Chapter – 9
Software for instrumentation and control applications

- Software is pervasive in electronic products such as televisions, video recorders, remote controls, microwave ovens, sewing machines, and cloth washers all have embedded microcontrollers.
- Software accounts for 50-75 percent of a microcontroller project.
- General methods to improve software are code generation, reliability, maintainability and correctness.

9.1 Types of software, Selection and Purchase

Types of software

Software is found in many different types of systems such as real-time control, data processing systems like payroll, and graphical systems such as games and CAD. Software can be divided into following types:

- System Software
  - Operating system, System drivers, Firmware etc.
- Programming Software
  - Compiler, Debugger, Interpreter, Linker, Text editor etc.
- Application Software
  - Industrial / Business automation, User interface, Games / Simulating software, Database, Image editing, Auto CAD, word processor etc.

Compiler versus Interpreter

Compiler	Interpreter
Source Code → Executable Code → Machine	Source Code → Intermediate Code →
Lots of time is spent in analyzing and processing the program.	Relatively little time than compiler
Result → Executable in form of machine specific binary code.	Result → Some sort of intermediate code
Computer (Hardware) interprets (executes) the resulting code.	Resulting code is interpreted by another program e.g. Java Virtual machine in Java.
Program execution is relatively faster.	Program execution is relatively slower.
Not required extra program to execute the code.	Requires extra program to execute intermediate code.
Standalone code	Not standalone code

Processed to develop software

1. **Algorithms**

- List of instructions or recipes for action
- Algorithms describe the general actions to be taken and consequently are independent of the specific programming language.
- Algorithms have the greatest effect on the utility, success, and failure of software
- Data structures provide another way of designing the processing architecture.
- Make a habit of collecting algorithms for your future programming efforts; when crisis hits, there will be no time for research.
- Understand each algorithm, its limitations and its boundary conditions
- Jack Ganssle writes: "It's ludicrous that we software people reinvent the wheel with every project… Wise programmers make an ongoing effort to built an arsenal of tools for current and future project…. Make an investment in collecting algorithms for future use. When a crisis hits there is no time to begin research."

2. **Languages**

- Software has many applications within embedded systems such as Firmware, Peripheral interface and drivers, Operating system, User interface, Application programs etc.
- May use variety of languages
    - Assembly language
    - High level language: Basic, C, C++, Java etc.

Assembly Language	High Level Language
Processor Architecture dependent and closer to hardware.	Processor Architecture Independent
Tedious because it requires steadfast attention to exacting detail.	Easier due to nitty-gritty details, structure and readability.
Best suited for small, simple projects with minimum memory, highest execution speed & precise control of peripheral devices.	Better for larger, more complex projects which require more memory and execute the code more slowly.

3. **Methods**

- Whatever language you choose, your objective will be to reduce complexity and improve understanding of the software.
- Design architecture may be Structured or Object oriented or CASE (Computer aided software engineering)
- Structured designs have strategy before starting to code; small modules with clear operational flow, easy debugging and testing.
- OOP can help by incorporating data abstractions, information hiding and modularity to aid structured design.
- CASE tools provide blend of environment, tools and language.
- Tools available are compilers, disassemblers, debuggers, emulators, monitors and logic analysers.
- Operating system and software libraries ease the task.

4. **Selection**

- The selection of a particular language depends on management directives, the knowledge and expertise of the software team, hardware and available tools.
- Function and performance depends on the speed and data path width of processor, memory (RAM and ROM), architectural features such as coprocessors, peripherals (ADC, timers, PWM, interrupt handlers), I/O communications, power-down modes and the level of integration.
- Choice of language also depends on manufacturers having following questions
    - Does the vendor provide reasonable documentation and support?
    - Does it provide toll-free telephone support and acknowledge application engineers?
    - Does it have liability support for life-critical systems.

5. **Purchase**

- Purchase the software after you have defined your software requirements and surveyed vendors for availability, reputation and experience.
- Some qualification of a vendor:
    - Acceptance testing
    - Review of vendor's quality assurance
    - Verification testing
    - Qualification report
- Furthermore, required documentations from a vendor:
    - Requirement specification
    - Interface specification
    - Test plans, procedures, results
    - Configuration management plan
    - Hazard analysis
- Don't buy cheap software tools just to save money! You will lose much more money in the long run from wasted time forced by delays and inadequacies of cheap tools.

9.2 Software Models and Their Limitations

1. Traditional software lifecycle / Waterfall Model

- Over the years many models have been proposed to deal with the problems of defining the critical activities and tying them together
- The first formally defined software life cycle model was the *waterfall model* [Royce 1970]
- The waterfall model is a software development model in which the results of one activity flowed sequentially into the next as seen as flowing steadily downwards (like a water) through different phases.
- Water cascades from one stage down to the next, in stately, lockstep, glorious order.
    - gravity only allows the waterfall to go downstream; it's hard to swim upstream
- The US Department of Defense contracts prescribed this model for software deliverables for many years, in DOD Standard 2167-A.

