Figure 5.7
Pentium timing diagram example.

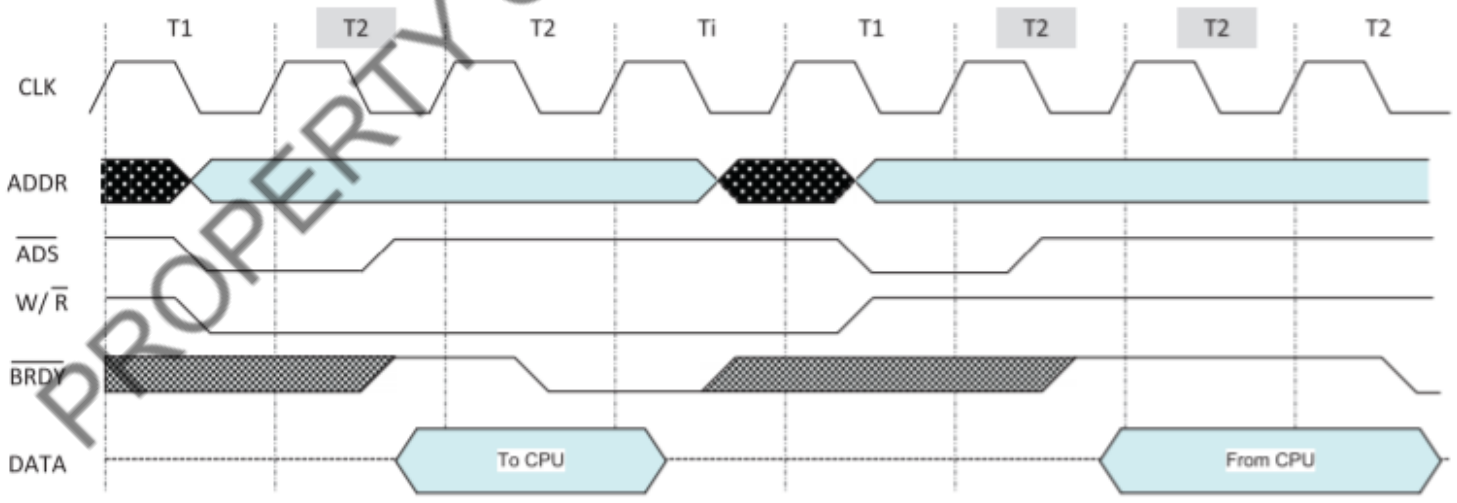


Figure 5.8
Pentium timing diagram: wait states.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

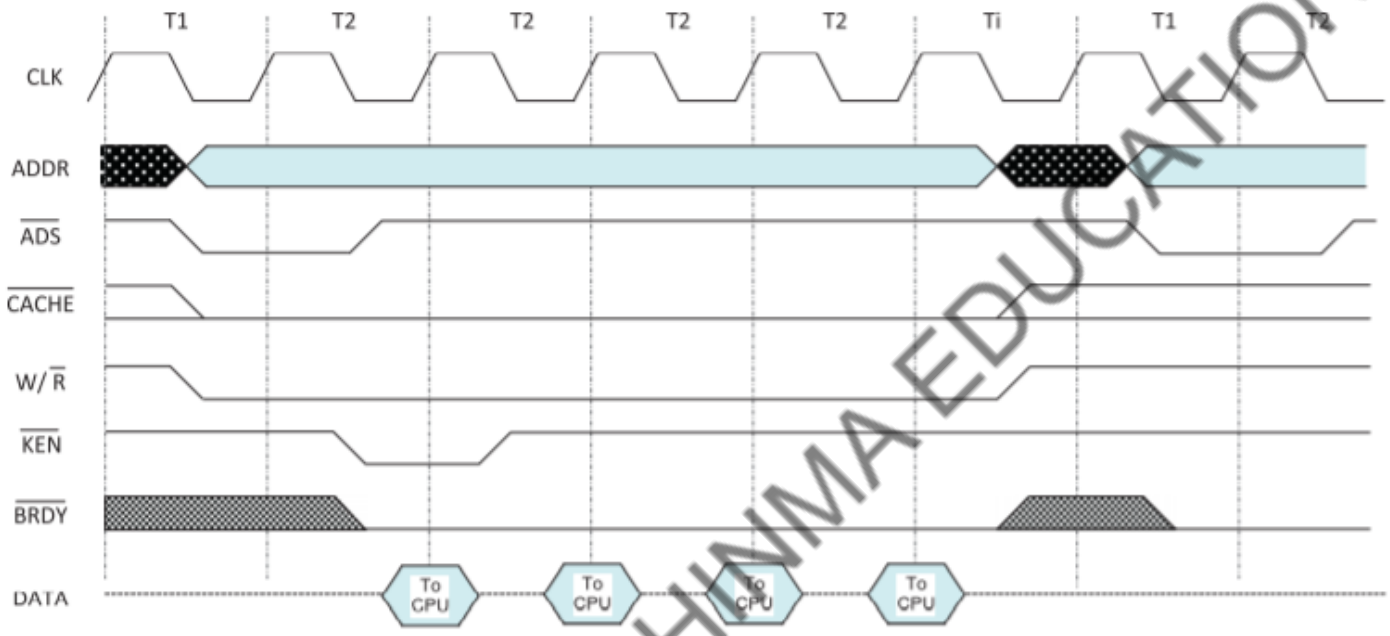


Figure 5.9
Pentium timing diagram: burst read.

Figure 5.10 shows the pipelining of a burst read cycle and a write cycle. The first bus cycle starts as a read cycle. BRDY is sampled high in the first T2 clock, indicating that another T2 clock is needed to finish the data transfer. BRDY is sampled active in the second T2 clock, indicating that one data item has been transferred. However, KEN is asserted in the second T2 clock, which transforms the read cycle into a burst read cycle.

Also notice that NA is asserted in the second T2 clock, indicating that the external memory system is ready to accept a new bus cycle although all data transfers for the current read cycle have not yet finished. The processor will issue ADS for a pending cycle two clocks later.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

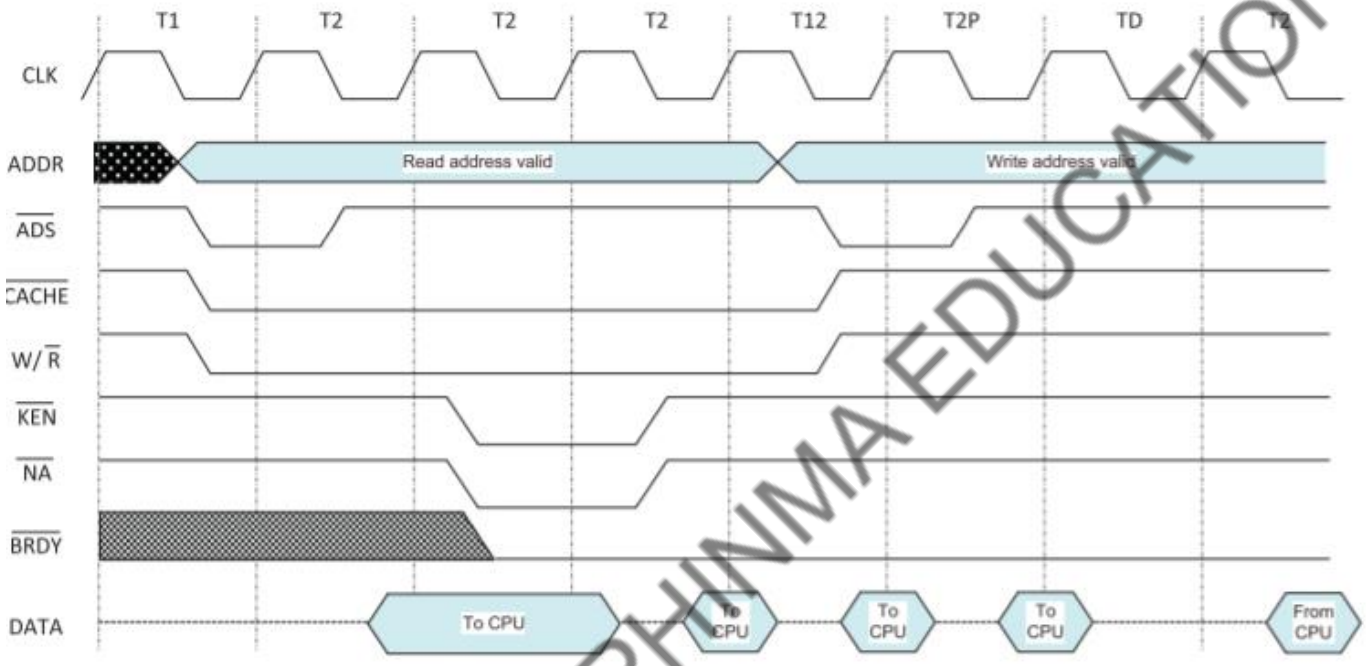


Figure 5.10
Pentium timing diagram: pipelining

$\overline{\text{BRDY}}$ is sampled low in the third T2 clock, indicating that another data item has been transferred. Also, the assertion of $\overline{\text{NA}}$ in the second T2 clock makes ¬$\overline{\text{NA}}$−1 true in the third T2 clock, which enables the transition from state T2 to state T12.

In the T12 clock, $\overline{\text{BRDY}}$ is sampled low, indicating that another data item has been transferred for the burst read cycle. During the same clock, the processor initiates a new write cycle, driving address, status, and $\overline{\text{ADS}}$ for this second outstanding bus cycle.

In the T2P clock, $\overline{\text{BRDY}}$ is sampled low, indicating that the fourth data item has been transferred for the burst read cycle. Up to this point, the burst read cycle has finished. Upon the next clock, the bus transits from the T2P state to the TD state, because in pipelining a dead clock is needed between read and write cycles. After the TD clock, the bus is ready for transferring data for the write cycle.

5.2.1 Memory Organization

The Pentium processor has a memory space of 4 GB (2^{32} bytes) and a separate I/O space with

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

64 KB of addressable locations. The memory space is organized as a sequence of 64-bit quantities. Each 64-bit location has eight individually addressable bytes at consecutive memory addresses. The I/O space is organized as a sequence of 32-bit quantities. Each 32-bit quantity has four individually addressable bytes at consecutive memory addresses.

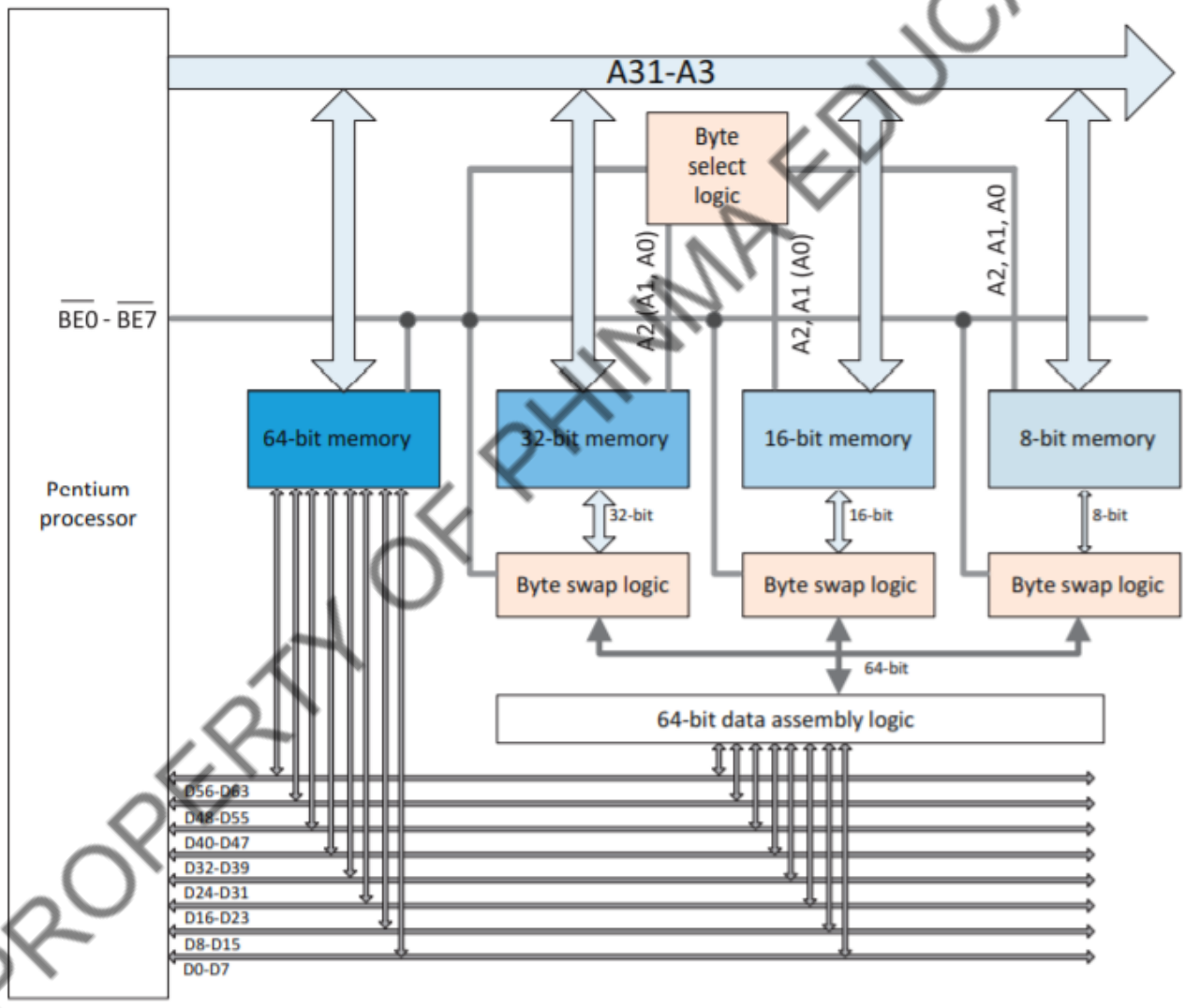


Figure 5.11
Pentium memory mapping.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Figure 5.11 illustrates how the Pentium processor interfaces with memory of various width:

- 64-bit memories are organized as arrays of quadwords (a quadword is a block of 8 bytes). Quadwords begin at addresses evenly divisible by 8. Address lines A31-A3 are used to access quadwords, and $\overline{BE7}$-$\overline{BE0}$ are used to access individual bytes within a quadword.
- 32-bit memories are organized as arrays of dwords (a dword is a block of 4 bytes). Dwords begin at addresses evenly divisible by 4. A dword can be accessed by address lines A31-A3 together with A2. A2 is not a physical address line, but is decoded from BE7-BE0 according to Table 5.4:

$$A2 = \overline{BE0} \wedge \overline{BE1} \wedge \overline{BE2} \wedge \overline{BE3} \wedge (\neg\overline{BE4} \vee \neg\overline{BE5} \vee \neg\overline{BE6} \vee \neg\overline{BE7}).$$

Individual bytes within a dword can be accessed by A1 and A0, which are also decoded from $\overline{BE7}$-$\overline{BE0}$ according to Table 5.4.

Table 5.4 Byte access enabled by BE7-BE0

A2	A1	A0	$\overline{BE7}$	$\overline{BE6}$	$\overline{BE5}$	$\overline{BE4}$	$\overline{BE3}$	$\overline{BE2}$	$\overline{BE1}$	$\overline{BE0}$
0	0	0	X	X	X	X	X	X	X	L
0	0	1	X	X	X	X	X	X	L	H
0	1	0	X	X	X	X	X	L	H	H
0	1	1	X	X	X	X	L	H	H	H
1	0	0	X	X	X	L	H	H	H	H
1	0	1	X	X	L	H	H	H	H	H
1	1	0	X	L	H	H	H	H	H	H
1	1	1	L	H	H	H	H	H	H	H

- 16-bit memories are organized as arrays of words (a word is a block of 2 bytes). Words begin at addresses evenly divisible by 2. A word can be accessed by address lines A31-A3 together with A2 and A1. A0 is used to access individual bytes within a word.
- Eight-bit memories are organized as array bytes. A byte can be accessed by address lines A31-A3 together with A2, A1, and A0.

Note that in Figure 5.11 external byte swapping logic and data assembly logic are needed for memory widths smaller than 64 bits so that data can be supplied to and received from the Pentium processor on the correct data pins.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

5. 1 State an example of an embedded system that uses Intel 8086. Explain the function.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Intel 8086?	
	2 Is Intel Pentium same as Intel 8086? Why?	

3) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It provides the basic timing for the processor and bus controller.
_____2. It is a level-triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation.
_____3. How many bits of address bus does an Intel 8086 has?
_____4. It performs serial-to-parallel conversion on data characters received from a peripheral device or a modem, and parallel-to-serial conversion on data characters received from the processor.
_____5. It is the fifth generation of the x86 microprocessor family. It is a 32-bit processor with a 64-bit data bus.

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. How many hardware modes does an Intel 8086 support?

Intel 8086 supports two hardware modes (hard wired into the circuit and unchangeable by software): maximum mode and minimum mode. In maximum mode pin 33 (MN/MX) is wired to ground, and in minimum mode pin 33 is wired to voltage. Changing the state of pin 33 also changes the function of some other pins. The minimum mode is intended for small single processor systems, while the maximum mode is for medium-sized or large systems that use more than one processor (e.g., 8087 coprocessor in an IBM PC).

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #6

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

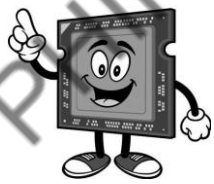
Lesson title: Interrupts
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand what are Interrupts;**
2. **Identify the different parts of Interrupts.**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"The key is not to prioritize what's on your schedule, but to schedule your priorities." So, gear up and get this priority done!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about the first part of Microprocessor Primer which was focused particularly on Intel 8086 and Intel Pentium.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Interrupts?	
	2 What are hardware interrupts?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

6.1 Introduction to Interrupts

Interrupt is a mechanism by which a microprocessor can alter its flow of execution to handle asynchronous or synchronous events. Asynchronous events are those that can occur at any time and typically occur at unanticipated spots of the running program, while synchronous events are those that can occur only at planned or anticipated spots of the running program.

Interrupts can be classified into three categories: external interrupts, software interrupts, and internal interrupts:

- External interrupts, also called hardware interrupts, are asynchronous events generated by external hardware devices to get the microprocessor's attention.
- Software interrupts, also called traps, are synchronous events generated by special processor instructions placed in a program. Software interrupts are unconditional in the sense that the execution of the special instruction will always generate a software interrupt.
- Internal interrupts, also called exceptions, are synchronous events generated by the processor itself whenever some abnormal condition occurs during instruction execution. Internal interrupts are conditional in the sense that the execution of some valid instruction (e.g., the division instruction) may cause an exception (e.g., if the divisor is 0).

6.2 External Interrupts

I/O devices are the liaison between an embedded system and its work environment. There are mainly two approaches for an embedded system to interact with I/O devices: polling and interrupts.

Polling is the simplest approach used in embedded systems to handle I/O activities synchronously. Assume that a peripheral I/O device intermittently receives data, which must be processed by the processor. In polling, the processor needs to continuously (or at least regularly) check if data have arrived. The processor typically does nothing other than check the status register of the I/O device until it is ready, at which point the device is accessed and serviced.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Polling is relatively straightforward in design and programming. In a simple system with only one I/O device, polling is perfectly appropriate because the system can just wait until the I/O device becomes ready for a service. Waiting for a device to get ready is no longer appropriate in systems involving multiple peripherals. In such a case, the polling mechanism could be improved such that the readiness of all the I/O devices is sequentially checked in an endless loop. When a device is not ready, instead of passively waiting until it becomes ready, the processor turns to check (and service, if possible) the next device.

When there are too many I/O devices to check, the time required to poll them can be considerable, and the system might break the deadlines of certain tasks. In such a case, the so-called interrupt-driven I/O comes into play.

Interrupt is a commonly used mechanism for computer multitasking, especially in real-time computing. Hardware interrupts are events generated by external hardware devices to get the microprocessor's attention. For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Interrupts allow an embedded system to respond rapidly to multiple real-time events.

Hardware interrupts are triggered when the interrupt request (IRQ) line(s) of a microprocessor is asserted active by the electrical signals sent from hardware devices. Typically, the processor samples its interrupt input request line(s) at predefined times during each bus cycle. An IRQ is detected if the interrupt line is active when the processor samples it.

Hardware interrupts are asynchronous in the sense that they can occur at any time and at any place in the running program. When an interrupt is received, the processor automatically suspends the program that is currently running, saves its status, and transfers control to a special program called the *interrupt service routine* (ISR). Once the ISR has run to completion, the control returns back to the original program that was suspended.
There are two approaches for a microprocessor to locate the ISR of an interrupting device: nonvectored interrupting and vectored interrupting.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

6.2.1 Non-vectored Interrupting

In non-vectored interrupting, a fixed memory location is used as a common vector (direction) for all interrupt sources.

