Design and Development Processes

I. Introduction

In this chapter, we will begin by presenting a general approach to modular design. In specific, we will discuss how to organize software blocks in an effective manner.

Computer scientists pride themselves in their ability to develop quality software. Similarly electrical engineers are well-trained in the processes to design both digital and analog electronics. Manufacturers, in an attempt to get designers to use their products, provide application notes for their hardware devices.

The main objective of this class is to combine effective design processes together with practical software techniques in order to develop quality embedded systems.

Good software combined with average hardware will always outperform average software on good hardware. In this chapter we will introduce some techniques for developing quality software.

Learning Objectives:

- Understand system development process as a life cycle
- Take requirements and formulate a problem statement.
- Learn that an algorithm is a formal way to describe a solution
- Define an algorithm with pseudo code or visually as a flowchart
- Translate flowchart to code
- Test in simulator (Test → Write code → Test → Write code … cycle)
- Run on real board

In all walks of life, people develop strategies to problem solving. What this means is asking the right questions at the right time, and making the right decisions to develop an effective solution to the problem at hand.

Today, we're going to look at an approach to problem solving using a process.

A process is a sequence of steps we follow to design, develop, test, and they deploy an embedded system. This is called the life cycle of system development.

II. Product Life Cycle

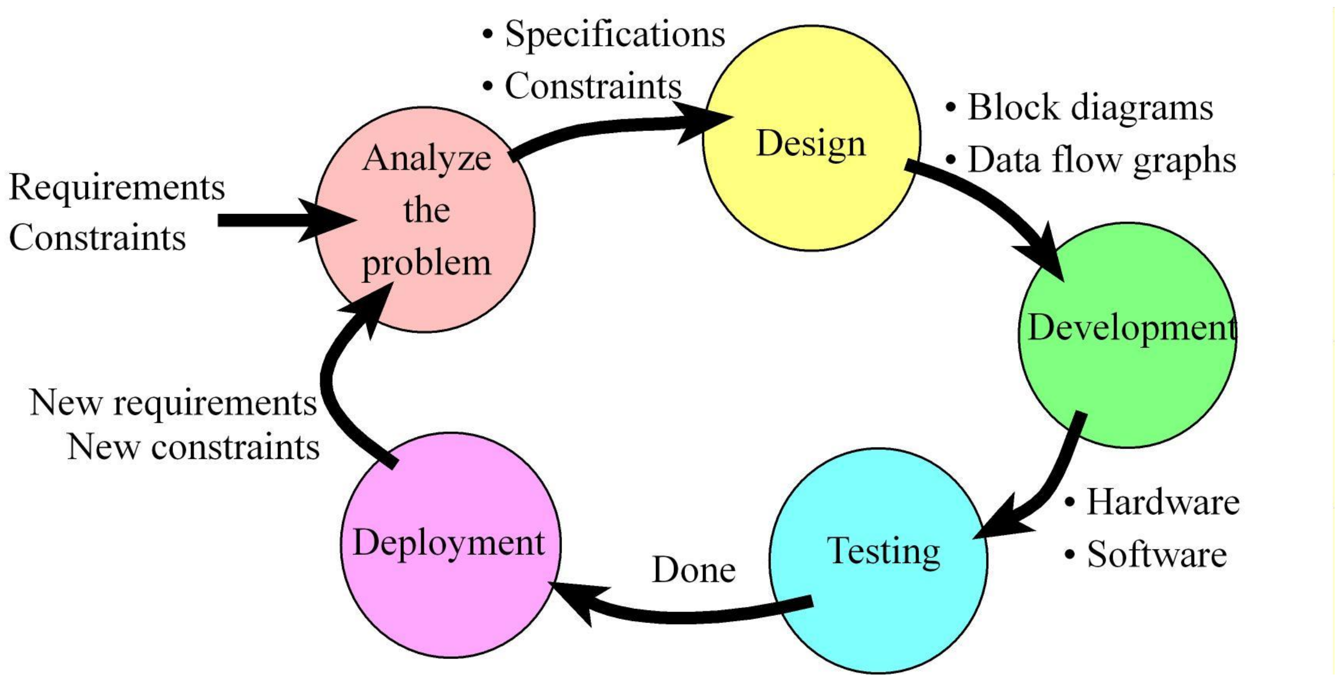
Design Processes

Product Life Cycle and Requirements

The first step of a life cycle is the requirements. The requirements is that we gather information about what the system has to be like. Then we design a solution.