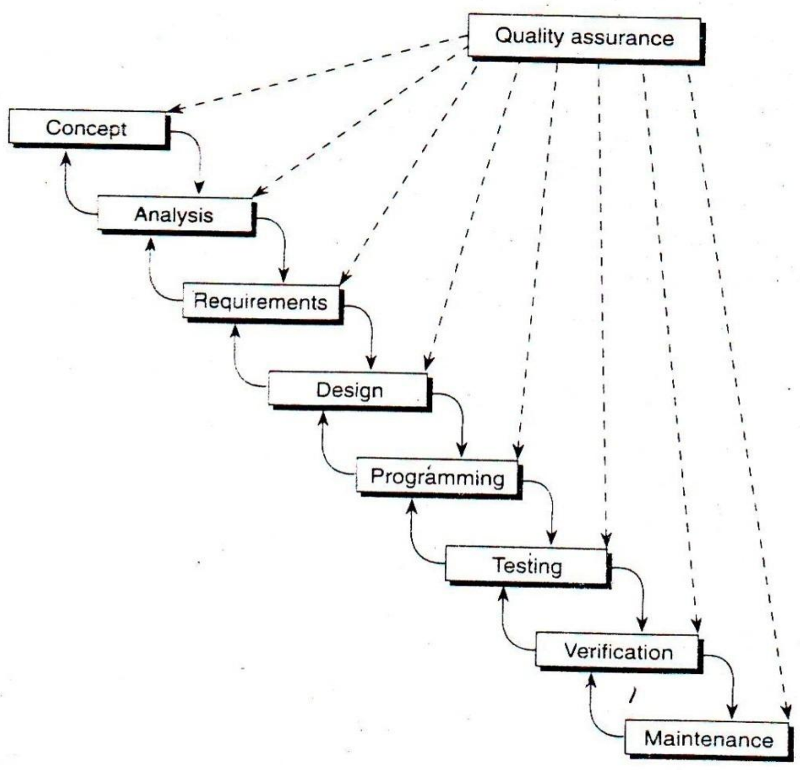

FIG. 11.1 The waterfall model of software development.

Quality Assurance

- Oversees each steps of model towards producing useful, reliable software rather than connection of modules
- Methods are necessary but not sufficient to produce useful, reliable software.
- First: Classify your system and its software according to any relevant standards.
- Second: Software development plan:
    - Hazard and fault tree analysis for life critical functions
    - Configuration management: ensures current and correct version is released.
    - Documentation
    - Traceability

Concept & Analysis:

- Problem described in human language.
- Model the concept with mathematical formulas and algorithms.
- Analyze the software within the framework of the system description and concept.
- Analyse on the interactions and interfaces between the software, the hardware and data inputs.
- Analyze should concentrate on where most problem occurs which include:
    - technical tradeoff
    - performance timing
    - human factor
    - hazard and risk analysis

  - Fault tree analysis: Graphical model of sequential and concurrent events that leads to failure of problems

Requirements

- Reduce the abstract intentions of the customer into realizable constraints.
- Tells what the software does
- Includes standards that the software must adhere to, the development process, the constraints, and reliability or fault tolerance
- Requirements may call for several, successive specifications:
  - General specification
  - Functional performance specification
  - Requirements specification
  - Design specification
- Needs constraints considerations such as memory, timing margin, hardware, communication, I/O and execution speed;
- Will the selected processor and associated hardware support the requirements?
- Can the software use these resources and satisfy the requirements.

Design

- Design tells how the software does its functioning.
- Specifies how the software will fulfill the requirements.
- Consider these software elements in preparing the design:
  - System preparation and setup
  - Operating system and procedures
  - Communication and I/O
  - Monitoring procedures
  - Fault recovery and special procedures
  - Diagnostics features
- Interfaces demands most attention communication format:
  - Polled I/O
  - Interrupt I/O
  - Synchronization between tasks
  - Intertask signaling
  - Communications of polling and queuing to avoid overrunning events
- Design can specify algorithms and techniques that optimize performance, management of memory.
- Design may reuse modules and libraries in an effort to improve productivity and reliability.

Programming

- The methods of programming – assembly language, high level languages.
- CASE tools are on the horizon
- Tools, language, methods

Source file (Assembly language mnemonics) → **assembler** → object file → **linker** → Binary machine code → **Burn** → PROM

Testing

A) Internal reviews
- By colleagues examine the correctness of software and can figure out mistakes and error in logic
- More than 50% of errors, can be found and correct by code inspection or audit

B) Black box testing:
- To ensure that input and output interfaces are functioning correctly without concerning what happens in software.
- Ensures the integrity of the information flow.

C) White box testing:
- Exercises all logical decisions and functional path within a software module.
- Requires intimate knowledge of software module
- Useful for determining bugs on special case
- Exhaustive testing is impossible; may take 100 of years to test each and every possible combination

D) Alpha and Beta testing:
- Type of black box testing in actual environment
    - Alpha testing - programmer collaborate with user
    - Beta testing - user isolated from programmer

Verification
- Debuggers, logic analyzer, in circuit analyzer in circuit debugger etc.