Figure 6.1 illustrates the process of non-vectored interrupting:

1. While the processor is executing an instruction j of a user program, one of the I/O devices raises an IRQ signal to the interrupt input pin (INT).
2. The processor detects the IRQ. Upon the completion of the instruction j, the processor starts an interrupt acknowledge cycle. At this point, the value contained by the program counter register (PCR) is the location of the next instruction of the user program.
3. In the interrupt acknowledge cycle, the status register (SR) and the PCR are saved (say, into spare registers) so that the user program can be resumed later.
4. The PCR is loaded with the fixed memory location where the common interrupt vector is saved.
5. This common interrupt vector is typically a jump instruction, pointing to the start location of the ISR.
    5.1. Inside the ISR, the commonly used registers are first saved onto the interrupt stack. In so doing, the processor switches its context from the user task to the ISR.
    5.2. Next, the processor searches through the devices, from high priority to low priority, to identify the interrupt source (requesting device). An I/O device often has one or more interrupt status registers that latch its IRQ; the processor can check such registers to ensure IRQs are not missed.
    5.3. The portion of code pertinent to the requesting device is executed, which typically entails the access of the ports (registers) on the device.
    5.4. At the end of the ISR, the top frame of the interrupt stack is popped up and the context of the user task is restored.
6. The original PCR value is restored.
7. The processor is ready to run the next instruction of the user program.

Non-vectored interrupting does not require extra hardware, but involves serial testing at step 5.2, which can introduce some unacceptably long delays in the response to the interrupting device. It is applicable when there are only a few external interrupt sources.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

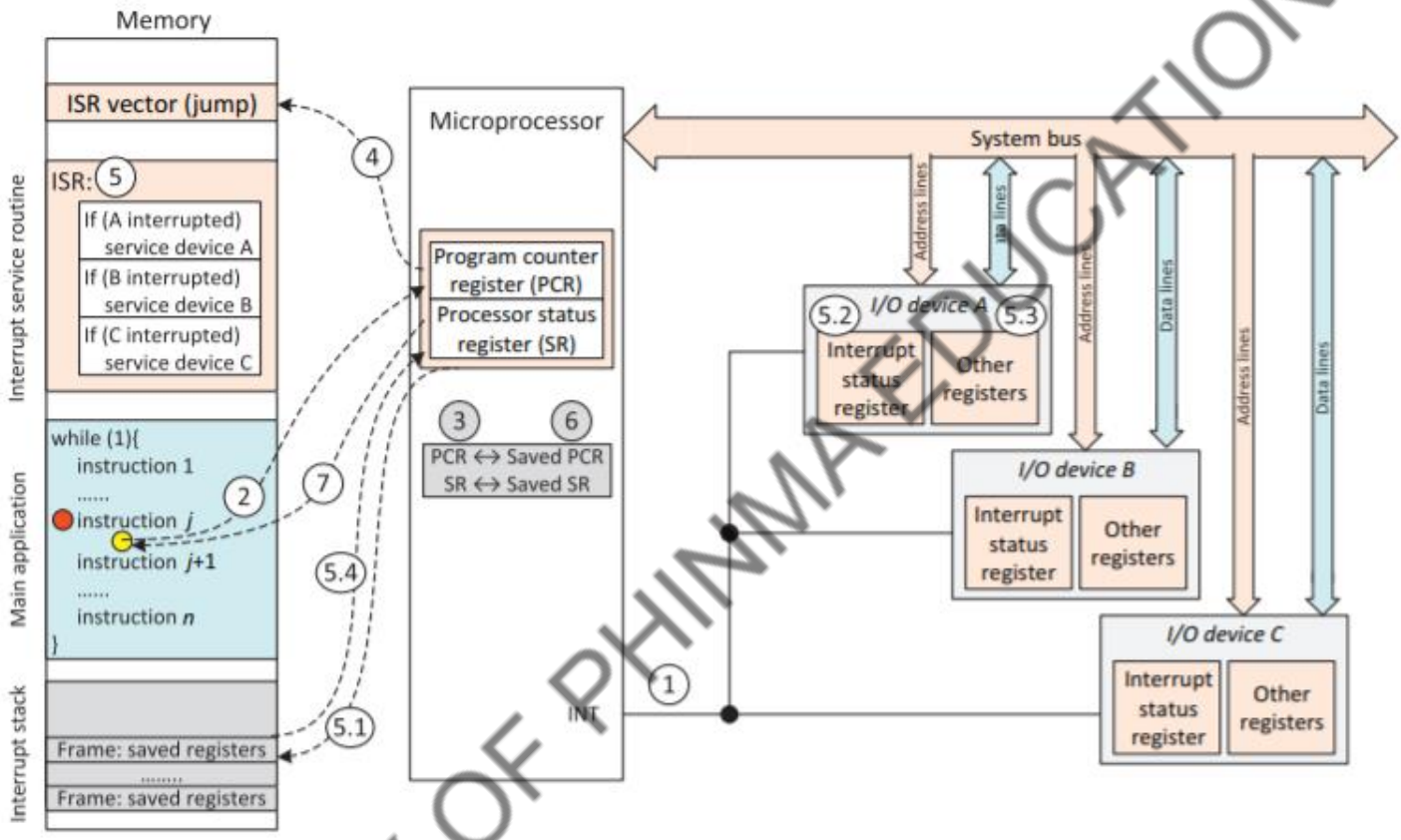


Figure 6.1
Non-vectored interrupting

6.2.2 PIC and Vectored Interrupting

Multiple interrupt sources can fire interrupt requests simultaneously. To respond to multiple requests, it is critical that the microprocessor can rapidly identify each interrupting device and jump to the exact memory location where its own ISR is stored. Here, the identity of a device is called its interrupt vector number, which is stored in a programmable register on the device itself, or more typically on a programmable interrupt controller (PIC).

A PIC, as shown in Figure 6.2, typically has an output pin, which is connected to the INT pin of the microprocessor, and several IRQ lines, each of which can take interrupt signals originating from one or multiple I/O devices. In such a sense, a PIC serves as a liaison to link many external devices and the microprocessor together.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

A PIC normally has three functions. It can
• provide a hub for maskable interrupt sources;
• implement prioritized interrupting at the hardware level;
• cope with different interrupt source modes.

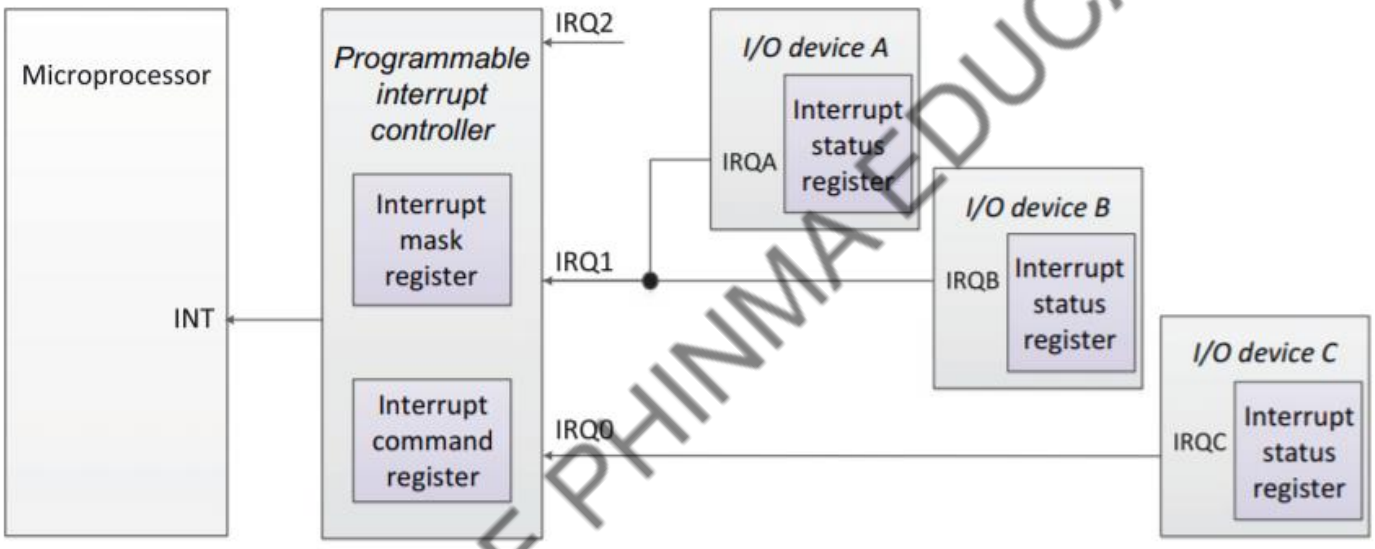


Figure 6.2
General PIC settings.

6.2.2.1 Maskable interrupts

Hardware interrupts can be either maskable or nonmaskable. A nonmaskable interrupt can never be ignored, and is used for critical tasks such as system resets and watchdog timers.

A PIC typically has an interrupt mask register (IMR), which allows you to individually enable and disable interrupts from devices on the system. In general, there is one bit in the IMR that corresponds to each IRQ line of the PIC: writing a 0 (or 1) to enable interrupts emanating from devices wired to the corresponding IRQ line; writing a 1 (or 0) to disable interrupts on the corresponding IRQ line.

When a maskable interrupt source is disabled, its IRQs will not be forwarded by the PIC to the processor.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

6.2.2.2 Interrupt priorities

Interrupt sources are typically prioritized on the basis of their importance. A PIC can be programmed to offer various priority schemes.

For instance, the Intel 8259A PIC chip used for x86 processors has eight IRQ lines. The BIOS (the name originates from "basic input/output system") typically initializes the 8259A chip to use a scheme with fixed priorities: the device on IRQ0 has the highest priority and the device on IRQ7 has the lowest priority.

As another example, the advanced interrupt controller (AIC) used for ARM processors can handle up to 32 interrupt sources. Each interrupt source can have a programmable priority level of 7-0: level 7 is the highest and level 0 is the lowest priority. The interrupt controller has a source mode register for each interrupt source. A user can assign an appropriate priority level to a device by setting the last 3 bits of the corresponding source mode register. If several interrupt sources of equal priority are pending, the interrupt on the IRQ line with the lowest number is considered first.

A priority level is used to dictate the order in which the interrupts will be serviced. When several IRQs are received simultaneously by the PIC, it serializes these requests according to their priority levels: the request from a device with a higher priority is always forwarded to the processor prior to requests from devices with a lower priority.

What happens if the PIC detects a new request from another interrupt source while the processor is executing an ISR? If the new request has a lower priority, the PIC will not forward it to the processor until after the completion of the existing ISR. If the new request has a higher priority, a mechanism called interrupt nesting comes into play.

Interrupt nesting allows interrupts with an equal or higher priority to interrupt an existing interrupt. This is illustrated in Figure 6.3, where the normal execution is interrupted by ISR x, which is interrupted by ISR y, which is further interrupted by ISR z. Once ISR z runs to completion (indicated by end of interrupt, EOI), the control is relinquished back to ISR y, which resumes from the interrupted point and runs to completion, the control is then relinquished back to ISR x, and so on. In other words, the execution of ISR z is nested within ISR y, which is nested within ISR x, which is injected into the normal program execution.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Also notice that interrupt nesting is prohibited at the two ends of an ISR, referred to as the context switching sections. This is for good reason: further interrupts are allowed only when enough of the processor context has been saved onto the interrupt stack. Since interrupts are disabled automatically during context switching, to support interrupt nesting, a designer has to explicitly enable interrupts inside an ISR. This can be done immediately after context switching in order to achieve faster response to higher-priority interrupts.

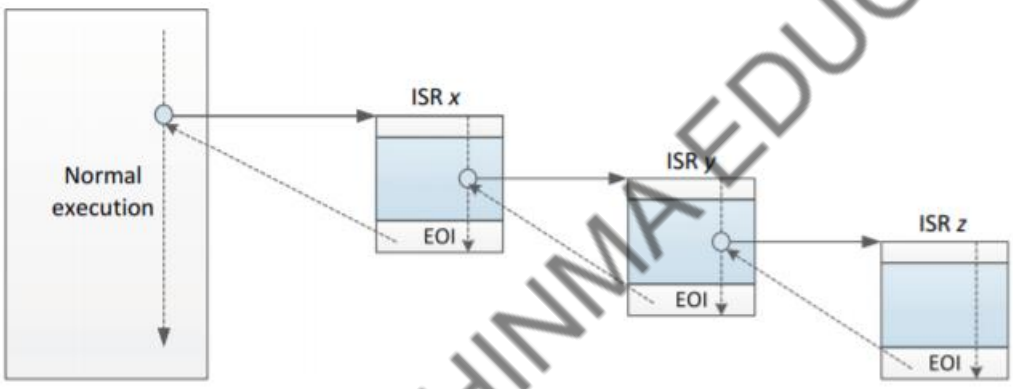


Figure 6.3
Interrupt nesting.

6.2.2.3 Interrupt source mode

The external interrupt sources can operate in one of two modes: level-sensitive mode or edge-triggered mode.

To send an interrupt signal to the PIC, a level-sensitive device needs to drive its IRQ line to the active level, and then hold it at that level until it has been cleared (say, upon service completion). For devices operating in this mode, PIC has a better chance of minimizing spurious signals from a noisy interrupt line (a spurious pulse is often too short to be detected by a PIC).

When multiple level-sensitive devices share an IRQ line, the line remains asserted as long as one or more than one of the devices has an outstanding IRQ. After a short hardware delay (typically a few clock cycles), the PIC drives the INT line to its active level to inform the processor about the IRQ. For a level-sensitive device, upon the completion of its ISR, it should be set to stop driving the corresponding IRQ line. If the interrupt source is not cleared in time, when the processor signals the EOI to the PIC, the PIC would mistakenly detect another outstanding request from the same device on the IRQ line. If this IRQ line has the highest

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

priority, the PIC would immediately reissue an interrupt to the processor, which would loop forever, continually calling the ISR.

When multiple devices—say, A and B—share an IRQ line, if device A sends an IRQ while the processor is serving device B, the request from device A might be transparent to the PIC (hidden by device B's request). Device A's request would be detected and honored only if device A is still driving an active signal on the IRQ line after source device B is cleared at the end of ISR-B. Should ISR-B take too long, the processor might not be able to service device A in a timely manner. Even worse, if there is a device that the processor does not know how to service, then any interrupt from that device could permanently block all interrupts from the other devices sharing the same IRQ line.

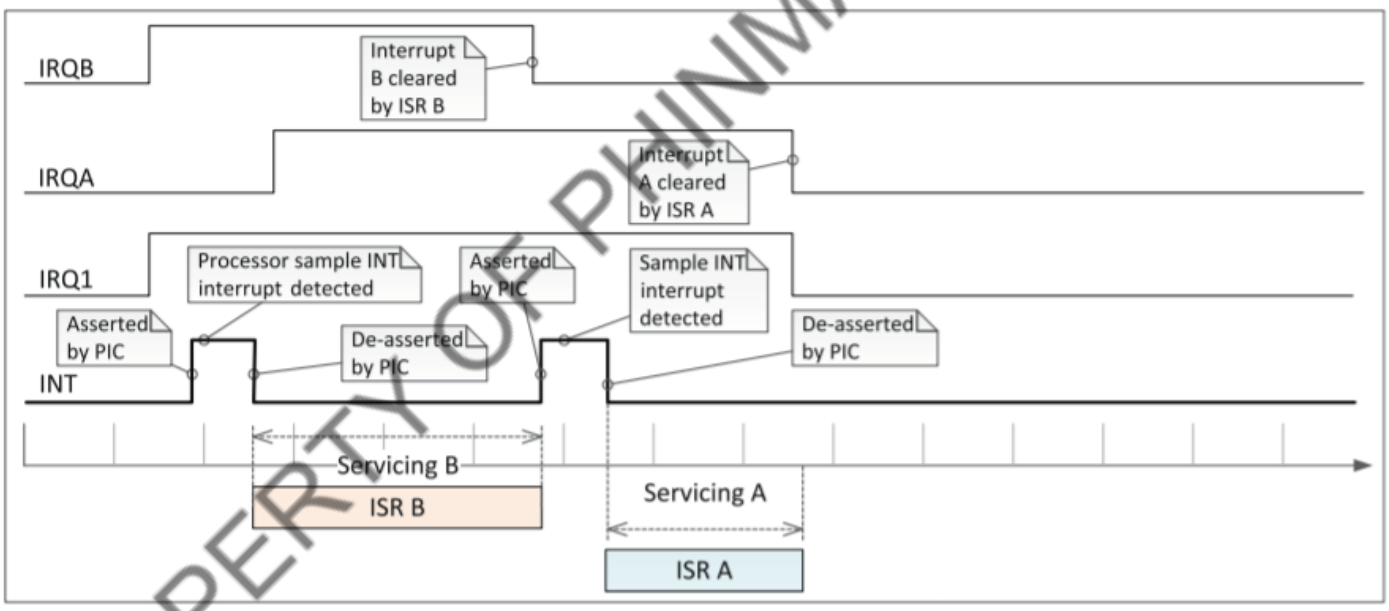


Figure 6.4
Level-sensitive sources.

Figure 6.4 shows a scenario with two level-sensitive sources. Notice when each of the interrupt sources is cleared and when the INT line is asserted and deasserted (there is a hardware delay between the IRQ1 and the INT signals).

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

To send an interrupt signal to the PIC, an edge-triggered device needs to drive a level transition on the interrupt line, either a falling edge (high to low) or a rising edge (low to high). In other words, a PIC detects IRQs from edge-triggered devices by clear-assert transitions. If the source of the interrupt can be cleared only by the corresponding ISR, there exists a critical issue when multiple edge-triggered devices share an interrupt line. Suppose device A asserts an IRQ before ISR-B clears its source (device B). The IRQ line is still asserted (by device A's request) even after ISR-B has cleared its source. However, the IRQ from device A cannot be detected by the PIC because it is waiting for a clear-assert transition. The code for clearing source device A is in ISR-A, which unfortunately is not running. Consequently, the PIC will never see a clear-assert transition on that IRQ line, and the system will behave abnormally.

Advanced interrupt controllers can clear the interrupt source after the processor has started to handle its IRQ. In such a case, service of one device can be postponed arbitrarily, and IRQs from other devices on the same IRQ line can still be detected and honored. Even though there is a device that the processor does not know how to service, it may be taken as spurious interrupts, and will not interfere with the IRQs from other devices.

Figure 6.5 shows a scenario with two edge-triggered sources. Notice when each of the interrupt sources is cleared by the PIC and when the INT line is asserted and deasserted. Also notice that the service to device B is interrupted by the service to device A.

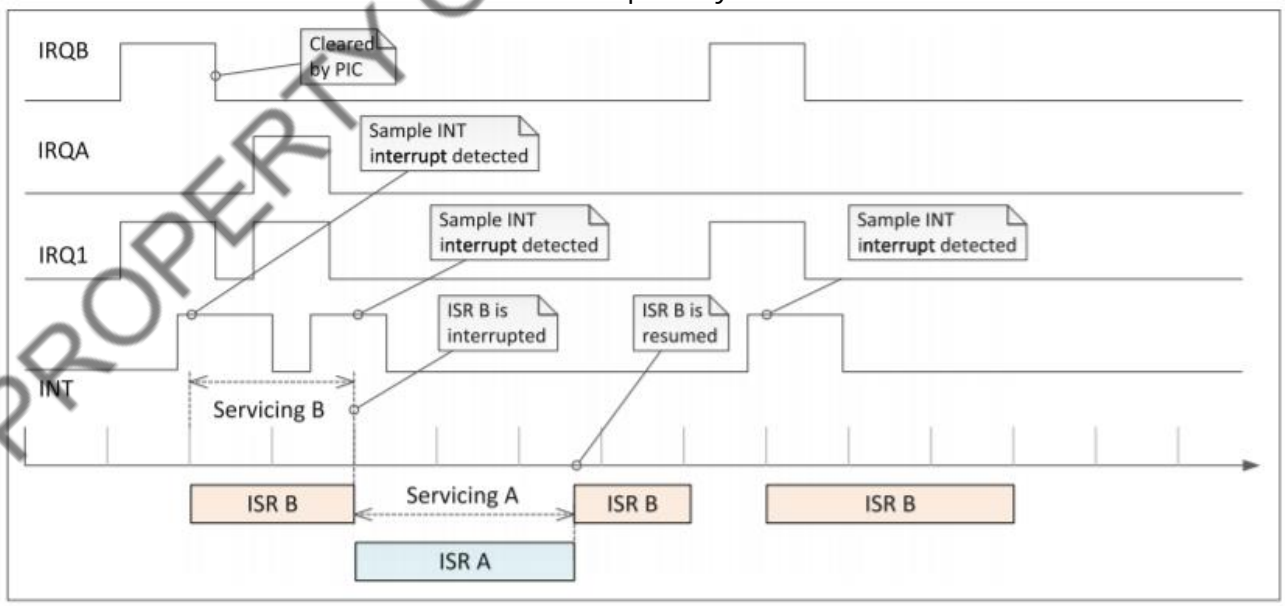


Figure 6.5 - Edge-triggered sources

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

6.2.2.4 Interrupt vectoring process

In vectored interrupting, a special block of memory is typically reserved to accommodate a data structure called an interrupt vector table—a table with a sequence of entries called interrupt vectors. Each interrupt vector contains information that points the processor to the start address of the corresponding ISR.