Life Cycle

We will introduce the product development process in general. The development of a product follows an analysis-design-implementation-testing-deployment cycle. For complex systems with long life-spans, we transverse multiple times around the life cycle. For simple systems, a one-time pass my suffice.



Documentation is an important aspect of design. Some say *if you don't write it down, it didn't happen.* Memories are formed in the brain by observations, but long lasting accurate memories are forged by repeated observations.

We as engineers pass information across time or to one another by written documentation.
We did not add documentation as a separate step in the life cycle because each step creates an output, and that output must be explained with written documentation. For example, the output of the "testing" step is how you tested the device and testing results.

During the **analysis phase**, we discover the requirements and constraints for our proposed system. We can hire consultants and interview potential customers in order to gather this critical information.

A **requirement** is a specific parameter that the system must satisfy.

We begin by rewriting the system requirements, which are usually written in general form, into a list of detailed **specifications**. In general, specifications are detailed parameters describing how the system should work.

For example, a requirement may state that the system should fit into a pocket, whereas a specification would give the exact size and weight of the device.
For example, suppose we wish to build a motor controller. During the analysis phase, we would determine obvious specifications such as range, stability, accuracy, and response time.

There may be less obvious requirements to satisfy, such as weight, size, battery life, product life, ease of operation, display readability, and reliability. Often, improving the performance on one parameter can be achieved only by decreasing the performance of another. This art of compromise defines the tradeoffs an engineer must make when designing a product.

A **constraint** is a limitation, within which the system must operate. The system may be constrained to such factors as cost, safety, compatibility with other products, use of specific electronic and mechanical parts as other devices, interfaces with other instruments and test equipment, and development schedule. The following measures are often considered during the analysis phase of a project:

Safety: The risk to humans or the environment
Accuracy: The difference between the expected truth and the actual parameter
Precision: The number of distinguishable measurements
Resolution: The smallest change that can be reliably detected
Response time: The time between a triggering event and the resulting action
Bandwidth: The amount of information processed per time
Maintainability: The flexibility with which the device can be modified
Testability: The ease with which proper operation of the device can be verified
Compatibility: The conformance of the device to existing standards
Mean time between failure: The reliability of the device, the life of a product
Size and weight: The physical space required by the system
Power: The amount of energy it takes to operate the system
Nonrecurring engineering cost (NRE cost): The one-time cost to design and test
Unit cost: The cost required to manufacture one additional product
Time-to-prototype: The time required to design, build, and test an example system
Time-to-market: The time required to deliver the product to the customer
Human factors: The degree to which our customers like/appreciate the product

Requirements Document

The following is one possible outline of a **Requirements Document**. IEEE publishes a number of templates that can be used to define a project (IEEE STD 830-1998). A requirements document states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. Write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable.

1. **Overview**
  1.1. **Objectives**: Why are we doing this project? What is the purpose?
  1.2. **Process**: How will the project be developed?
  1.3. **Roles and Responsibilities**: Who will do what? Who are the clients?
  1.4. **Interactions with Existing Systems**: How will it fit in?

1.5. **Terminology**: Define terms used in the document.
1.6. **Security**: How will intellectual property be managed?

2. **Function Description**
2.1. **Functionality**: What will the system do precisely?
2.2. **Scope**: List the phases and what will be delivered in each phase.
2.3. **Prototypes**: How will intermediate progress be demonstrated?
2.4. **Performance**: Define the measures and describe how they will be determined.
2.5. **Usability**: Describe the interfaces. Be quantitative if possible.
2.6. **Safety**: Explain any safety requirements and how they will be measured.

3. **Deliverables**
3.1. **Reports**: How will the system be described?
3.2. **Audits**: How will the clients evaluate progress?
3.3. **Outcomes**: What are the deliverables? How do we know when it is done?

For more information search the internet for "requirements document template" or "IEEE STD 830-1998", and you will find a large variety of approaches to define a project before starting to build it. A wise person once said, "A project without a plan is never complete." The syntax of this document is not important; what is important is to have a clear plan/purpose for the project while building.

Data Flow Graph

When we begin the **design** phase, we build a conceptual model of the hardware/software system. It is in this model that we exploit as much abstraction as appropriate.

The project is broken into modules or subcomponents.

During this phase, we estimate the cost, schedule, and expected performance of the system. At this point we can decide if the project has a high enough potential for profit.