Figure 6.6 illustrates the process of vectored interrupting:

1. Device A drives an IRQ on the IRQ0 line of the PIC.
2. The PIC detects the IRQ on IRQ0. It first converts the request into a vector number corresponding to device A, stores it in a register, and then asserts the INT line to inform the processor.
3. While the instruction j of the current program is being executed, the processor samples INT and detects an asserted line. After the execution of the instruction j is completed, the current program is suspended. At this point, the value contained by the PCR is the location of the next instruction of the current program.
4. In the interrupt acknowledge cycle, the processor asserts the INTA (interrupt acknowledge) line to the PIC, expecting an interrupt vector number.
5. The PIC drives the interrupt vector number associated with device A to the system bus.
6. The PIC then deasserts the INT line (so that a new IRQ can be asserted);
7. The status register and the PCR are saved (say, into spare registers or pushed to the interrupt stack). This will allow the processor to resume the execution of the original program later.
8. To protect context switching, further interrupts are disabled. The PCR is loaded with the address of the appropriate interrupt vector (an entry in the interrupt vector table).
9. The interrupt vector is typically a jump instruction, pointing to the start location of the ISR for the requesting device.
    9.1. Inside the ISR, the commonly used registers are first saved onto the interrupt stack. At this point, the processor has switched its context from the last task to the current ISR. Further interrupts can now be enabled, if interrupt nesting is desired.
    9.2. The portion of code pertinent to the requesting device is executed, which typically entails the access of the ports (registers) on the device.
    9.3. At the end of the ISR, interrupts are again disabled during context switching. Sometimes a special instruction is needed to inform the PIC about the end of the current interrupt. The top frame of the interrupt stack is popped up and the context of the original task is restored.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

10. The PCR and the status register are restored to their respective values before the interrupt;
11. The processor is ready to run the next instruction of the original program.

For a level-sensitive device, it can continuously assert its IRQ line until it is serviced. By deasserting the INT in step 6, the PIC is virtually isolating the interrupt source being serviced from the processor. Whenever there is a new IRQ with a higher priority, the PIC can assert the INT line again so that the processor can suspend the execution of the current ISR and switch its context to service the higher-priority interrupt.

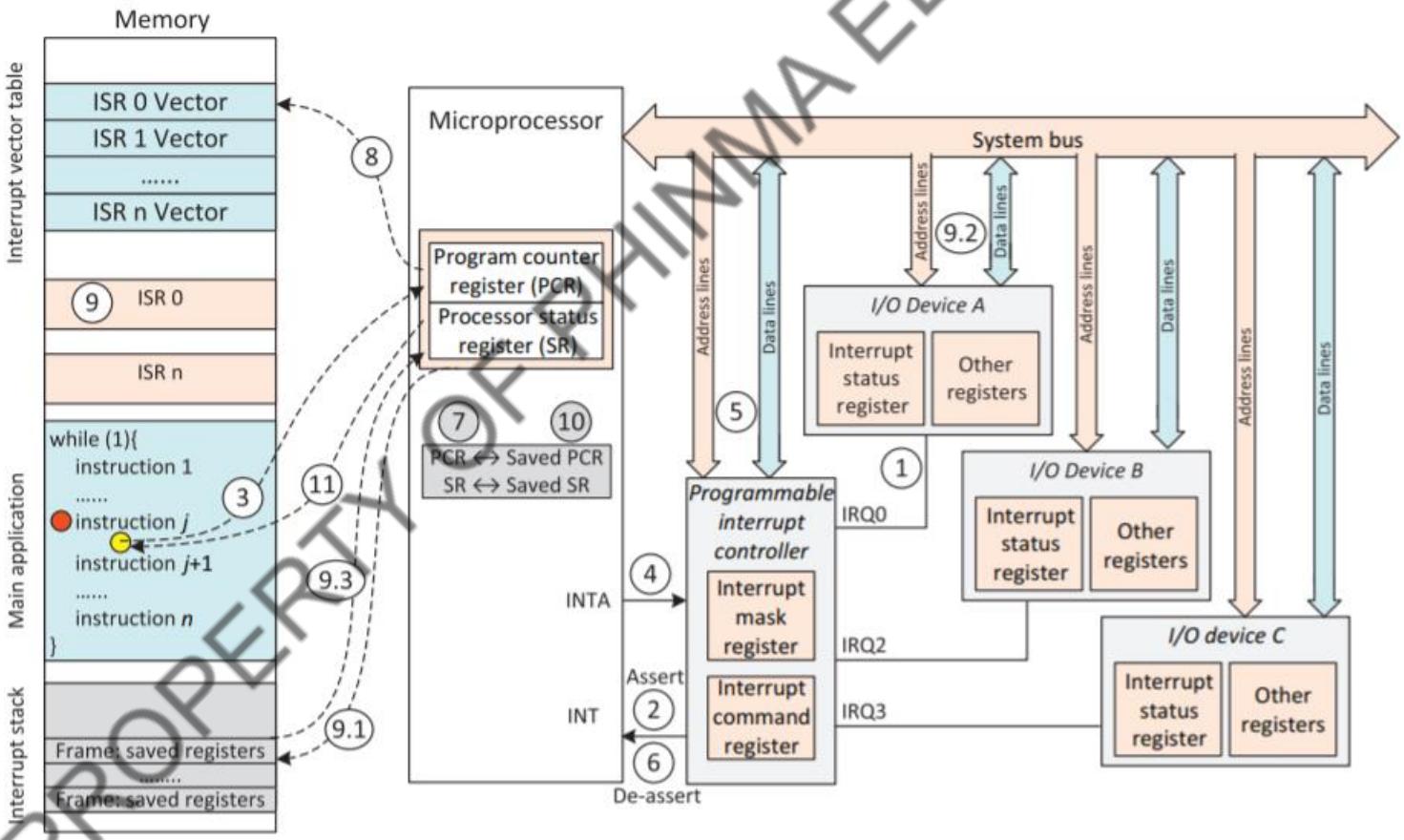


Figure 4.6
Vectored interrupting.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

6.3 Software Interrupts

Software interrupts are synchronous events generated by special processor instructions placed in a program.

For example, pressing a key on the keyboard triggers hardware interrupts (the interrupt vector number is 09h on x86 systems), and the corresponding ISR will typically place the code of the key into a keyboard buffer—a circular queue in a protected memory block. However, those keyboard inputs are not useful unless they are processed by a user application, and only programmers know when and where an input is needed. We all know that functions such as getChar() can be explicitly used in a program to read characters typed through a keyboard. Under the veil, however, it is software interrupts that carry the ball: indeed, the function getChar() uses some assembly instruction to trigger a special ISR to read a character from the protected circular queue.

When a PC is turned on, a firmware called the BIOS takes control and populates the processor's interrupt vector table with the addresses of default ISRs. Later, an operating system, if installed, can take advantage of the services (i.e., ISRs) offered by the BIOS to talk to the installed hardware. In so doing, the operating system is actually using software interrupts to request services from the BIOS.

An operating system can also extend the interrupt vector table, if allowable, by adding more vectors to the table to offer extensive services. Many modern operating systems (e.g., QNX) even choose to install their own ISRs to directly control hardware, completely bypassing the built-in BIOS interrupt facility. The interrupt services offered by an operating system are available to the users in the name of "kernel functions." Next time your code invokes a system call, you will know that it is indirectly raising a software interrupt to the processor (through the operating system).

For example, in order to request a hardware service (e.g., read from the keyboard buffer), a user program can use a software interrupt to get the operating system's attention, causing control to be passed back to the operating system's kernel. The kernel will then process the request, and the execution of the user program is resumed once the service has been done.

On x86 systems, software interrupts are initiated by executing the int instruction :
int vector_number;
where vector_number is an integer in the range 0-255.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

For example, the BIOS offers application-level interfaces to several hardware-related services [32, 40]. An interface with the keyboard services is under the vector number 16H. By int 16H, a program can invoke the ISR addressed by the entry 16H. This ISR actually provides several functions. The AH register is used to indicate the desired function number. For example, the following two instructions,

```
mov AH, 00H;
int 16H;
```

together are used to read a character from the keyboard buffer. If the keyboard buffer is empty, it waits until a character is entered.1 The ASCII code of the key is returned in the AL register.

An ARM processor has a single vector at 0x08, which stores the address of the ISR for software interrupts. On systems built upon ARM processors, software interrupts are initiated by executing the swi instruction:

```
swi sn;
```

where sn is a 24-bit number used to indicate a specific service type. The service number sn is ignored by the processor, but is utilized by the ISR to branch to a second-level ISR corresponding to the number sn. In other words, by varying sn, an operating system can implement a collection of privileged system functions which can be invoked by applications running in user mode.

6.4 Internal Interrupts

Internal interrupts, also called exceptions, are synchronous events generated by the processor itself whenever some abnormal condition occurs during instruction execution. Internal interrupts are coerced rather than requested.

A processor may assign a fixed interrupt number to an exception type. Take the x86 processor as an example. The vector number 0x00 is reserved for divide error exception; this exception occurs whenever a value is divided by zero. The vector number 0x01 is reserved for single-step exception, which occurs after each instruction if the “trace” bit of the flags register is set to 1 (say, by a debugger).

For ARM processors, the vector at 0x04 is reserved for “undefined instruction” exception, which is raised when an instruction is not recognized after decoding. The vector at 0x0C is reserved for “pre-fetch abort” exception, which occurs when an attempt to load an instruction results in a memory fault. The vector at 0x10 is reserved for “data abort” exception, which

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

occurs when the memory controller indicates that an invalid memory address has been accessed (say, if there is no physical memory for an address, or if the processor does not currently have permission to access a memory region).

It is typically the responsibility of an operating system to install an appropriate ISR for each exception type.

To summarize, let us use an example to clarify the differences among the three types of interrupts. Every human being can be viewed as a smart processor. If, as a student, you are programmed (get used) to sleep during the third 10 min of a class, it is a software interrupt when you fall asleep; everyone in the class is well prepared to see that happen. While you are sleeping, the instructor or your neighbor awakens you. This is an external interrupt. If the topic is very interesting, or the instructor does a good job, or you simply feel guilty about sleeping, then you have fortunately revived this is an internal interrupt.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1) **Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)**

6.1 Describe the process of nonvectored interrupting.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Interrupts?	
	2 What are hardware interrupts?	

3) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is a mechanism by which a microprocessor can alter its flow of execution to handle events.

_____2. These events are those that can occur at any time and typically occur at unanticipated spots of the running program each instruction to determine if the processor should enter into an interrupt acknowledge operation.

_____3. These events are those that can occur only at planned or anticipated spots of the running program.

_____4. These are events generated by external hardware devices to get the microprocessor's attention.

_____5. These are events generated by special processor instructions placed in a program.

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2											Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. Where does interrupt commonly used?

Interrupt is a commonly used mechanism for computer multitasking, especially in real-time computing. Hardware interrupts are events generated by external hardware devices to get the microprocessor's attention. For example, pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position. Interrupts allow an embedded system to respond rapidly to multiple real-time events.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #7

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

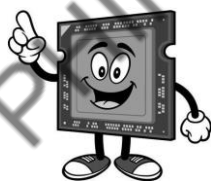
Lesson title: Embedded System Boot Process
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand how System Bootloader works;**
2. **Understand the System Boot Process.**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"Ordinary people think merely of spending time, great people think of using it." So, what are you waiting for? Let's use your time wisely by feeding your brain with these new learning!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about Interrupts including: Internal Interrupts; Software Interrupts; and External Interrupts. Now, we proceed to how embedded system does its booting process.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is a bootloader?	
	2 How does system boot works?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

7.1 System Bootloader

Given that an embedded application—an object in Executable and Linking Format (ELF)—is ready, our focus now is on how to run the application on its target platform. This job is performed by a special piece of software called the system bootloader or bootstrap code,1 which is typically programmed in the nonvolatile memory (NVM) at a location where the reset vector points.
The bootloader's main job is to initialize some hardware components to prepare for loading an ELF object. In particular, it:

- prepares a "loading path" for the ELF object by initializing peripheral devices such as the system timers and the serial/network interfaces;
- prepares a "storage room" for the ELF object by initializing the memory controller and memory chips;
- prepares a "running environment" by initializing the interrupt controller and installing default interrupt service routines.

Some specialized bootloaders are primarily used in the development phase. For instance, a monitor is a bootloader that further allows a developer to debug the target system at run time. A monitor typically has a well-defined command interface. Via a terminal connected to the debug serial port of the target board, a developer can issue commands to debug a running application, including activities such as:

- resetting the embedded system;
- accessing system registers or memory locations;
- setting/clearing breakpoints; and
- stepping through instructions.

Some specialized bootloaders are primarily used in released products. For instance, some bootloaders, upon system reset, are directed to load software (user application) from an attached electrically erasable programmable read-only memory (EEPROM) accessible through an Inter-Integrated Circuit (I2C) interface or a serial peripheral interface (SPI). Such an I2C/SPI bootloader is designed to allow easy upgrade of software in the field: changing the firmware is as simple as replacing a memory chip in a socket. As another example, the controller area network (CAN) bootloader, extensively used in the automotive industry, allows in-field upgrades of software/firmware over the CAN bus to fix bugs on cars that have been sold.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

7.2 System Boot Process

Figure 7.1 illustrates the system boot (or bootstrapping) process. Upon reset of an embedded system, the microprocessor, directed by the instruction at its reset vector, starts to run the bootloader.

After minimum hardware initialization, the system bootloader comes to a decision point, where it may follow a specific scanning sequence to locate an ELF object.

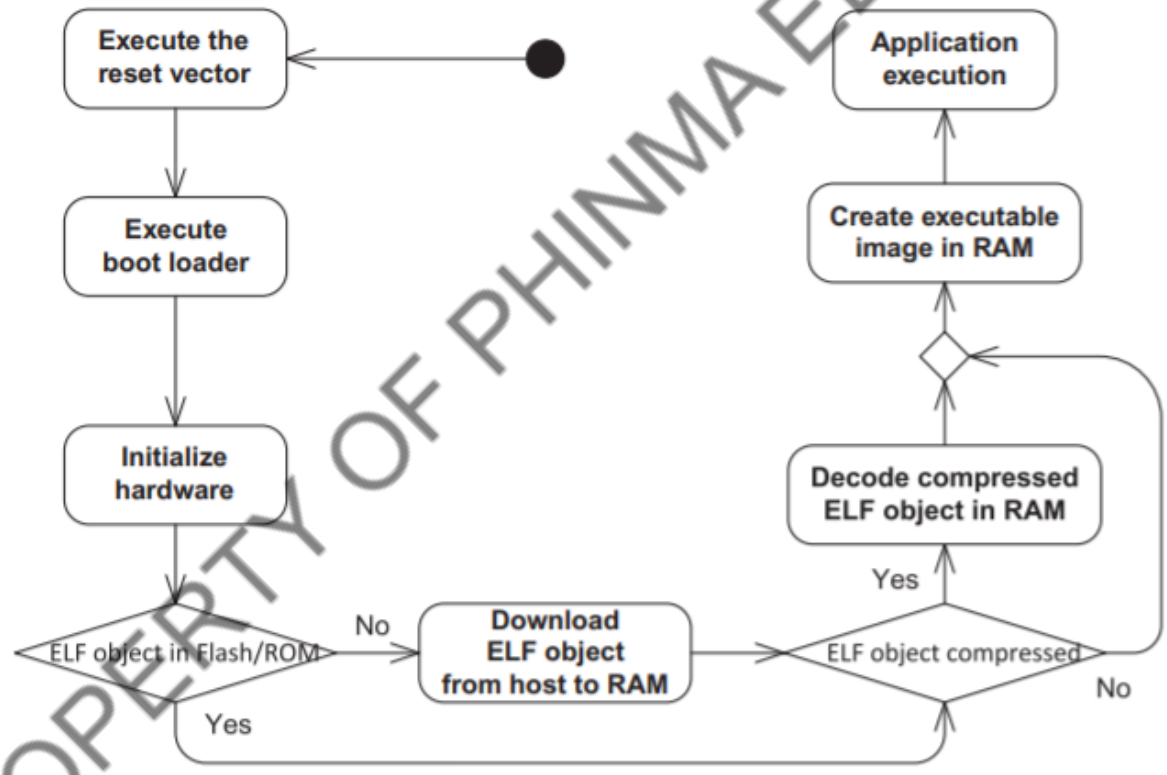


Figure 7.1
Embedded system boot process.

development phase, in order to avoid repeatedly programming the target board, it is preferable to load an ELF object from the host platform via a serial or network connection. Once the embedded software is finalized and ready for performance testing or delivery, the bootloader can be set up to load an ELF object directly from the NVM on board.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

7.2.1.1 From host node

In this case, the system is configured such that the bootloader expects to receive an ELF object from a host node.

The bootloader first needs to initialize a hardware interface prior to "listening" for incoming bytes via that interface. Commonly used interfaces include a serial interface and a network interface:

- Serial interface: The bootloader cooperates with a utility program running on the host platform to make an agreement on communication parameters such as baud rate and packet size. Afterward, commands and binary data (i.e., the ELF object) can be transferred over the serial connection.
- Network interface: The bootloader needs to work with a utility program, such as a File Transfer Protocol server or a Trivial File Transfer Protocol server, running on the host platform. Commands and binary data can be transferred over the network connection after communication parameters have been set up.

If the bootloader detects data arriving, the data are saved at a temporary location in random access memory (RAM). The bootloader is responsible for checking the integrity of the transferred object once all the data have been received.

If no data come through the intended interface in a given amount of time, the bootloader will time-out, then execute a jump instruction to NVM and start executing instructions there, if applicable.

7.2.1.2 From NVM on board

In this case, the bootloader is instructed to get the embedded software from some specific NVM on the target board itself. Hence, the assumption is that the software has been directly "burned" into the NVM. If so, the boot process simply moves to the next step.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

7.2.2 Prepare Embedded Software for Execution

7.2.2.1 Decode compressed ELF object

An ELF object, regardless of whether it is programmed in NVM or loaded temporarily in RAM, might be in a compressed form to reduce demand for storage memory or transmission time.

A compressed ELF object is not directly executable until it has been decompressed. Decompression can be performed by the system bootloader, but often it is performed by a secondary bootloader to which the system bootloader transfers the execution control. The uncompressed ELF object is typically relocated to another location in RAM, and the memory space holding the compressed object can be recycled afterward.

7.2.2.2 Create executable image

At this point, the (secondary) bootloader can create an executable image for the ELF object—mapping file segments into memory segments. In this way, the loadable segments of the ELF object are relocated to their respective run addresses:
• The .text section (segment) is copied to its run address specified by vaddr
• The .data section (segment), which contains initialized data, is copied to its run address.
• The .rodata section (segment), which contains constant data, is copied to its run address.
• The .bss section (segment) contains uninitialized data; its filesz is zero. The loader reserves memory space for the .bss section according to the memory requirement specified by vaddr and memsz.

• In addition, the bootloader also reserves a stack space in RAM for the newly created "image" (recall that the stack space is used for holding local values in function calls). The processor's stack register is set to point to the beginning of this stack.

Lastly, the bootloader executes a processor-specific instruction to jump to the beginning of the .text section of the newly created image. At this point, the boot process completes. The processor continues to execute the application code in the .text section until it runs to completion or the system is powered off.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1) **Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)**

7.1 What is a system bootloader? What is a monitor? What is their major difference?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) **Activity 4: What I Know Chart, part 2 (2 mins)**

What I Know	Questions:	What I Learned (Activity 4)
	1 What is a bootloader?	
	2 How does system boot works?	

3) **Activity 5: Check for Understanding (5 mins)**

Illustrate the Embedded System Boot Process.

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. What is the main job of bootloader?

The bootloader's main job is to initialize some hardware components to prepare for loading an ELF object.

2. Why is it called "bootloader"?

It is called a bootloader in reference to a legend about Baron Müanchhausen, who made an impossible task possible—to pull himself up by his bootstraps.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #8

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

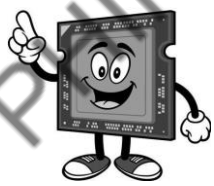
Lesson title: Architecture Modeling in UML (Part 1)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Identify the Levels of Architectural Abstraction;**
2. **Understand the UML Structure Diagram**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"If there are nine rabbits on the ground, if you want to catch one, just focus on one." Stay focus and let's start this thing ahead!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about how embedded system does its booting process. Now, we proceed to the first part of Architecture Modeling in UML.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are the Levels of Architectural Abstraction?	
	2 What is a UML Structure	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

8.1 Levels of Architectural Abstraction

At the systems engineering level, a system is the integration of many designs/implementations, each from a unique angle. For example, an embedded system may involve engineering activities from several disciplines: mechanical engineers see the mechanical parts such as engines and heat transfers, electrical engineers see the electrical parts such as power supplies and motors, computer engineers see the circuit parts such as processors and wirings, and software engineers "see" the intangible parts such as task scheduling and collaboration.

From each disciplinary perspective, a system can be further organized at several abstraction levels. Let us now take the software engineering perspective.

At the system level, a software system has a boundary which represents its scope of responsibility or functionality. Through its interface, the system provides services to or request services from human users or other systems.

At the subsystem level, the system is treated as the integration of many interacting subsystems. The term "software architecture" comes into play at this level. It is used to refer to the high-level structure of a software system, concerning key software elements (or modules) and their relationships. For example, a software system may be composed of several subsystems: one is designated for data management, one is responsible for security control, and yet another supports graphical user interfaces. All the subsystems collaborate together, directly or indirectly, to form a complete functional system.

Below the subsystem level is the component level, where a subsystem is formed by many components that collaborate with each other through their respective ports and interfaces. For example, a home intrusion detection system may have a subsystem running at the security center and a subsystem deployed within each house. The subsystem at the security center may be composed of a monitoring component, a communications component, and an image processing component. The subsystem in a private dwelling may be composed of a control panel component and a sensing component with several sensors.

The lowest abstraction level is the object level, where each component is composed of many role parts to be played by objects. A component can be passive in the sense that the contained

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

objects execute only in response to an external service invocation. An active component, on the other hand, has at least one active object; each active object has its own thread of execution. The question now is how to document the design of components and subsystems of a complex system. We already know that a UML package diagram can be used to organize classes and data types belonging to certain software elements (or modules). However, UML package diagrams are not appropriate for capturing intermodule relations other than "import" and "merge." Indeed, a package merely serves as a namespace to group semantically related concepts (classes) together; it does not dictate how objects will be organized and deployed at run time.

We next introduce the notion of a structured class, which is the basis for modeling tasks, components, and subsystems.

Throughout the rest of this lesson, we take the computer engineering perspective, showing how to model components, subsystems, and the complete system of the AT91SAM9G45 evaluation board.

8.2 UML Structure Diagram

A structured class is a class whose behavior can be completely or partially described through interactions between parts.

A structured class can have properties, each of which is an element with a name, a type, and a multiplicity. When an instance of the structured class is created, depending on its multiplicity, a set of instances can be created for each property. The term "part" refers to a special kind of property that is strongly owned by the containing class.

A port is a property of a structured class that specifies a distinct interaction point between the class and its environment (public port) or between the class and its internal parts (protected port). A port may be associated with provided interfaces, specifying the services the class offers to its environment, as well as required interfaces, specifying the services that this class expects from its environment. A simple port has just a single required or provided interface; a complex port can have as many provided and/or required interfaces as needed. The provided and required interfaces at a port completely characterize the interactions that may occur between the class and its environment through that port. Multiple ports can be defined for a class, enabling different types of interactions to be distinguished on the basis of the port through which they occur.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

By decoupling the internals of a class from its environment, ports allow a class to be defined independently of its environment, making the class reusable in any environment that conforms to the interaction constraints imposed by its ports.

The properties, parts, and ports of a structured class are related by connectors (sometimes also called contextual associations). A connector specifies a link that enables communication between two or more property instances. The link may be realized by something as simple as a pointer or by something as complex as a network connection. In contrast to associations, which specify links between any instances of the associated class, connectors specify links between those property instances involved merely in a specific instance of the structured class.

Besides parts and ports, a structured class may also have properties that are "weakly" contained by reference. Such a property is called a "referenced property." Whenever an instance of a structured class is deleted, excepts for referenced properties, all the instances corresponding to its parts, connector links, and ports are destroyed recursively.

A structured class can be shown in a UML internal structure diagram, a special kind of composite structure diagram. As an example, Figure 8.1(a) shows a structured class Interrupt Controller. The structured class itself is shown by a rectangle separated into a name compartment and a body compartment. Inside the body compartment, a part is shown by a box with a solid outline. The box symbol may contain a string of the form name:type [int], specifying the property name, type, and multiplicity. For example, d: EdgeSignalDetector[1] denotes a property d which is an object of type EdgeSignalDetector. A referenced property is

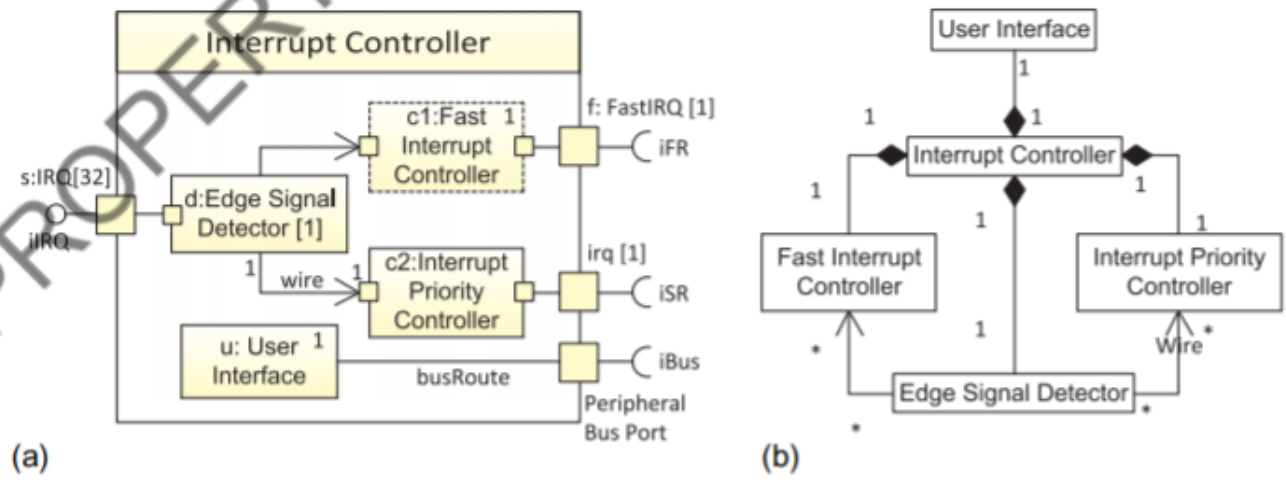


Figure 8.1 - An example UML internal structure diagram

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

shown by a box with a dashed outline (e.g., c1: FastInterruptController). The type of property can be elided, implying that the property has a type which is an anonymous class nested within the namespace of the structured class. The multiplicity for a property may also be shown in the top-right corner of the box symbol (e.g., the UserInterface part). When elided, the default multiplicity is one.

A port of a class is shown as a small square, placed either overlapping the boundary of the class box (public port) or inside the class box (protected port). All the ports in Figure 8.1(a) are public ports. A port can be optionally annotated by a string of the form name: type [int], specifying the port name, port type, and multiplicity. A provided interface may be shown using the "ball" notation attached to the port. A required interface may be shown by the "socket" notation attached to the port. If there are multiple interfaces associated with a port, the list of interfaces is separated by commas. The class Interrupt Controller has four ports declared:

(1) s:IRQ[32]: a port named s of type IRQ with a provided interface iIRQ. At run time, an object of Interrupt Controller has 32 interrupt request port instances, each of which can be individually referenced. Through these ports, an Interrupt Controller object supports up to 32 interrupt request lines from peripheral devices.

(2) f:FastIRQ[1]: a port named f of type FastIRQ with a required interface iFR. Through this port, an Interrupt Controller object can request a fast interrupt service from a connected processor.

(3) irq[1]: a port named irq of anonymous type with a required interface iSR. Through this port, an Interrupt Controller object can request a standard interrupt service from a connected processor.

(4) Peripheral Bus Port: a port of an anonymous type with a required interface iBus. Through this bus port, an Interrupt Controller object can access peripheral devices.
The required and provided interfaces of a port specify everything that is necessary for interactions through that interaction point. If all interactions of a class with its environment are achieved through ports, then the internals of the class are fully isolated from the environment.

This allows the class to be used in any context that satisfies the constraints specified by its ports. In Figure 8.1(a), the classes Interrupt Controller, FastInterruptController, and InterruptPriorityController are completely isolated from their respective environments.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Specifically, Interrupt Controller can be designed and implemented without any knowledge of the environment where it will be embedded. Afterward, an Interrupt Controller object can be connected to any microcontroller and still function properly as long as the two parties obey the constraints specified by the provided and required interfaces.

A connector is drawn using the notation for association; it thus can have a name and end names, and can be bidirectional or unidirectional. The multiplicity on a connector end indicates the number of instances that may be connected to each instance of the role on the other end. Connectors need not necessarily attach to parts via ports. For instance, the wire connector attaches to a port of the InterruptPriorityController object but attaches directly to the EdgeSignalDetector object. Recall that when a port and/or an interface is used, this implies that the corresponding service is completely encapsulated and the port is the only way to invoke the service.5 Since wire is unidirectional, the object c2 would never invoke a service from the object d, and it thus makes sense to attach wire directly to d without a port or an interface. The connector busRoute, although bidirectional, is also attached directly to the UserInterface object u. In general it is preferable to use ports, but the choice of whether to use a port or not really depends on judgment and the nature of the problem.

The whole-part composition relations of a structured class captured by an internal structure diagram can also be modeled by using a class diagram. However, an internal structure diagram is semantically rich. Let us compare Figure 8.1(b) and Figure 8.1(a) to see the differences:

(1) An internal structure diagram shows not only the structure of a complex class, but also captures run-time information such as objects and object roles, which is beyond the capability of a class diagram.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

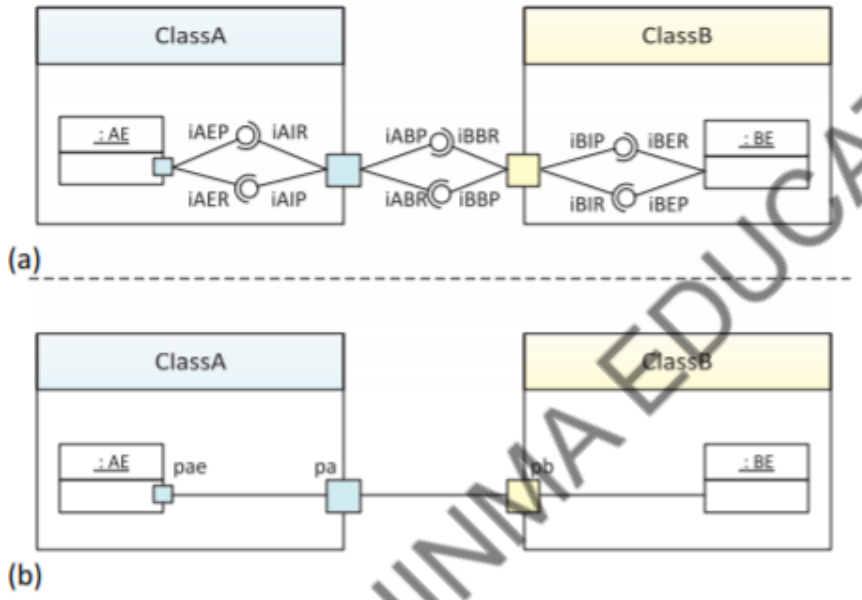


Figure 8.2

Two forms of port connections: (a) interface agreements are shown explicitly; (b) interfaces are suppressed.

(2) An internal structure diagram gives a specific context (the enclosing class) to make sense of the contained relations, whereas a class diagram, owing to lack of context, can represent relations in only a general sense. For example, Figure 8.1(b) specifies that any instance of class EdgeSignalDetector can be linked to an arbitrary number of instances of class InterruptPriorityController. This might be true before board production because a designer is free to use one EdgeSignalDetector to test many interrupt priority controllers. In contrast, Figure 8.1(a) specifies that within the context of class Interrupt Controller, the EdgeSignalDetector object playing the role of d may be connected to only one InterruptPriorityController object playing the role of c1. This exactly captures the postproduction board composition. Of course, instead of modeling the general case, you could modify Figure 8.1(b) such that the wire association becomes a one-to-one relation. This, however, is still not as good as the model in Figure 8.1(a), which also asserts additional constraints on the linked instances: the two objects are not related by chance; they are related (linked) merely because they were owned by the same Interrupt Controller object. The model in Figure 8.1(b) would not prevent such a situation where an EdgeSignalDetector object is linked to one InterruptPriorityController object but they belong to two different Interrupt Controller objects.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

(3) An internal structure diagram exposes services by ports only. As will be shown in the sections below, the notation of a structured class can be carried over to model large-scale concepts such as components and (sub)systems.

Connectors between ports can be shown in two ways. Figure 8.2(a) shows interface agreements explicitly—say, the service provided by interface iABP meets the needs specified by interface iBBR. Figure 8.2(a) also indicates that the ports on the boundary of a structured class serve as a service relay: the service specified by interface iABP is actually offered by the AE object—an internal part of ClassA; similarly, the needs specified by interface iBBR are exactly derived from iBER—the needs of the BE object inside ClassB. Going deeper, we have the following relations among the interfaces in Figure 8.2(a):

- iAEP = iABP = iBIP;
- iAIR = iBBR = iBER;
- iAER = iABR = iBIR;
- iAIP = iBBP = iBEP.

Figure 8.2(b) shows a simplified form where the provided and required interfaces are suppressed. This can be used at or below a certain level of abstraction to prevent balls and sockets from cluttering a diagram. A connector is called a delegation connector when it connects a public port of a class to its parts. The connector between ports pa and pae and the connector between pb and the anonymous BE object are delegation connectors.

Let us look at a slightly bigger structured class. Figure 8.3 gives a class model of the ARM926EJ-S processor:

- An ARM926EJ-S processor is composed of a processor core, a memory management unit, two memory interface units, a 32 KB data cache, a 32 KB instruction cache, a data bus interface unit, an instruction bus interface unit, and an in-circuit emulator.
- Eight public ports are shown on the boundary of the ARM926EJ-S Processor class, where
  - ice is a port providing debugging functions such as downloading code and single-stepping through programs. The interface iJTAG uses the JTAG protocol.
  - d is a port providing debugging information (e.g., to serial port). The interface iDEBG uses the Xmodem protocol.
  - t is a trace port providing a real-time trace capability for the processor core. The provided interface iETM is a trace protocol. An external trace port analyzer can be used to capture the trace information.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• the two anonymous ports connected to the two bus interface units are used to request bus services for accessing data or instructions in external memories.
• c is a port for clock signals (400 MHz).
• i refers to two ports for servicing interrupt requests.

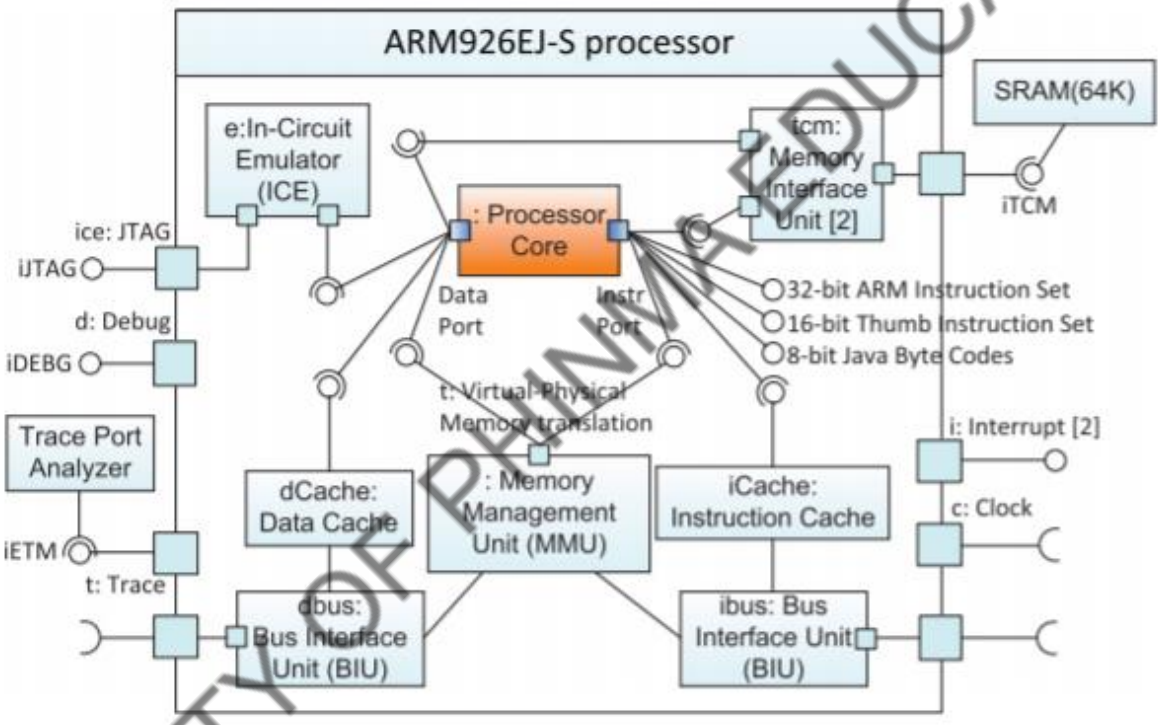


Figure 8.3
The internal structure of the ARM926EJ-S processor.

• the port connected to the two Memory Interface Unit objects is used to request access to fast memory devices. The required service is described by the interface iTCM, which, in this case, is fulfilled by a 64 KB static random-access memory chip.
• The Memory Management Unit object offers a memory address translation service to the processor core.
• The processor core object is encapsulated by two public ports: a data port and an instruction port.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

- The data port of the processor core
  - offers a service to e—an In-Circuit Emulator object;
  - relies on data-access services from tcm, dCache, and the anonymous Memory Management Unit object.
- The instruction port of the processor core
  - has three provided interfaces—accepting a 32-bit ARM instruction set, a 16-bit Thumb instruction set, and eight-bit Java bytecodes;
  - relies on instruction-access services from tcm, iCache, and the anonymous Memory Management Unit object.

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

8.1 For a complex system, why is it necessary to represent its system architecture at multiple abstraction levels?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are the Levels of Architectural Abstraction?	
	2 What is a UML Structure	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is the lowest abstraction level, where each component is composed of many role parts to be played by objects.
_____2. At this level, the system is treated as the integration of many interacting subsystems.
_____3. At this level, a software system has a boundary which represents its scope of responsibility or functionality.
_____4. It is a class whose behavior can be completely or partially described through interactions between parts.
_____5. It is a property of a structured class that specifies a distinct interaction point between the class and its environment (public port) or between the class and its internal parts (protected port).

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. How does a port of a class is shown?

A port of a class is shown as a small square, placed either overlapping the boundary of the class box (public port) or inside the class box (protected port). All the ports are public ports. A port can be optionally annotated by a string of the form name: type [int], specifying the port name, port type, and multiplicity. A provided interface may be shown using the "ball" notation attached to the port. A required interface may be shown by the "socket" notation attached to the port. If there are multiple interfaces associated with a port, the list of interfaces is separated by commas.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #9

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

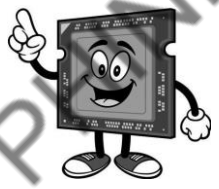
Lesson title: Architecture Modeling in UML
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand the Modeling Components ;**
2. **Understand Modeling Subsystems.**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"We have a strategic plan. It's called doing things." So, let's make that plan into reality!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about the first part of Architecture Modeling in UML which includes the topics: Levels of Architectural Abstraction and UML Structure Diagram. Now, we proceed to the second part which covers the topics Modeling Components and Modeling Subsystems.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Modeling Components?	
	2 What are the Modeling Subsystems?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

9.1 Modeling Components

Component is a design-time concept for organizing run-time objects. It is especially useful in component-based system structuring and development.

A component is an autonomous unit with its internals hidden and inaccessible other than through the provided and/or required interfaces. The provided interfaces specify a formal contract of the services that it provides to its clients, and the required interfaces specify what it requires from other components or parts of the system. The environment is able to interact with a component as long as the environment complies with the component's contract expressed by the provided and required interfaces. This makes a component substitutable: it can be replaced at run time by any other component that offers/requires equivalent or compatible services.

Components in UML are just structured classes. A component is shown as a rectangle with a name, optionally followed by the keyword «component» or a component icon (a component box with two tiny interface boxes on the border). Just like any structured class, a component can contain ports, parts, and connectors, which, respectively, define interfaces, roles, and their connections. Note that objects of the same type (class) may be instantiated to play different roles of a component.

A port can be a simple port—exposing just a single required or provided interface—or a complex port—exposing more than one interface (required and/or provided). Two related parts inside a component can be connected directly by a connector, or indirectly through their ports. Alternatively, the ball-and-socket notation can be used to make the contract explicit: the offered interface (ball) on the port of one part meets the required interface (socket) on the port of the other part. Although visually the ball is connected to the socket, in fact it is the two parts that are connected.