A **data flow graph** is a block diagram of the system, showing the flow of information. Arrows point from source to destination. The rectangles represent hardware components, and the ovals are software modules. ***We use data flow graphs in the high-level design, because they describe the overall operation of the system while hiding the details of how it works.***

Issues such as safety (e.g., Isaac Asimov's first Law of Robotics "*A robot may not harm a human being, or, through inaction, allow a human being to come to harm*") and testing (e.g., we need to verify our system is operational) should be addressed during the high-level design. A data flow graph for a simple position measurement system is shown in Figure 7.2.

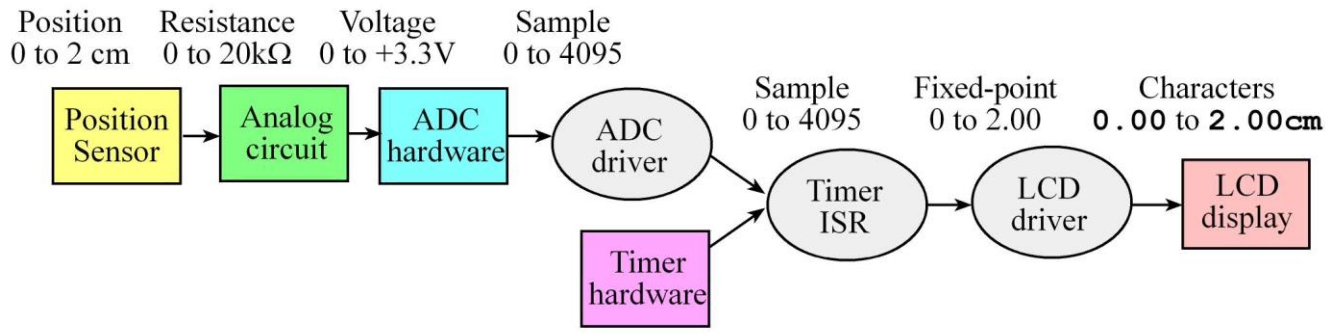


Figure 7.2. A data flow graph showing how the position signal passes through the system (this is the essence of Lab 14).

Call Graph

This is a low-level design

A preliminary design includes the ***overall top-down hierarchical structure***, the basic I/O signals, shared data structures, and overall software scheme.

At this stage there should be a simple and direct correlation between the hardware/software systems and the conceptual model developed in the high-level design.

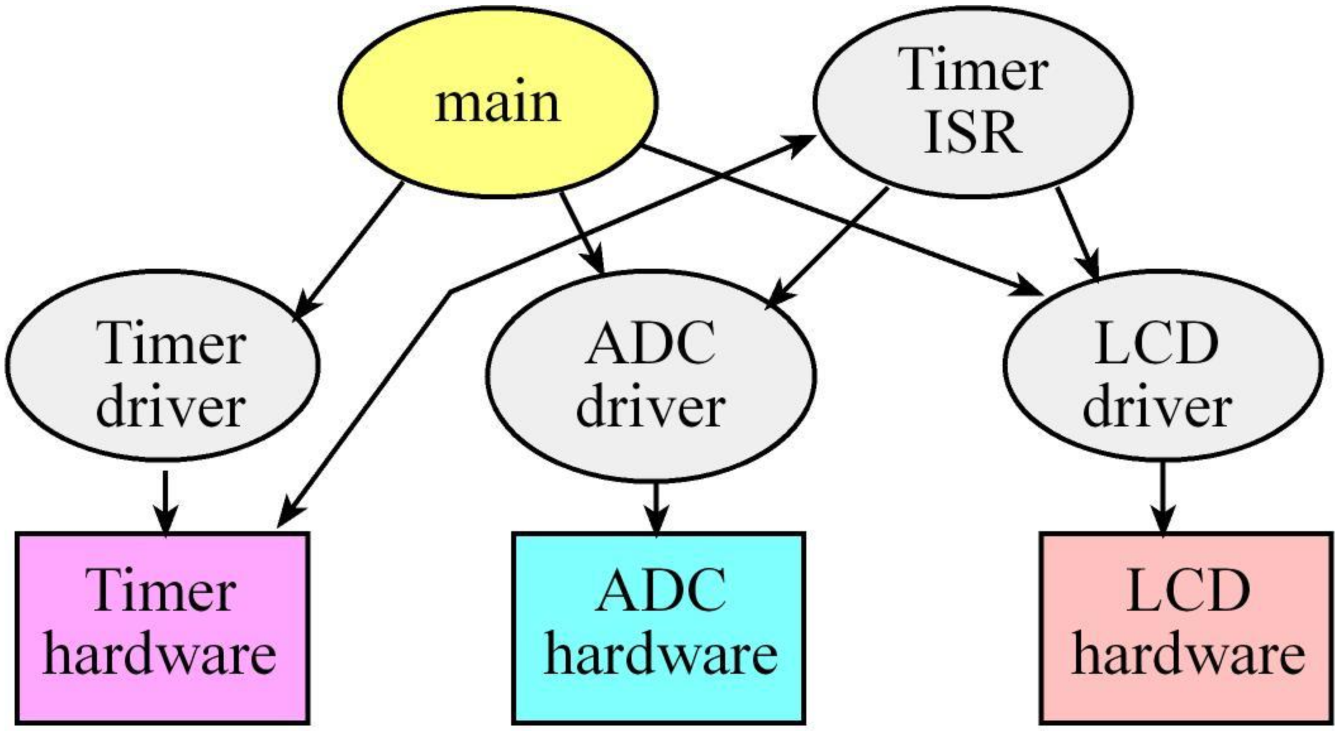
Next, we finish the top-down hierarchical structure and build mock-ups of the mechanical parts (connectors, chassis, cables etc.) and user software interface.