As a side note, a component, being a structured class, can inherit from supertype components and can be extended by subtype components. In such a situation, the Liskov substitution principle (type conformance) applies to the contracts defined on the public ports. The inherited interfaces are indicated on the component diagram by preceding the name of the interface by a forward slash.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Figure 9.1 gives a diagram depicting the Peripheral Composite component of the AT91SAM9G45 chip. A Peripheral Composite has 13 parts; there is one instance for each part except for Media Card Controller and USART (according to the multiplicity, a Peripheral Composite has two media card controllers and four universal synchronous/asynchronous receiver/transmitter units). The Peripheral DMA Controller object manages four direct memory access devices: an LCD controller, an Ethernet controller, a USB controller for programming the AT91SAM9G45 chip, and a controller for normal USB connections.

A Peripheral Composite object also has 12 public ports, each of which is connected to an internal part through its port and offers an interface to the external users.

Figure 9.2 gives a more complicated diagram depicting the System Controller component of the AT91SAM9G45 chip. A System Controller contains six parts: an instance of a structured

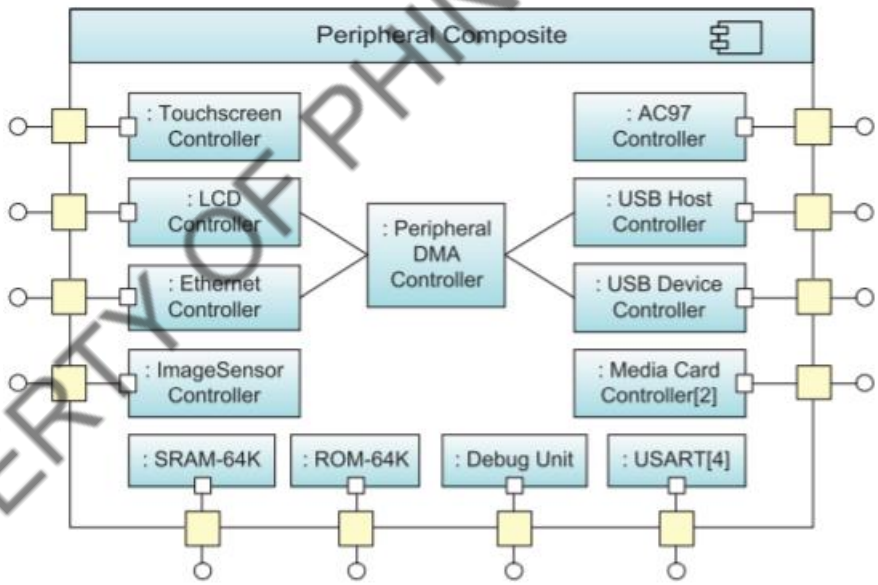


Figure 9.1 - A UML Component Diagram: Peripheral Composite of the AT91SAM9G45 chip

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

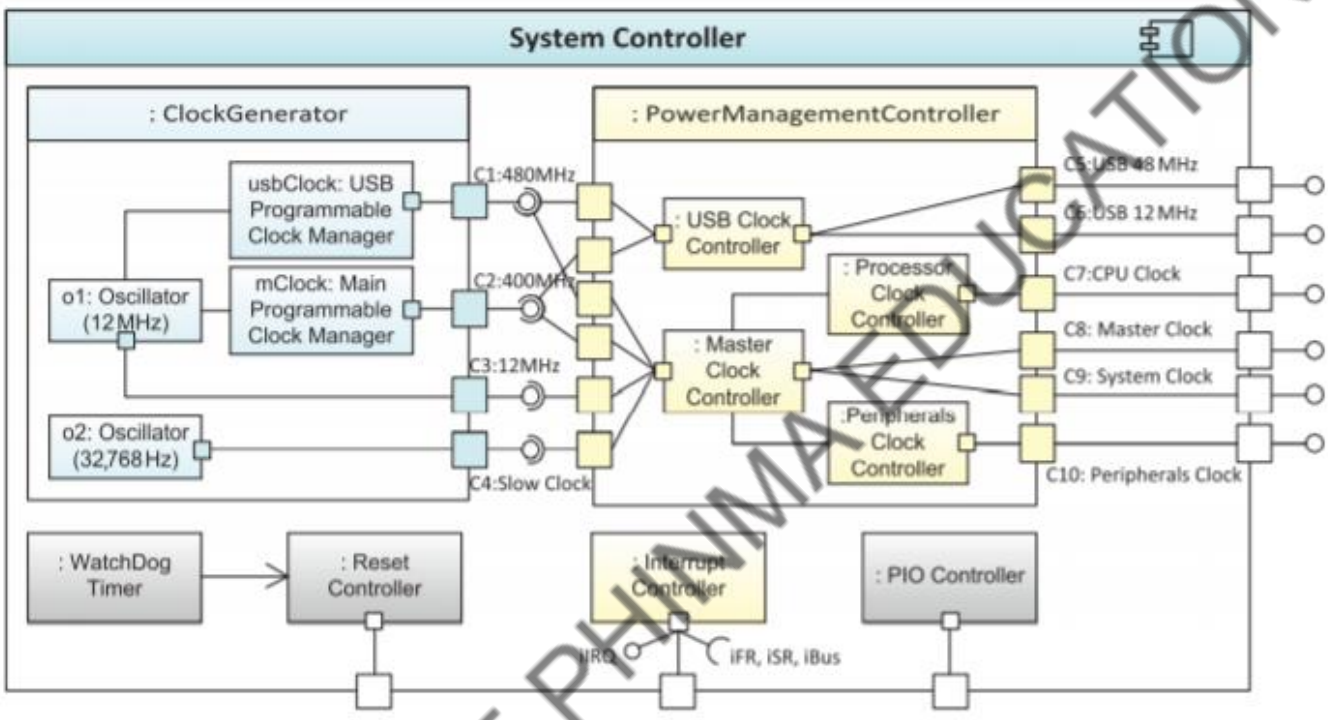


Figure 9.2
A UML Component Diagram: System Controller.

class named ClockGenerator, an instance of a structured class named Power Management Controller, an instance of a structured class named InterruptController, a watchdog timer, a reset controller, and a set of parallel I/O controllers. The internal structures of WatchDog Timer, Reset Controller, and PIO Controller are not our concern here.

A ClockGenerator object contains four parts: an Oscillator object o1 operating at a frequency of 12 MHz, an Oscillator object o2 operating at a frequency of 32,768 Hz, a USB Programmable Clock Manager object usbClock, and a Main Programmable Clock Manager object mClock. Through the four public ports, a ClockGenerator object offers clock signals at 480 MHz (port C1), 400 MHz (port C2), 12 MHz (port C3), and 32,768 Hz (port C4).

A PowerManagementController object contains four parts: a USB clock controller, a master clock controller, a clock controller for the processor, and a clock controller for peripherals. A PowerManagementController object has 12 public ports. On the left border there are six ports, each of which requires clock signals of a certain frequency offered by the clock

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

generator. On the right border there are also six ports, offering clock signals to the system, each of which is linked by a connector to a public port of the System Controller object.

The Interrupt Controller object has a port with one provided interface iIRQ and three required interfaces IFR, iSR, and iBus. Since this port is connected to a public port of the System Controller object, the provided/required interfaces are thus available to the environment of the System Controller object.

9.2 Modeling Subsystems

A subsystem is simply a large-scale component, formed by assembling multiple related components. Thus, a subsystem diagram is visually the same as a component diagram, except that it can be optionally annotated by the stereotype «subsystem» or a subsystem icon (a fork symbol).

Figure 9.3 gives a subsystem diagram, depicting the composition of the AT91SAM-9G45 chip. It has four components: a system controller, a peripheral composite, an ARM926EJ-S processor, and a bus matrix. As labeled, the processor has two bus ports connected to the bus matrix, three system ports (one for clock signals and two for interrupts) connected to the system controller, and a port with four provided/required interfaces connected to the peripheral composite.

As another example, Figure 9.4 shows the Internet Protocol stack as a subsystem composed of subsystems. Internally, the subsystems inside the Internet Protocol stack form a layered architecture, with each offering a service to its immediate upper layer. At the same time, the service offered at each layer can be used by the clients through the corresponding public port. For example, a Web browser relies on application-layer protocols such as HTTP and SMTP; the OSPF protocol functions above the network layer.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

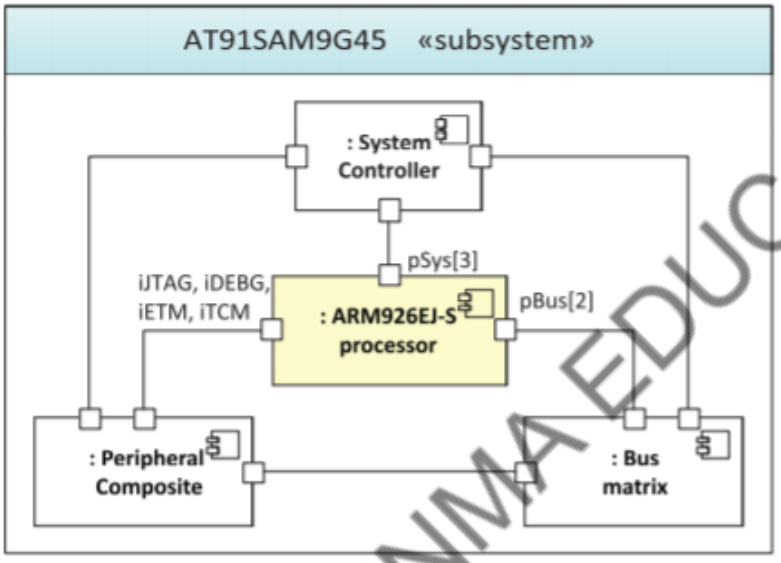


Figure 9.3
The AT91SAM9G45 subsystem.

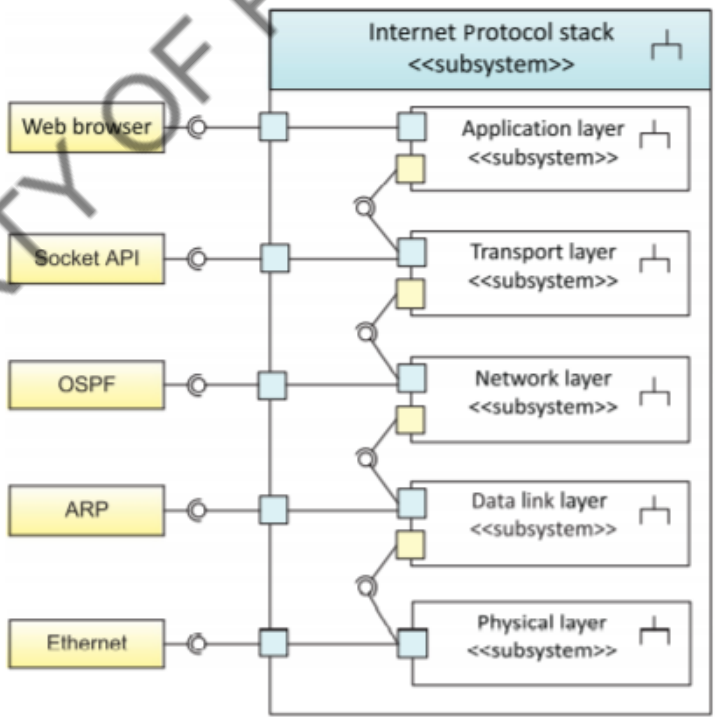


Figure 9.4 - The Internet Protocol stack in UML.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

9.2.1 Class Diagram Versus Subsystem Diagram

Let us use another example to see the differences between class diagrams and subsystem diagrams.

In Figure 9.5 we use a class diagram to model the key concepts in a data processing subsystem. Note that here the term "subsystem" is used as a synonym of "package," quite differently from the subsystem concept defined in UML. Figure 9.6 shows the subsystem diagram representing a run-time configuration instantiated from classes defined in the data processing package.

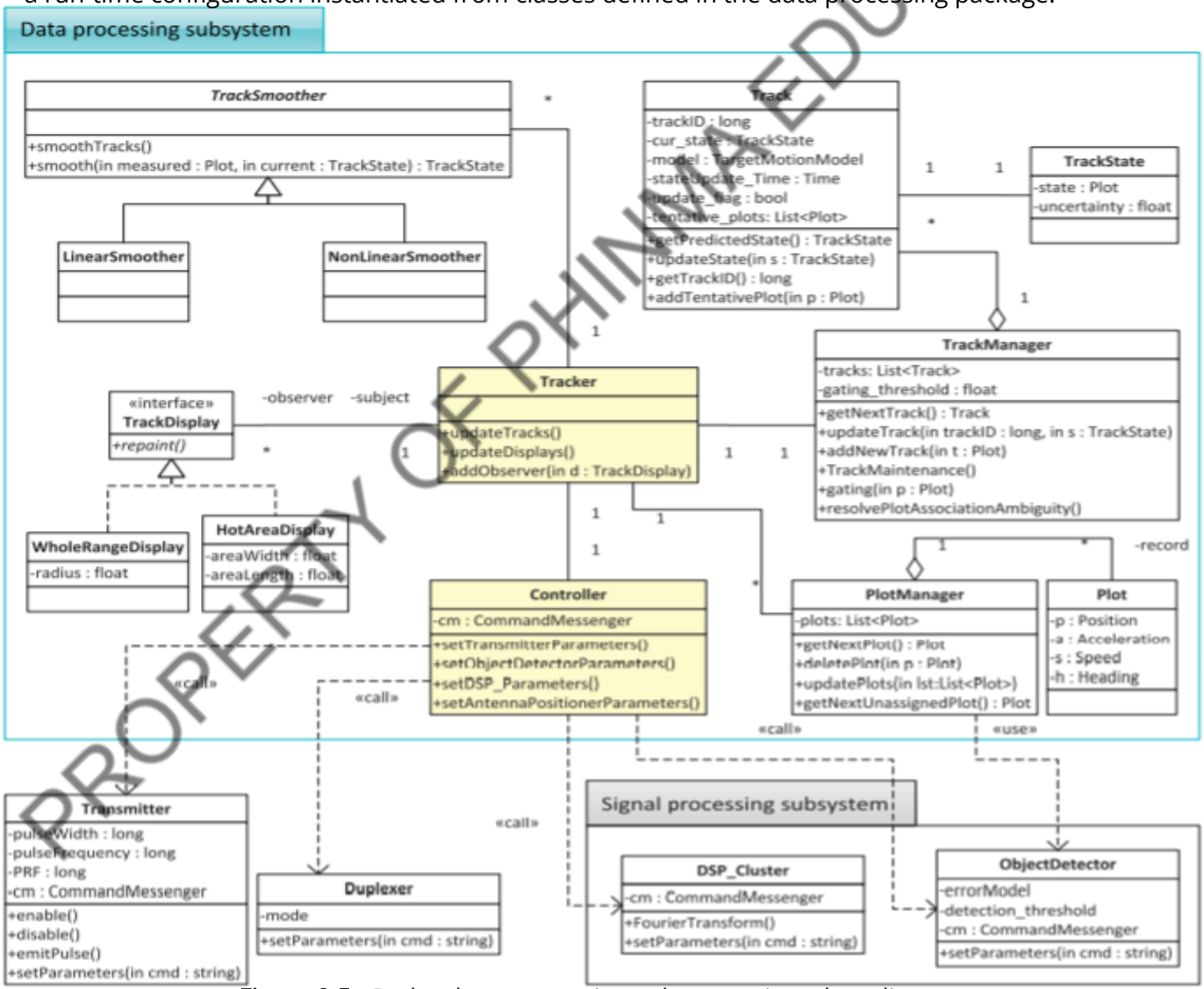


Figure 9.5 - Radar data-processing subsystem in a class diagram.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

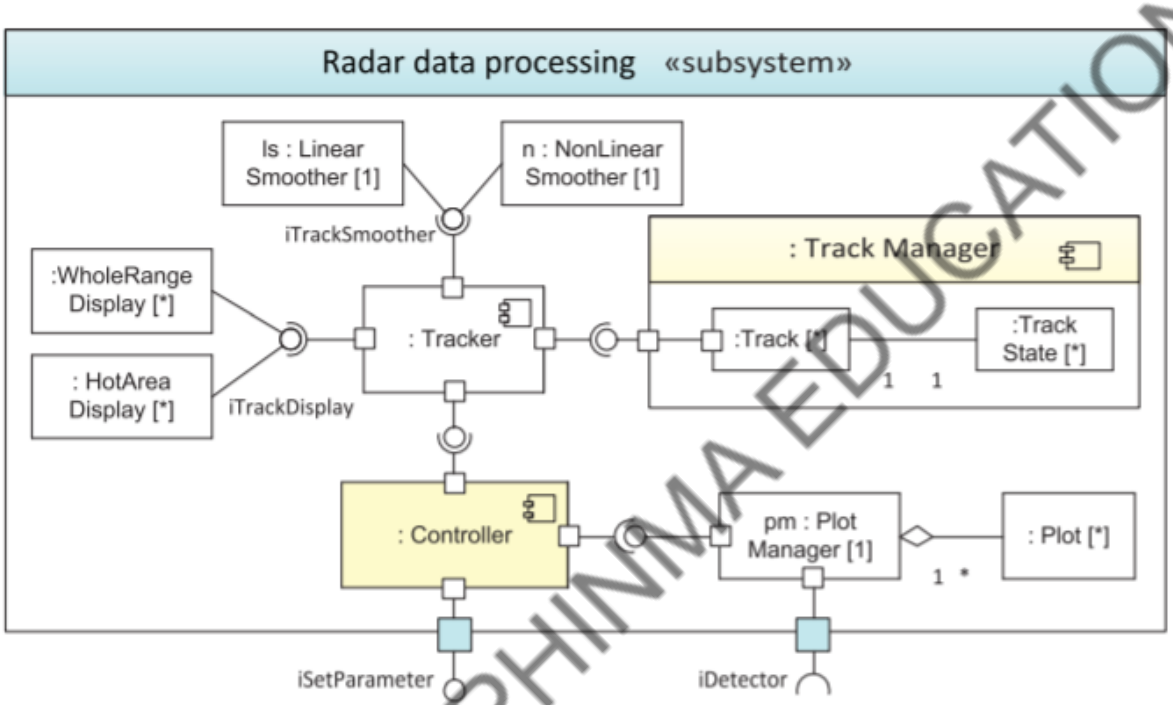


Figure 9.6
Radar data processing subsystem diagram.

Obviously, the class diagram is a static model, while the subsystem diagram is a dynamic model. Particularly,

- a class diagram defines a template for infinitely many object diagrams, specifying object relationships in a general sense. In a class diagram, the objects of one class can play different roles specified by different associations. As a comparison, a subsystem diagram defines a template for many subsystem instantiations, specifying object relationships in terms of roles and role fulfillment under a specific context. In a subsystem diagram, multiple objects of the same class can be referenced by one part to play the same role or by different parts to play different roles.
- a class is typically defined in a single place (package). Objects of a class can exist in many places of a system. Neither class diagrams nor object diagrams can specify where the objects are supposed to go. In contrast, a subsystem diagram (and its kinds such as structured class diagram and component diagram) dictates a specific environment (context) where the contained objects will reside.
- a subsystem diagram (and its kinds) hides the internal structure and exposes its services only through the public ports. For example, in Figure 9.6, the subsystem offers a service

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

specified by the provided interface iSetParameter, and needs a service specified by the required interface iDetector. Interaction points cannot be explicitly specified in class diagrams. In contrast, a class diagram uses visibility to hide or expose attributes and behaviors.

- in a subsystem diagram, an interface can be used to define a part type or to label the ball or socket associated with a port. The interface detail is seldom shown. In a class diagram, the detail of an interface is typically shown, especially when it is further extended. For example, the TrackDisplay interface in Figure 9.5 plays the "observer" role of the Observer design pattern. Its detail is suppressed in Figure 9.6, and the ball-and-socket notation is used instead.

- generalization is a class-level relation, and it is typically not used in subsystem diagrams. It is better to capture generalization in class diagrams.

- in a subsystem diagram, the detail of a class (as a type of some part) is suppressed. It is better to capture class features in a class diagram.

9.3 Modeling a Complete System

A system is composed of subsystems. Figure 9.7 gives a "simplified" system diagram for the AT91SAM9G45 evaluation board.

The evaluation board contains the AT91SAM9G45 chip and many other objects, such as an LCD, a touchscreen, USB ports, debug ports, an Ethernet port, on-board buttons, joysticks, and memory chips. In particular, the Test Program part has a dashed outline, indicating that it is not physically owned by the board; a user can load a different test program to the NAND memory chip.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

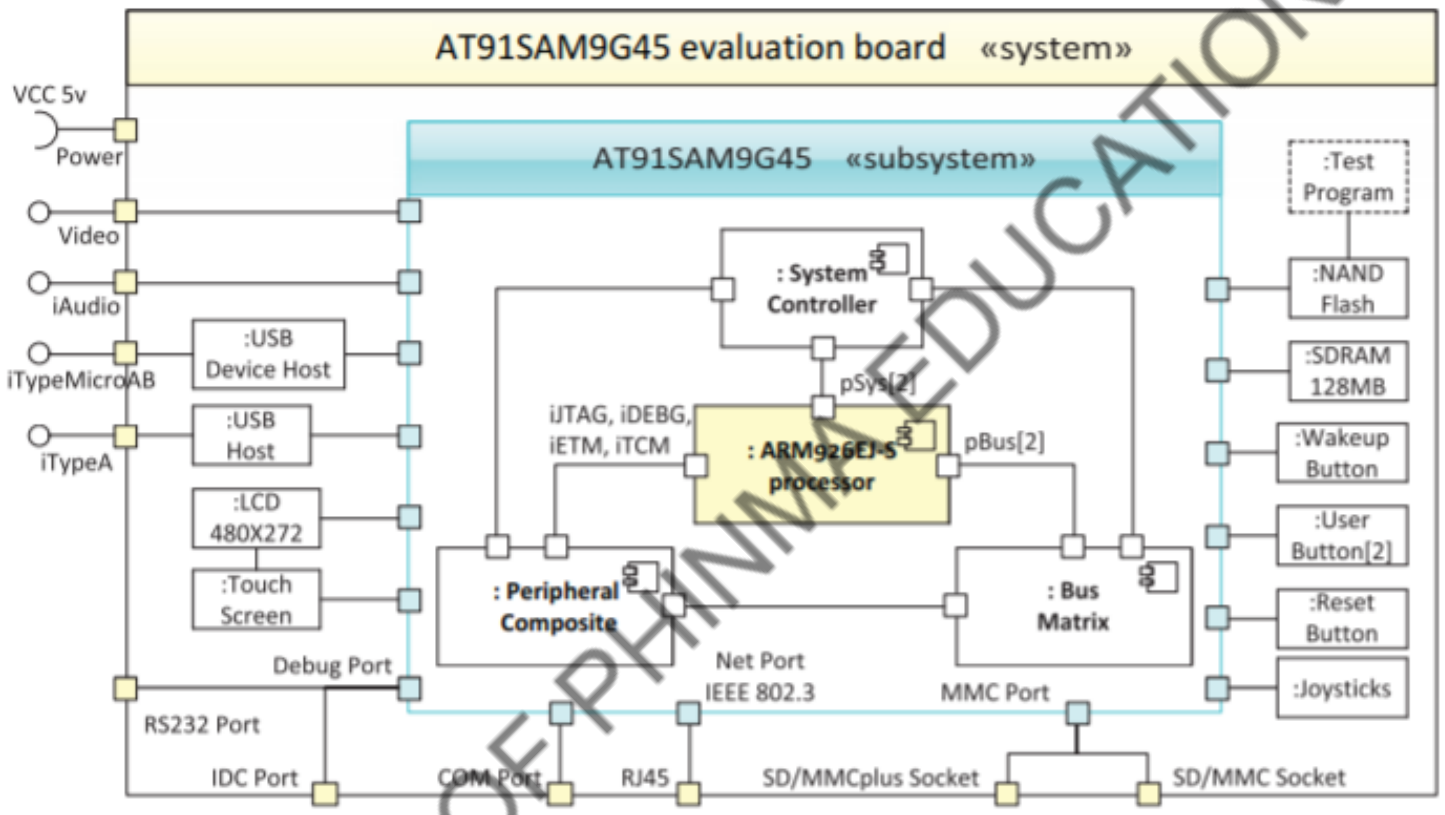


Figure 9.7
UML system diagram.

9.4 Deployment Diagram

A software artifact is a physical piece of information that is used or produced by a software development process, or by the execution of a system. Examples of artifacts include model files, source files, scripts, and binary executable files.

A node is a computational resource upon which artifacts may be deployed for execution. A node can be a device (e.g., a computer) or an execution environment (e.g., a Web services container). In UML deployment diagrams, a node is shown as a figure that looks like a three-dimensional view of a cube. Nodes can be connected to each other by associations or links, which indicate communication paths for the nodes to exchange signals and messages.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

A deployment is a dependency relationship from one or more artifacts to a node. Figure 9.8 gives an example deployment diagram, representing the execution architecture of a radar data processing subsystem. A deployment can be shown by placing an artifact within the node to be deployed. For example, the Tracker object is contained in the Tracker node. Alternatively, it can be shown by a dependency labeled «deploy» that is drawn from the artifact to the node. In Figure 9.8, the Controller object is deployed in the Controller node.

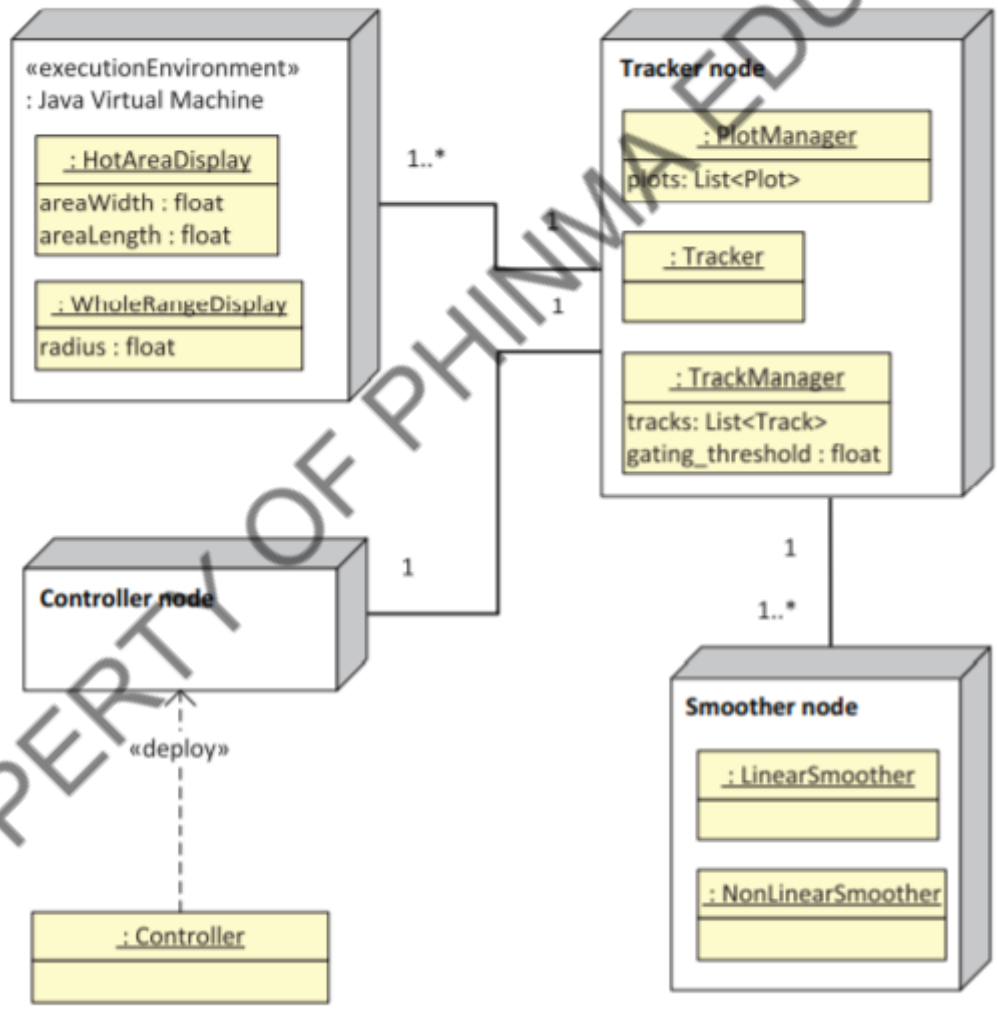


Figure 9.8
A UML deployment diagram.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

1) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

9.1 Find an example system to practice component modeling and subsystem modeling.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What are Modeling Components?	
	2 What are the Modeling Subsystems?	

3) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is a physical piece of information that is used or produced by a software development process, or by the execution of a system.
_____2. It is a computational resource upon which artifacts may be deployed for execution.
_____3. It is a dependency relationship from one or more artifacts to a node.
_____4. It defines a template for infinitely many object diagrams, specifying object relationships in a general sense.
_____5. It is simply a large-scale component, formed by assembling multiple related components. parts (protected port).

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. What is a component?

A component is an autonomous unit with its internals hidden and inaccessible other than through the provided and/or required interfaces. The provided interfaces specify a formal contract of the services that it provides to its clients, and the required interfaces specify what it requires from other components or parts of the system.

2. What is a subsystem?

A subsystem is simply a large-scale component, formed by assembling multiple related components. Thus, a subsystem diagram is visually the same as a component diagram, except that it can be optionally annotated by the stereotype «subsystem» or a subsystem icon (a fork symbol).

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #10

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

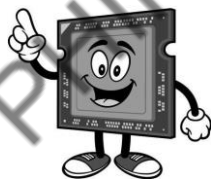
Lesson title: Fundamental UML Behavioral Modeling (Part 1)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand Use Case Diagram;**
2. **Understand Use Case Modeling.**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"Action is the foundational key to all success." Well, make way for your actions to build that success you are aiming for!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about the Architecture Modeling in UML. Now, we move on to the first part of Fundamental UML Behavioral Modeling. In this topic, we'll be covering up Use Case Diagram and Use Case Modeling.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is a Use Case Diagram?	
	2 What is Use Case Modeling?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

10.1 Use Case Diagram and Use Case Modeling

In software engineering, the term "actor" is used to refer to a role that a user can play when she/he/it interacts with a system. Here, a user can be a person, a company or organization, another system (hardware or software), or a physical environment.

Potential actors can be identified from the stakeholders of the system under consideration. However, many stakeholders are not actors if they never interact directly with the system. Moreover, while interacting with a system, a user may play multiple roles, thus acting as different actors.

The concept "use case" was originally introduced by Jacobson and was later developed by others. A use case of a system describes a main sequence of interactions through which one or more actors work with the system to accomplish a specific goal.

10.1.1 Use Case Diagram

In system requirements analysis, the relationships between use cases and actors are represented in one or more UML use case diagrams. A use case diagram modeling of an online social game system is shown in Figure 10.1.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

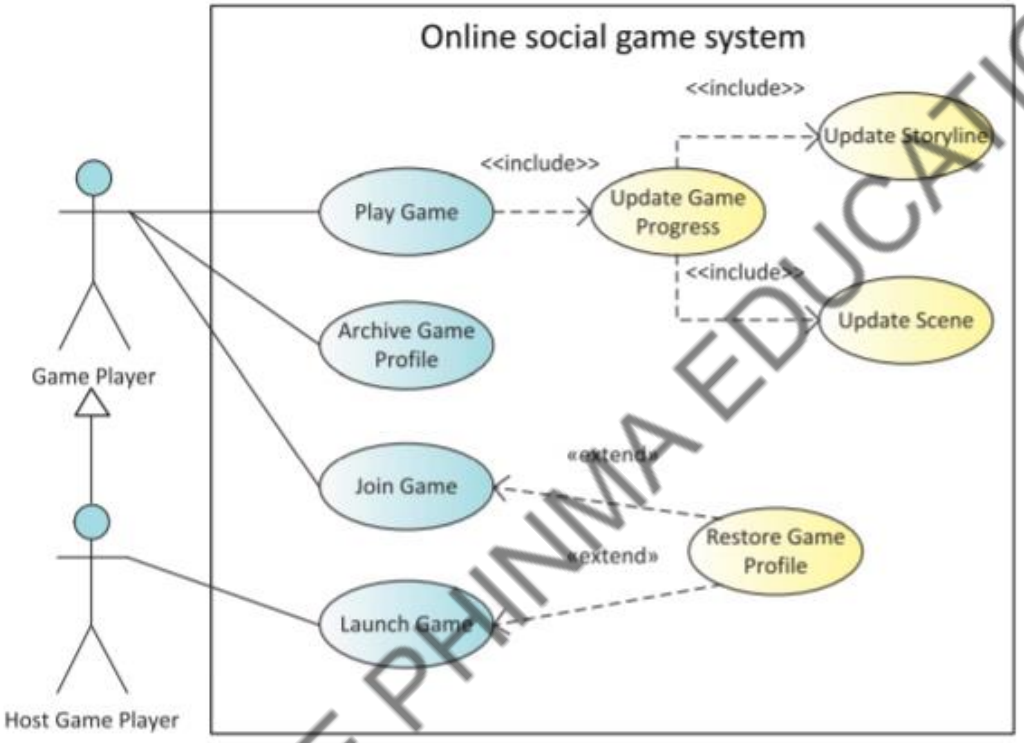


Figure 10.1
A use case model of a game system

As shown in Figure 10.1, the system under consideration is isolated from the rest by a system boundary (box). Stick figures are used to represent system actors. In this example, we have two actors: Game Player and Host Game Player, which are connected by a generalization relation (solid line with triangle head).

A use case is denoted by an oval labeled by an active verb phrase, representing a sequence of actions (activities or tasks) to be performed by a user and/or the system. An actor can be related to a use case by a communication link, over which information or control flows between the actor and the system. In Figure 10.1, there are three communication links between the Game Player and the system: the Game Player can join a game, play a game, and archive personal game profiles. The system has only one use case "Launch Game" connected to the Host Game Player, meaning that only the Host Game Player has the privilege to initiate a game. However, owing to the generalization relation, the Host Game Player can also conduct the three use cases linked to the Game Player, even though they are not explicitly linked to the

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Host Game Player. Generally, the set of use cases directly linked to external actors represent the main system functionality (or "functional requirements" in software engineering jargon).

Some use cases are not directly linked to an external actor. In such a case, they are typically connected with other use cases by two types of internal links: include and extend. In UML, stereotyping is an extensibility mechanism that allows designers to extend the vocabulary of UML in order to create new model elements from existing ones. A stereotype is rendered as a name enclosed by guillemets «». The UML standard predefines a few built-in stereotypes, including «include» and «extend». Both are used to label the dependency relations (dashed line with a stick arrowhead) between use cases.

X «include» Y means that X has to include Y in the flow of events: in the process of completing the tasks in use case X, tasks in use case Y will be completed at least once. An analogy in programming would be sub-procedure calls. The main reason of separating Y from X is for factoring commonality to achieve potential reusability.

X «extend» Y means that the base logic Y may invoke the extra logic X under certain conditions: X contains a few extra tasks that go above and beyond the tasks that are performed in Y. It is a dependency relation where the extending use case X defines logic that may be conditionally required during the execution of the base use case Y. An analogy in programming would be exception handling.

In Figure 10.1, the Play Game use case includes the Update Game Progress use case, which in turn includes the Update Storyline use case and the Update Scene use case. This means that the logic defined in the Update Game Progress use case will be triggered at least once for every input activity when a user is playing a game.

The Restore Game Profile use case extends both the Join Game use case and the Launch Game use case. Here «extend» is used because the Restore Game Profile use case is triggered conditionally by the system:

> *As a player is launching/joining a game, if his/her game profile exists (say, this is not the player's first time to play the game and the player archived his/her profile before), the system would automatically load the player's profile.*
> *If the Restore Game Profile use case is designed as a main functionality of the system—a task that a game player can directly initiate, then it could be connected to the Game Player via a*

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

communication link.

As another example, Figure 10.2 gives a use case diagram for a complex system that consists of two subsystems.

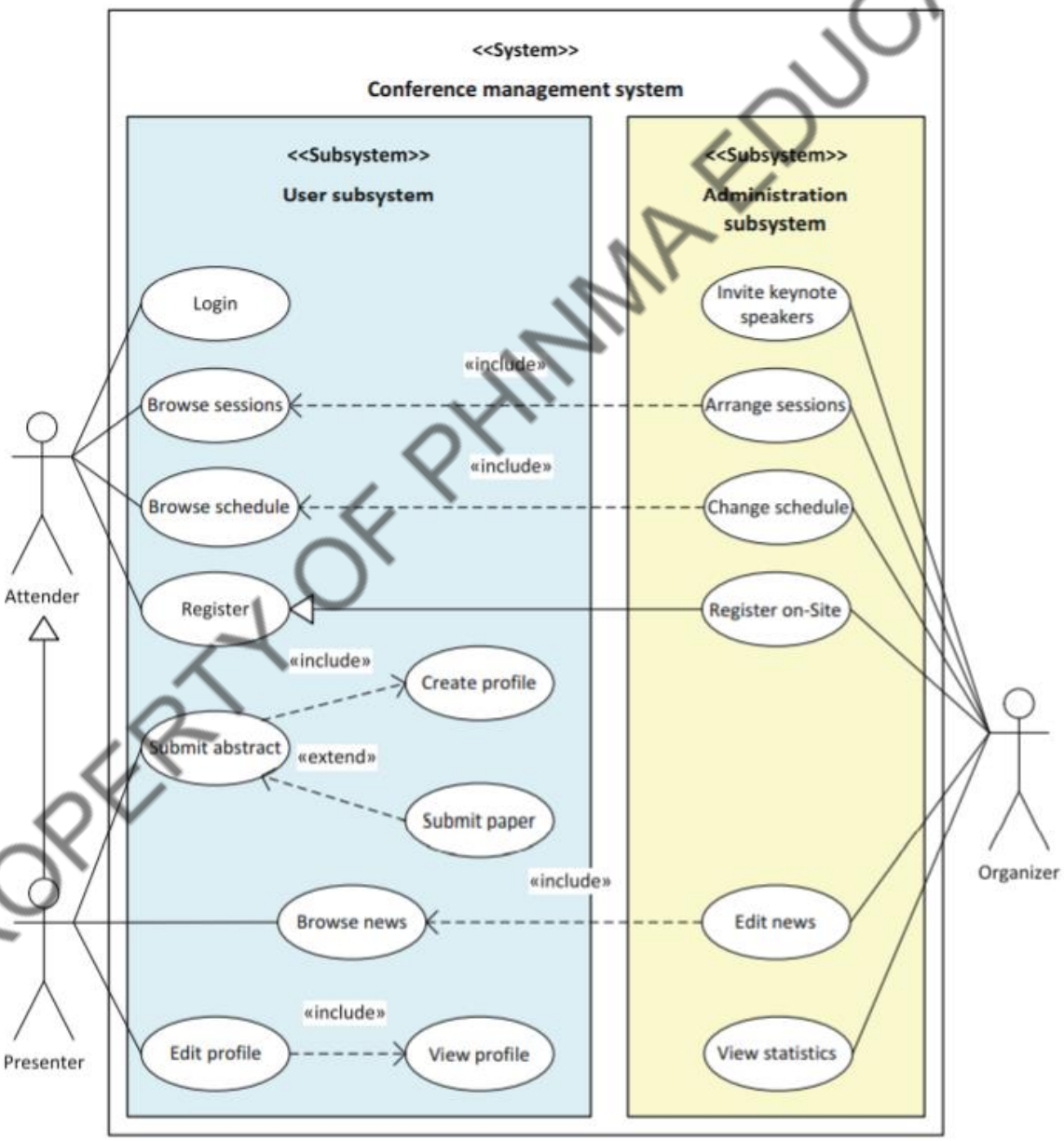


Figure 10.2 - A use case model of a system with subsystems

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

9.1.2 Use Case Descriptions

Use cases capture the scope of functional requirements of a system from the users' perspective, which is critical to the entire software development process.

- In requirements analysis, use cases allow stakeholders to virtually go through all the critical scenarios related to the system functionality. This can help lay out the business workflow, reduce ambiguity in requirements, elicit implicit requirements, and clarify the primary capabilities of the system.
- In the design phase, use cases suggest a potential system partition—objects closely related to a use case can be organized into one package.
- In system implementation, use cases can be used to plan development iterations—use cases with higher business values are implemented and evaluated with higher priorities.
- In user acceptance testing, use cases are important sources for generating test plans or test cases. Test cases can be created by enumerating all the potential sequences of user-system interactions.

However, as we can see from Figures 10.1 and 10.2, a use case diagram is nothing but a list of use case names offered to the potential users of the system. It captures nothing about the detailed course of actions that an actor or the system may perform. Software practitioners typically use a table-like template to document use case details. One such template is given in Table 10.1.

In this template, a use case is structured into three sections: (a) a section describing meta-level information such as the use case name, a unique ID, goal, scope, level, primary actor, trigger action, preconditions, and postconditions. Their meanings are given in Table 10.1; (b) a success scenario section describing the sequence of actions when everything goes well; and (c) an extension section containing a few branching blocks describing what to do when certain condition happens (i.e., when it is necessary to do special processing, or when something goes wrong).