Sophisticated 3-D CAD systems can create realistic images of our system. Detailed hardware designs must include mechanical drawings.

It is a good idea to have a second source, which is an alternative supplier that can sell our parts if the first source can't deliver on time.

Call graphs are a graphical way to define how the software/hardware modules interconnect.

Data structures, which will be presented throughout the class, include both the organization of information and mechanisms to access the data. Again safety and testing should be addressed during this low-level design.



Implementation

The next phase involves **developing an implementation**. An advantage of a top-down design is that implementation of subcomponents can occur simultaneously.

During the initial iterations of the life cycle, it is quite efficient to implement the hardware/software using simulation. One major advantage of simulation is that it is usually quicker to implement an initial product on a simulator versus constructing a physical device out of actual components. ***Rapid prototyping is important in the early stages of product development.*** This allows for more loops around the analysis-design-implementation-testing-deployment cycle, which in turn leads to a more sophisticated product.

The next technological advancement that has greatly affected the manner in which embedded systems are developed is simulation. Because of the high cost and long times required to create hardware prototypes, many preliminary feasibility designs are now performed using hardware/software simulations. A simulator is a software application that models the behavior of the hardware/software system. If both the external hardware and software program are simulated together, even though the simulated time is slower than the clock on the wall, the real-time hardware/software interactions can be studied.

During the **testing** phase, we evaluate the performance of our system.

Maintenance is the process of correcting mistakes, adding new features, optimizing for execution speed or program size, porting to new computers or operating systems, and reconfiguring the system to solve a similar problem. No system is static. Customers may change or add requirements or constraints.

To be profitable, we probably will wish to tailor each system to the individual needs of each customer. Maintenance is not really a separate phase, but rather involves additional loops around the life cycle.

A top-down designer starts with a problem, conceives of a solution, procures the parts, builds a prototype, and then tests to see if it works.

Structured Programming

In this section, we discuss the process of converting a problem statement into an algorithm.

Later in the course, we will show how to map algorithms into software. We begin with a set of general specifications, and then create a list of requirements and constraints. The general specifications describe the problem statement in an overview fashion, requirements define the specific things the system must do, and constraints are the specific things the system must not do. These requirements and constraints will guide us as we develop and test our system.

Observation: Sometimes the specifications are ambiguous, conflicting, or incomplete.

There are two approaches to the situation of ambiguous, conflicting, or incomplete specifications. The best approach is to resolve the issue with your supervisor or customer. The second approach is to make a decision and document the decision.

Successive refinement, **stepwise refinement**, and **systematic decomposition** are three equivalent terms for a technique to convert a problem statement into a software algorithm.

We start with a task and decompose the task into a set of simpler subtasks. Then, the subtasks are decomposed into even simpler sub-subtasks. We make progress as long as each subtask is simpler than the task itself. During the task decomposition we must make design decisions as the details of exactly how the task will be performed are put into place. Eventually, a subtask is so simple that it can be converted to software code. We can decompose a task in four ways, as shown in Figure 7.4.

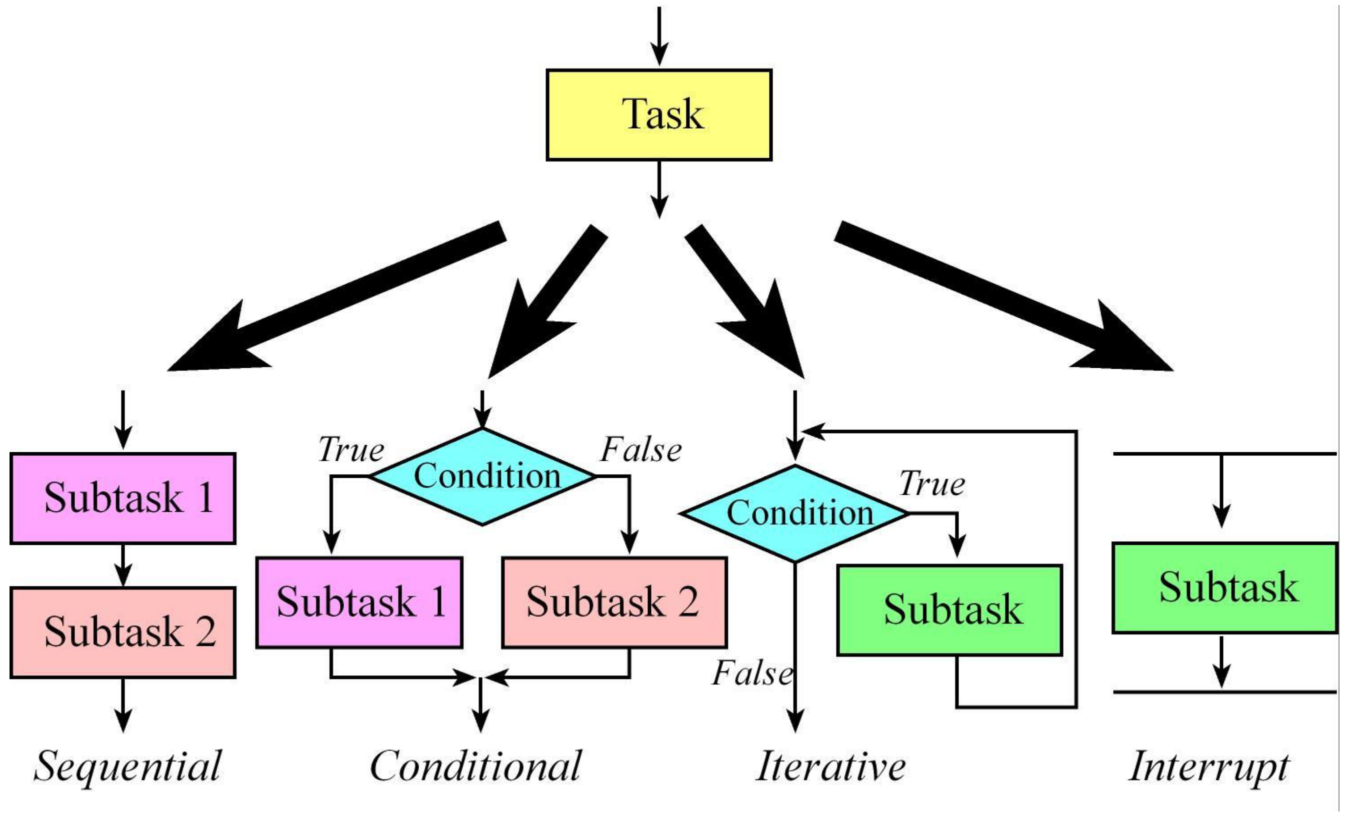


Figure 7.4. We can decompose a task using the building blocks of structured programming.

he **sequence**, **conditional**, and **iteration** are the three building blocks of structured programming. Because embedded systems often have real-time requirements, they employ a fourth building block called interrupts. We will implement time-critical tasks using interrupts, which are hardware-triggered software functions. Interrupts will be discussed in more detail in Chapters 9, 10, and 11. When we solve problems on the computer, we need to answer these questions:

What does being in a state mean?
List the parameters of the state

What is the starting state of the system?
Define the initial state

What information do we need to collect?
List the input data

What information do we need to generate?
List the output data

How do we move from one state to another?
Specify actions we could perform

What is the desired ending state?
Define the ultimate goal