The extension section of a use case is really a collection of stripped-down use cases. The greatest value of a use case lies not in the main success scenario, but in those alternative behaviors captured in the extension section, which need to be carefully considered in system design, testing, and validation.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Each entry in the extension section has

- an index referencing the relevant action step given in the success scenario section;
- a branching condition describing the trigger condition under which the system takes a different behavior; and
- a sequence of action steps to be performed (by the user or the system) in response to the branching condition

Table 10.1 - A template for documenting use cases

Use Case	#num: < the use case name as a short active verb phrase>
Goal in Context	< what the users want to accomplish by this use case>
Scope	< the system (or one of its subsystems) in which this use case belongs>
Level	< one of summary, primary task, subfunction>
Primary Actor	< the primary role who participates in this use case>
Preconditions	< the relevant situation description prior to the start of this use case>
Minimal Guarantee	< the relevant situation description upon an unsuccessful pass of the use case>
Success Guarantee	< the relevant situation description upon a successful pass of this use case>
Trigger	< the action upon the system that starts this use case>
Success Scenario	Action step
1	< the first step of the success scenario>
2	< the second step of the success scenario>
...	.....
i	< a reference to another use case related by «include»>
...	.....
n	< the last step of the success scenario>
Extension Step	Branching action (a step may have several branches)
2a	< the condition causing this branch>
	2a1: < a reference to another use case related by «extend»>
2b	< the condition causing this branch>
	2b1: < the first step of this branch> 2b2: < the second step of this branch>
	.....
	2bj: < the last step of this branch>

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The goal of an extension is either to complete the use case goal or to recover from the exception that caused this extension. An extension may rejoin the success scenario at a specified step or simply at the step where the branching happened, or may end with success or failure

A common pitfall in writing use cases is the inclusion of design specifics. Use cases are black-box requirements of system behaviors. A use case describes very well what the system should do, without saying how the system is to do it, which is a design issue. For instance, being a veteran in computing, a developer often has the tendency to use specific user interface terms (such as buttons, text fields, menus, mouse clicks, and list selections) to describe use case steps. Design specifics such as these should be avoided because the readers of use cases include not only the development team, but also other stakeholders who might be unfamiliar with the user interface terms. Moreover, including design details in the requirements document will later either limit your design choices or cause unnecessary revisions to use cases when the actual design changes.

10.1.3 Use Case Levels

Use cases are typically specified at three levels: summary, primary task, and subfunction. The three use case levels are compared in Table 10.2.

First, they are used for different purposes. A summary-level use case is generally used as an index table, just like the table of contents of a book. A primary task (aka user goal) level use case is used to lay out user-system interactions in detail, while a subfunction-level use case is used to describe the reusable interaction steps that may be shared by several primary-task-level use cases.

Second, use cases at different levels differ in their completion times. It may take a user a few hours or days to finish a summary-level use case. A primary-task-level use case can typically

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Table 10.2 - Comparisons of use case levels

Aspect	Summary Level	Primary Task Level (User Goal Level)	Subfunction Level
Purpose	Used as "Table of contents." One for each (sub)system	Used to describe base (backbone) logic	Used to describe small reusable logic
Level test	Takes a long time to finish, maybe a few hours or days	Takes one-sitting time to finish, maybe 2–20 min	Takes a very short time, say, less than 1 min
Diagram indicators	Covers nearly all the use cases inside the system or subsystem boundary	May have direct links to actor(s); May «include», or be «extended» by, other use cases	Has no direct link to the actor; its primary actor is the one linked to the base use case (by «extend» or «include»)
Scenario steps	Generally has "user steps" only; often contains references to primary task level use cases	Generally lists "user-system" steps in an interleaving manner; often contains references to other use cases (via «include» or «extend»	Generally has only a few "user-system" steps in an interleaving manner

be finished in a few minutes or less than 1 h, which is often referred as people's one-sitting time. A subfunction-level use case can be completed in a very short time—say less than 1 min.

Third, use cases at different levels may be revealed differently by their relationships with each other in the use case diagram. Each (sub)system boundary suggests one or more summary-level use cases. For example, an actor may follow a certain order to perform most or all of the use cases inside the (sub)system boundary. A use case inside the (sub)system boundary must be at the primary task level if it links directly to one or more actors. A primary-task-level use case may include or be extended by other use cases. Subfunction-level use cases usually have no direct links to actors, and they should be included by or extend several other use cases.

Fourth, the scenario of a summary-level use case normally consists of user action steps only, providing a sequence of references to several primary-task-level use cases. A primary-task-level use case contains "user-system" action steps in an interleaving manner; sometimes it may include or be extended by other use cases. A subfunction-level use case contains only a few "user-system" action steps in an interleaving manner; it can rarely include or be extended by other use cases.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Table 10.3 gives an example use case at the summary level.

Table 10.3 A summary-level use case

Use case	**#1: play the online social game**
Goal in Context	A host player playing the game
Scope	The game system
Level	Summary
Primary Actor	Host game player
Preconditions	The game is not launched yet
Minimal Guarantee	The game terminated
Success Guarantee	The game profile updated
Trigger	The host player starts the system
Description Step	Action
1	The host player launches the game «Launch Game»
2	The game system initializes the environment
3	A player Bob joins the game «Join Game»
4	The host player plays the game «Play Game»
5	Bob plays the game «Play Game»
6	Bob saves his profile and exits «Archive Game Profile»
7	The host player saves her profile and exits «Archive Game Profile»
Extension Step	Branching action
2a	Environment performance is too low
	a1: The game terminated

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

10.1 Explain the differences among the three levels of use cases.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is a Use Case Diagram?	
	2 What is Use Case Modeling?	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. This concept was originally introduced by Jacobson and was later developed by others.
_____2. It is denoted by an oval labeled by an active verb phrase, representing a sequence of actions (activities or tasks) to be performed by a user and/or the system.
_____3.-5. Three sections of use case.

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2											Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. Who are the potential actors of a system?

Potential actors can be identified from the stakeholders of the system under consideration. However, many stakeholders are not actors if they never interact directly with the system. Moreover, while interacting with a system, a user may play multiple roles, thus acting as different actors.

PHINMA EDUCATION
MAKING LIVES BETTER THROUGH EDUCATION

CPE 053 : Embedded Systems 1
Student Activity Sheet #11

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

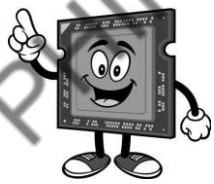
Lesson title: Fundamental UML Behavioral Modeling (Part 2)
Lesson Objectives:
At the end of this lesson, you should be able to:

1. **Understand what a Sequence Diagram is;**
2. **Understand how an Activity Diagram works.**

Materials:
Pen and paper

References:
Real-Time Embedded Systems, Design Principles and Engineering Practices, Xiaocong Fan, 2015

Productivity Tip:
"Action is the foundational key to all success." Well, make way for your actions to build that success you are aiming for!



A. LESSON PREVIEW/REVIEW

1) Introduction (2 mins)

In the previous lesson, we talked about the first part of Fundamental UML Behavioral Modeling which includes: Use Case Diagram and Use Case Modeling. Now, we move on to the second part where Sequence Diagram and Activity Diagram will be discussed.

2) Activity 1: What I Know Chart, part 1 (3 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Sequence Diagram?	
	2 What is an Activity Diagram?	

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

B.MAIN LESSON

1) Activity 2: Content Notes (13 mins)

11.1 Sequence Diagram

A system is composed of objects. A sequence diagram shows object interactions arranged in a time sequence.

A UML sequence diagram can be used together with a use case, depicting the objects involved in a scenario of the use case and the sequence of messages exchanged between the participating objects as they collaborate to fulfill the goal of the use case. This allows a developer to virtually step through the sequence of events to flesh out the high-level business process and validate the completeness of a use case.

UML sequence diagrams can also be used within the design context to explore a system or algorithm design, allowing a designer to trace the thread(s) of execution that may involve many system objects. Let us use Figure 11.1 to explain the sequence diagram notation.

First, a sequence diagram may be enclosed by an interaction frame, which is a solid-outline rectangle with a pentagonal area in its upper-left corner. The pentagonal area is used to display the keyword sd followed by the interaction name (and parameters, if applicable). Although the interaction frame is optional, sometimes it can be useful to cross-reference a sequence diagram by its name (say, composing simpler diagrams into a complex sequence diagram by references).

Inside the interaction frame there are five objects: a WebBrowser object named w, an anonymous WebServer object, an anonymous VideoServer object, a VideoPlayer object named aPlayer, and an anonymous VideoViewer object. Notice that three stereotypes—«Model», «View», and «Controller»—are used to specify the use of the Model-View-Controller design pattern explicitly.

Vertically, each object has a lifeline that represents the lifetime of its participation in the interaction. The order of events/actions along a lifeline denotes the order in which the events/actions will occur. In the example, there are five lifelines, two of which are created in the middle of the interaction.

Horizontally, objects can exchange messages. A message reflects either an operation call and

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

start of execution or the sending and reception of a signal. Each message has a sending event, which occurs at the sender object, and a receiving event, which occurs at the receiver object. Messages of the following forms can be used:

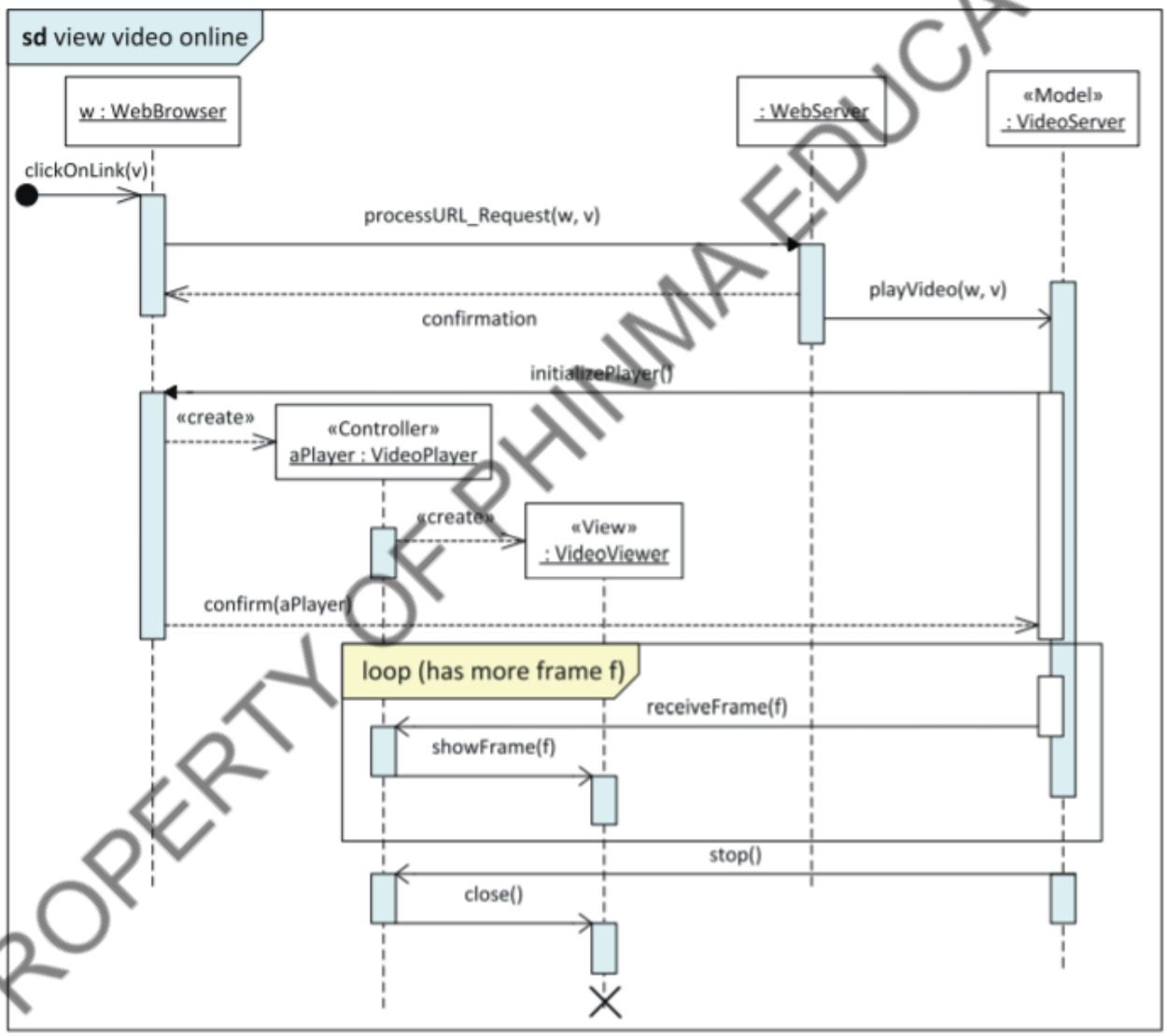


Figure 11.1
A UML sequence diagram for an online video player

• Asynchronous message: this represents a signal or asynchronous operation call and is shown as a line with an open arrowhead.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

• Synchronous message: this typically represents a synchronous operation call and is shown as a line with a filled arrowhead.
• Reply message: acknowledgment to a previously received message. It is shown as a dashed line with a stick arrowhead.
• Object creation message: a message indicating the creation of a new object. It is shown as a dashed line with an open arrow, usually labeled by «create».
• Object deletion message: a message indicating the destruction of the receiver object. It is shown as a cross at the bottom of a lifeline.
• Lost message: a message where the sending event is known, but there is no receiving event. It is shown as a small black circle at the arrow end of the message.
• Found message: a message where the receiving event is known, but the sending event is unknown or outside the scope of the current context. It is shown as a small black circle at the starting end of the message.

In the example, clickOnLink(v) is a found message; the objects VideoPlayer and VideoViewer are created by object creation messages; the object VideoViewer sends to itself an object deletion message; processURL_Request(w,v) and initializePlayer() are synchronous messages; confirmation and confirm(aPlayer) are return messages; all others are asynchronous messages. Now, you may realize why some of the objects have a name, while others are anonymous: a named object can be referenced by its name. For example, the VideoServer object needs to know who is the receiver object of its video frames.

In response to a message, the receiver object performs an appropriate operation, which is indicated by a thin rectangle drawn on top of the receiver's lifeline. If an object sends a message to itself, an execution rectangle can be overlapped on top of another to indicate a further level of processing. In the example, the VideoServer object has two operations performed in the execution of playVideo(w,v).

Sometimes a block of interactions can be grouped together by the notion of interaction fragments. A few kinds of interaction fragments are defined in UML, including alt and opt for choice of behavior depending on the truth value of the associated guard expression, par for parallel behavior, seq for sequential execution, loop for iterative execution, critical for critical regions, and ref for cross-reference. In Figure 11.1, there is a 'loop' fragment, saying that the VideoServer object keeps sending frames to aPlayer until there is no more frame left.

Figure 11.2 shows another sequence diagram named plot_gating, which has three nested interaction fragments. A more complicated example is given in Figure 11.3, which has a ref

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

fragment referencing the plot_gating sequence diagram.

Figure 11.2 depicts the sequence of operations for data processing in radar control systems. There are three parallel regions. It is worth noting that in the second parallel region there are five named rectangles with rounded corners on the Tracker lifeline. The named rectangle is called a state symbol. As the sequence of execution reaches a state symbol, the object enters that specific state and the object's behavior is relatively stable and predictable while it is in that state. A state could be the internal state of the corresponding lifeline object (say, a thread has states such as idle, running, and blocked), or it could be an imaginary system state from an external view (say, we can externally ascribe states to a traffic light system by the colors of the lights). In the former case, the object is called a stateful object, and its behavior can be further modeled by a state diagram

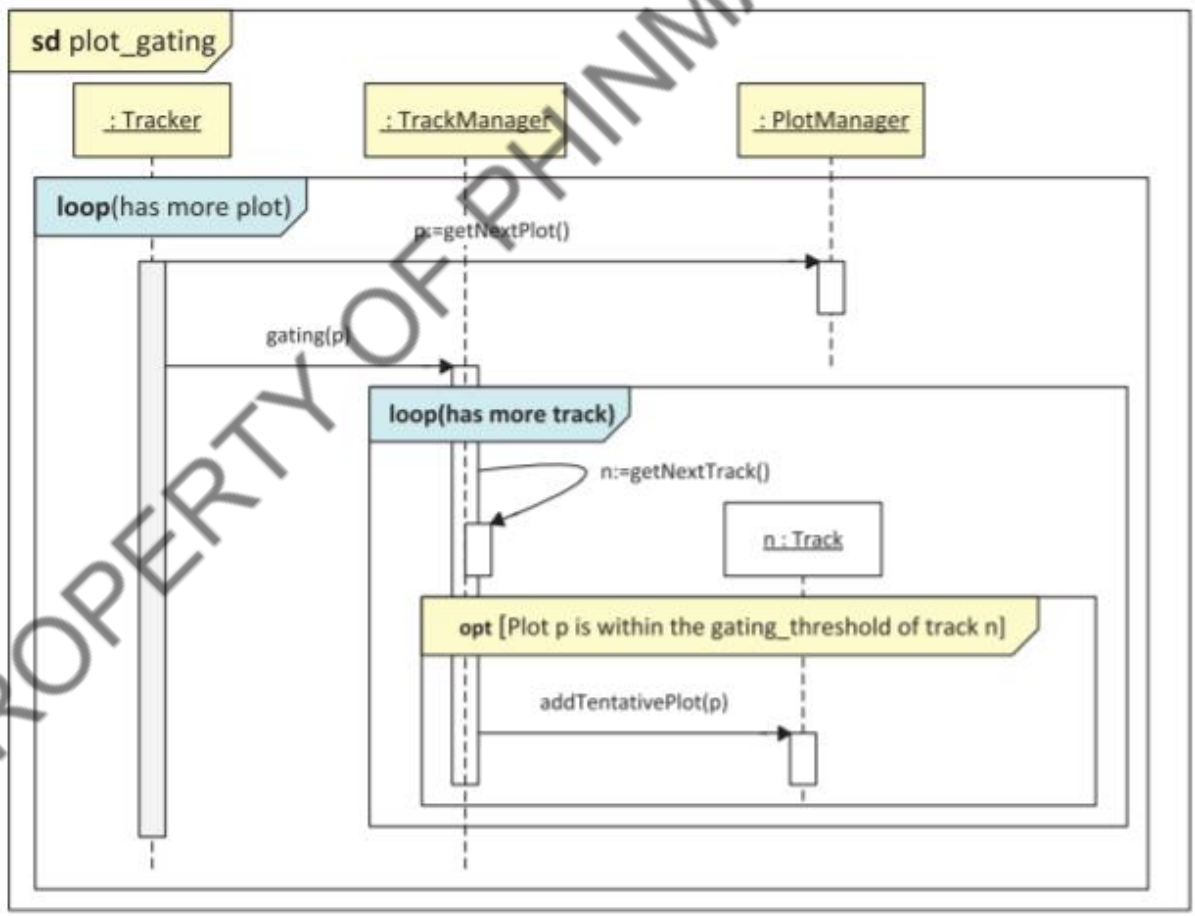


Figure 11.2
A UML sequence diagram with nested fragments.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The "stableness" of an object state can be described by state invariants—a concept similar to class invariants. A state invariant represents run-time constraints on the participants of the interaction. The constraints associated with a state should be valid while the object is in that state. In a sequence diagram, a state invariant is shown as text in curly brackets on the lifeline. For example, the invariant Tracker.smoother != Null says that the Tracker object should have a smoother while it is in the TrackAssociation state.

11.2 Activity Diagram

UML activity diagrams are used for modeling the control flows and data flows of a business process, an engineering workflow, or a procedural computation.

An activity represents a behavior that is composed of individual actions. Each action within an activity represents a single-step operation that may or may not be atomic, and a nonatomic action can be arbitrarily complex and even refers to another activity capturing a behavior at a finer level of granularity.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

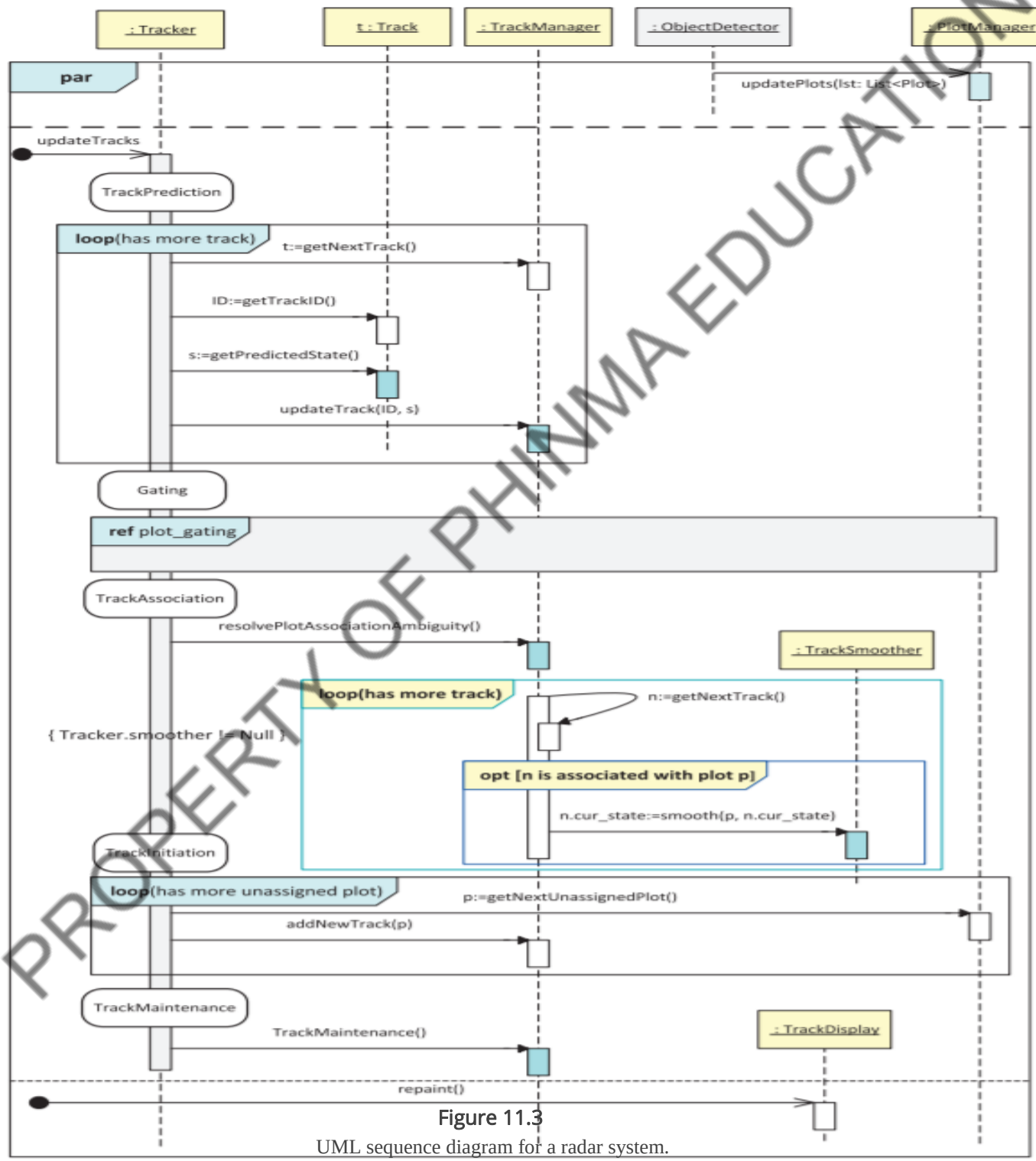


Figure 11.3
UML sequence diagram for a radar system.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Each activity diagram defines an activity that may contain three kinds of nodes—control nodes, object nodes, and action nodes:

(1) Control nodes. A control node is used as a traffic switch to coordinate the flows between other nodes. There are several kinds of control nodes, including

(a) initial node. This is a control node at which a flow starts when the activity is invoked. An initial node, notated as a solid circle, is a starting point for executing an activity.

(b) activity final node. This is a control node that stops all flows in an activity. In particular, it stops all executing actions in the activity. An activity final node is notated as a solid circle within a hollow circle.

(c) flow final node. This is a control node that terminates a single flow. It has no effect on other flows in the activity. It is notated as a circle with an "X" through it.

(d) fork node. This is a control node that splits a flow into multiple concurrent flows. A fork node has one incoming edge and multiple outgoing edges, which are either all object flows or all control flows. A fork node is notated as a thick line segment, with a single edge entering it and two or more edges leaving it.

(e) join node. This is a control node that synchronizes multiple flows. A join node has multiple incoming edges and one outgoing edge. The outgoing edge is a control flow if all the incoming edges are control flows; otherwise, it is an object flow. A join node is notated as a thick line segment, with two or more edges entering it and a single edge leaving it. A join node is an AND-type filter in the sense that the flow passes to the outgoing edge if and only if a flow has traversed each of the incoming edges.

(f) decision node. This is a control node that chooses between outgoing flows. A decision node is shown as a diamond with one incoming edge and several edges leaving it. The edges are either all object flows or all control flows.

An outgoing edge may have a guard expression. Which of the outgoing edges is actually traversed depends on the run-time evaluation of the guards on the outgoing edges. For a well-designed decision node, exactly one outgoing

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

edge is traversed at a time, which is warranted if (i) each of the outgoing edges has a guard expression, (ii) all the guards together form a complete set, and (iii) they are disjoint (no overlapping area).

A decision node may also have a behavior/operation without any side effects—say a Boolean expression involving some objects being processed by the activity. The output of the behavior is available to the guards on the outgoing edges, which are evaluated afterward. The behavior of a decision node can be specified by a note with the keyword «decisionInput» followed by an appropriate decision condition.

(g) merge node. This is a control node that brings together multiple alternate flows. A merge node has multiple incoming edges and a single outgoing edge, all of which must be either all object flows or all control flows. The notation for a merge node is a diamond-shaped symbol with two or more edges entering it and a single edge leaving it. Unlike the AND-type join node, a merge node is an OR-type filter in the sense that the flow passes to the outgoing edge if and only if a flow has just traversed one of the incoming edges.

(2) Object nodes. An object node is notated as a rectangle with a name inside, where the name indicates the type of the object node, or the name and type of the node in the form "name:type."

(3) Action nodes. An action node is notated as a round-cornered rectangle with a name inside. An action node represents the execution of a subordinate behavior, such as an invocation of another activity, a call to an operation, or manipulation of an object attribute.

An action node may have sets of incoming and outgoing edges that specify control flow and data flow from and to other nodes. When the set of incoming edges is not a singleton, it is treated as if the incoming edges are connected to the action node through an implicit join, implying that the action can be invoked until all the incoming edges have been traversed. If this is not desired, you should use a merge node between the incoming edges and the action node. When the set of outgoing edges is not a singleton, it is treated as if the action node is connected to the outgoing edges through an implicit fork, leading to parallel flows.

An action execution may need to process or produce certain objects. This is captured by pins, small rectangles attached to an action node, each specifying a typed parameter

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

expected by the action as an input (input parameter) or offered by the action as an output (output parameter). A parameter can be stream or nonstream. While an action must have all the nonstream inputs ready before its execution, it may contiguously accept values to its stream inputs or post values to its stream outputs while it is executing. In other words, values for streaming input parameters may arrive anytime during a single execution of the action, not just at the beginning, and multiple values may be posted to a streaming output parameter during the execution, not just at the end. A streaming parameter is marked by the string "[stream]" near the pin or one end of the corresponding edge if the pin is elided.

Control flow is used to model the sequencing of behaviors that does not involve the flow of objects. A control flow edge is notated by an arrowed line connecting a source action node to a target action node (there may be some control nodes in between), indicating that the target action cannot start until the completion of the source action. The source node is also called the predecessor node of the target.

Object flow is used to model the flow of data and objects in an activity. An object flow edge is typically denoted by an arrowed line connecting a source action node to an object node and another arrowed line connecting the object node to a target action node, indicting that the object flows from the source node to the target node (for further processing or transformation). Alternatively, when the "pin" notation is used, an object flow edge is notated as one arrowed line connecting an output pin and an input pin attached to the source action node and the target action node, respectively.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Figure 11.4 gives an activity diagram depicting the process for software changes. The activity starts with the Receive Change Request action, which has an incoming edge from the initial

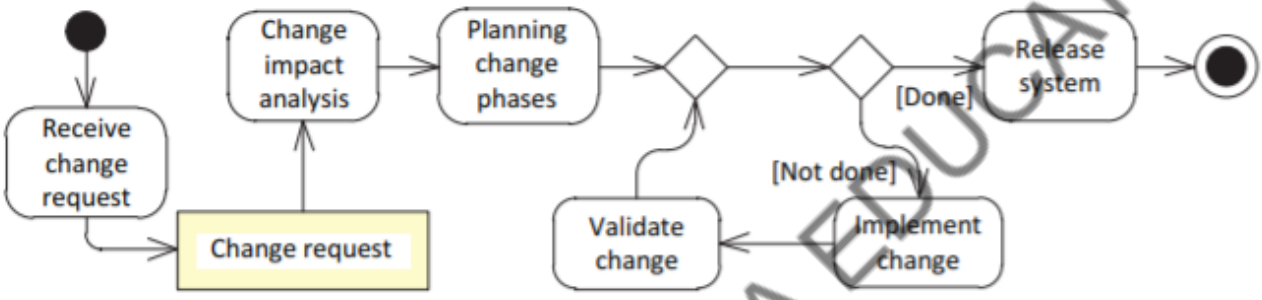


Figure 11.4
UML activity diagram example: software process for changes.

node. In between the actions Receive Change Request and Change Impact Analysis there is a Change Request object, which indicates that the Change Request object flows between the two actions. After Change Impact Analysis, the control comes to the action Planning Change Phases, which is followed by a merge node and a decision node. After the decision node, the guards of the two outgoing edges are evaluated to decide which edge is to be traversed. The new system is released and the whole activity is finished if all the changes have been implemented; otherwise the next change phase is implemented and validated.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

The activity diagram shown on the left in Figure 11.5 depicts how a video server offers services to its clients. The activity is enclosed within a frame with the keyword ad followed by the activity name. When a video server starts, it first initializes a pool of service agents. A service agent can be scheduled to serve at most one client at any time, and it is not available to serve a second client until it is released back to the pool. The server has a dedicated "listening" thread for accepting service requests from clients. Whenever a video server receives a service request, it passes the request object to the decision node, which, in this case, has a decision behavior specified by a note. At the decision node the server checks whether there are any service agents available. If there are none available, the request is rejected (maybe ask the client to try later) and the server continues to listen for client requests after a short sleep (some service agents could have been released in the meantime). If a service agent is available, the flow is split into two: one is used to handle the client request, and the other still acts as the "listening" thread after the pool has been updated.

Note that the activity comes to a flow final node after the completion of the action Handle Request. As explained before, the flow final node stops only this specific service thread. The activity of the video server is still executing, and most likely there are multiple service agents offering video services to their respective clients.

In addition, the Handle Request action is marked by a "rake" symbol, indicating that this action is a call to another activity. The activity diagram for Handle Request is shown on the right in Figure 11.5, and is enclosed by a rounded rectangle with a pin (small rectangle) on the border. Up to this point, we have shown three styles of activity diagrams, and this one is especially useful to explicitly specify the input and output parameters by pins.

The Handle Request activity accepts one parameter, which is a VideoRequest object as can be seen from the activity diagram on the left in Figure 11.5. According to the VidroRequest object, the Load Video action loads the requested video from the storage media. The "[stream]" annotation near the output pin of the action Load Video indicates that the action produces a stream of objects (video frames), the target of which is called an expansion region—a block enclosed by a rounded box with a dashed outline.

An expansion region may have one or more input collections, each containing elements of the same type and shown as a small rectangle with several tiny compartments placed on the boundary of the dashed box. The expansion region is executed once for each element in the input collection. If there are multiple input collections, they must have the same number of elements, and a value is taken from each of them for each execution of the region. An

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

expansion region may also have one or more output collections. On each execution of the region, an output value from the region is inserted into each output collection at the same position as the input elements.

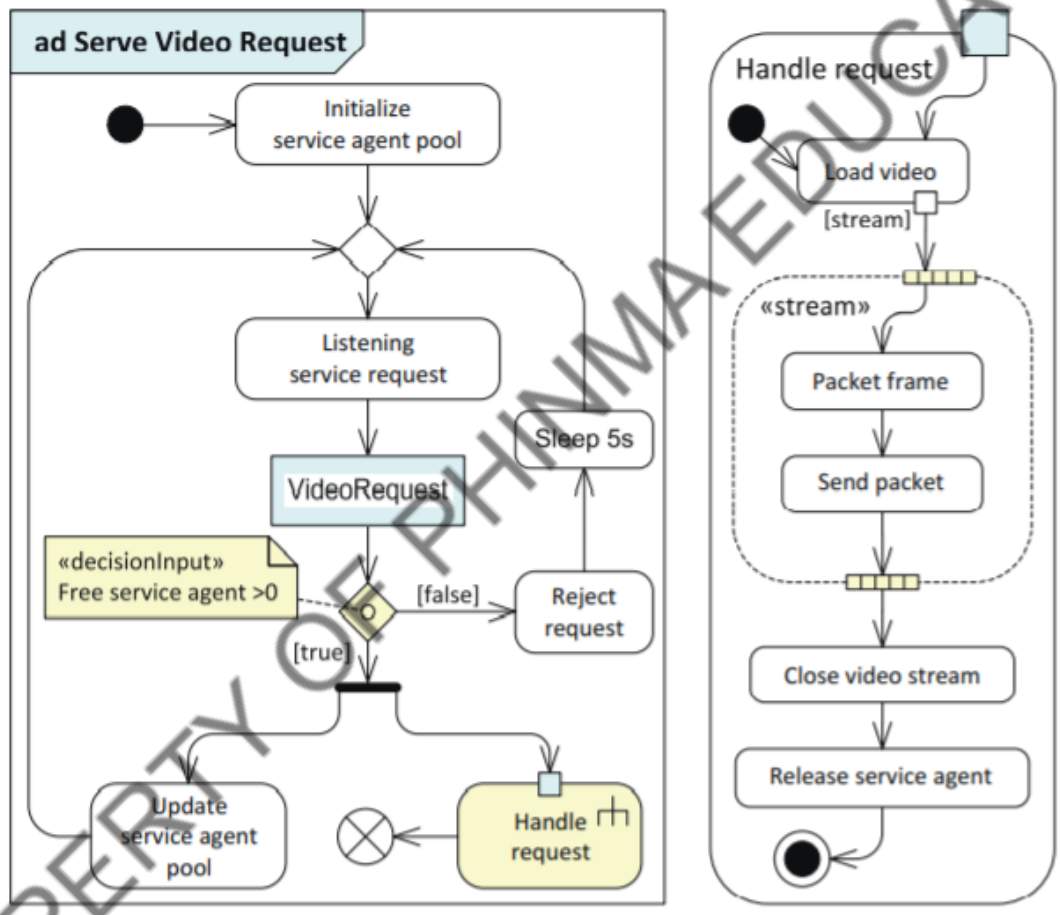


Figure 11.5
UML activity diagram example.

An expansion region can be in one of three modes: iterative, parallel, and stream:

1) Iterative mode: the element executions form a sequence, with one finishing before another can begin. This is the default mode.
(2) Parallel mode: the element executions may happen in parallel, or overlapping in time.
(3) Stream mode: like pipelining, the element executions form a stream. At any given time, it is likely that multiple or all elements are being processed at different stages by different actions in the region.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

Now, let us come back to the expansion region in the activity diagram on the right in Figure 11.5. It is in stream mode, which means the service agent may be sending a frame packet while it is packeting another frame. Note that the output collection in this example is not used to hold the packets to the client, which should have been done by the action Send Packet. The output collection in this case could hold flags for load analysis.

Figure 11.6 shows an activity diagram which is partitioned into several "swimlanes." A swimlane is a way to group actions performed by the same actor or in the same thread. In the example, there are five participants, where the Web browser, video player, and video viewer are on the client workstation, while the Web server and video server are in the cloud. In general, partitions can be hierarchical and multidimensional.

Since the action Handle Request performed by a video server is covered in Figure 11.5, Figure 11.6 focuses on the behaviors of the video player and video viewer, which are in the same flow (thread) generated by the Web browser object. In particular, the video player and video viewer execute actions in a loop: keep receiving and showing video frames until time-out happens. Alternatively, the actions Receive Frame and Show Frame can both be modeled using stream pins, like the action Load Video in the activity Handle Request.

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

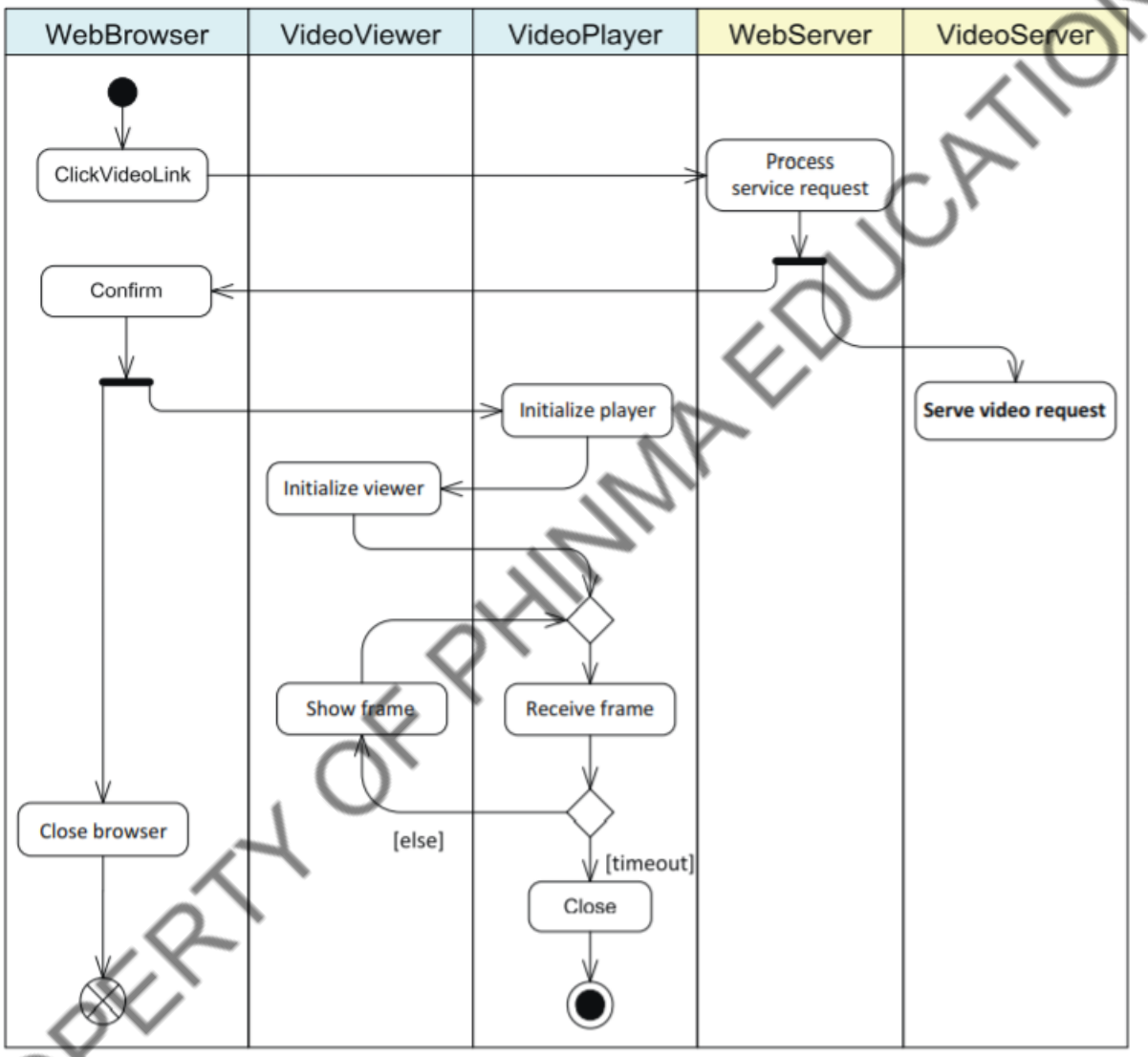


Figure 11.6
UML activity diagram: multiple participants

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

2) Activity 3: Skill-building Activities (with answer key) (18 mins + 2 mins checking)

11.1 In an activity diagram, how can you tell that an edge represents an object flow rather than a control flow?

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

3) Activity 4: What I Know Chart, part 2 (2 mins)

What I Know	Questions:	What I Learned (Activity 4)
	1 What is Sequence Diagram?	
	2 What is an Activity Diagram?	

4) Activity 5: Check for Understanding (5 mins)

Write the correct answer in the space provided below:

_____1. It is used to model the sequencing of behaviors that does not involve the flow of objects.
_____2. It is used to model the flow of data and objects in an activity.
_____3. This is a control node that brings together multiple alternate flows.
_____4. This is a control node that chooses between outgoing flows.
_____5. This is a control node that synchronizes multiple flows.

C. LESSON WRAP-UP

You are done with the session! Let's track your progress.

Period 1											Period 2										Period 3								
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Name: _____ Class number: _____
Section: _____ Schedule: _____ Date: _____

FAQs

1. What are UML activity diagrams used for?

UML activity diagrams are used for modeling the control flows and data flows of a business process, an engineering workflow, or a procedural computation.

2. What do you mean by activity in Activity Diagram?

An activity represents a behavior that is composed of individual actions. Each action within an activity represents a single-step operation that may or may not be atomic, and a nonatomic action can be arbitrarily complex and even refers to another activity capturing a behavior at a finer level of granularity.