Maintenance:
- Require to control the software configurations: reports, measurement, personnel costs and documentation
- Plans for releasing software upgrades, to achieve consistency and continuity of the product.
- Cannot separate software maintenance from system concern

Disadvantages of Waterfall Model:
- Traditional view of software development
- Develop each component sequentially
- Not iterative - difficult to climb up the waterfall
- Focused on software rather than work design
- One phase is completed, documented and signed off before next phase for quality assurance
- Difficult to respond to changing customer requirement
- Software only available at late development schedule
- Based on hardware engineering model widely used in defense/aerospace
- Waterfall develops each component sequentially and usually does not iterate through more than a stage. In reality software seldom develops according to that model.

Benefits of Waterfall Model

- Managers *love* waterfall models
- Minimizes change, maximizes predictability
- Costs and risks are more predictable
- Highly documented
- Can be used for the projects whose requirements will not changeable
- Each stage has milestones and deliverables: project managers can use to gauge how close project is to completion
- Sets up division of labor: many software shops associate different people with different stages:
  - Systems analyst does analysis,
  - Architect does design,
  - Programmers code,
  - Testers validate, etc.

Problems with Waterfall Model

- Offers no insight into how each activity transforms artifacts (documents) of one stage into another
- Fails to treat software a problem-solving process
  - Unlike hardware, software development is not a manufacturing but a creative process
  - Manufacturing processes really can be linear sequences, but creative processes usually involve back-and-forth activities such as revisions
  - Software development involves a lot of communication between various human stakeholders
- Complex documentation requires highly profiled manpower
- Cannot be used for changing requirements
- Nevertheless, more complex models often embellish the waterfall,
  - incorporating feedback loops and additional activities

2. Prototyping Model

- In this model the developer and client interact to establish the requirements of the software.
- Accommodates the problem of changing requirements and make a subset of the software available early.
- Essence of prototyping is a quickly designed model that can undergo immediate evaluation.
- Define the broad set of objectives.
- This is follow up by the quick design, in which the visible elements of the software, the input and the output are designed.
- The quick design stresses the client's view of the software.
- The final product of the design is a prototype.
- The client then evaluates the prototype and provides its recommendations and suggestion to the analyst.
- The process continues in an iterative manner until the all the user requirements are met.
- Accommodates problem of changing requirements

- Helps to identify missing client requirements
- A prototype may take one of three forms
  - A paper model or computer based simulation
  - A program with a subset of functions
  - An existing program with other features that will be modified for your product.

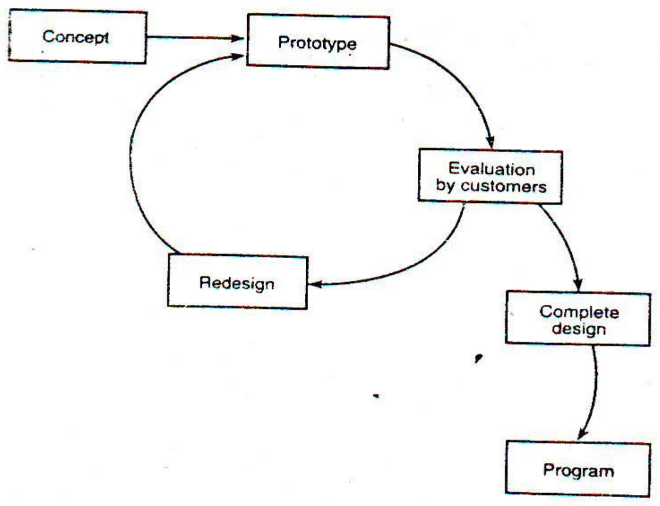

FIG. 11.4 Prototyping model for software development.

Advantages of Prototyping Model

The following are the advantages of Prototyping model:

- Due the interaction between the client and developer right from the beginning, the objectives and requirements of the software is well established.
- Suitable for the projects when client has not clear idea about his requirements.
- The client can provide its input during development of the prototype.
- The prototype serves as an aid for the development of the final product.

Disadvantages of Prototyping Model

The prototyping model has the following disadvantages.

- The quality of the software development is compromised in the rush to present a working version of the software to the client.
- The clients looks at the working version of the product at the outset and expect the final version of the product to be deliver immediately. This cause additional pressure over the developers to adopt shortcut in order to meet the final product deadline.
- It becomes difficult for the developer to convince the client as why the prototype has to be discarded.
- Sometimes prototype ends as final product which result in quality + maintenance problem
- Client may divert attention solely to interface issue
- Testing + documentation forgotten
- Designer tends to rush product to market without considering long term reliability, maintenance, configuration control

- Creeping featurism: The customer voices new desires after each evolution, and the project effort balloons.

3. **Spiral Model**

- Uses incremental approach to development that provides a combination of waterfall and prototyping model.
- Each cycle around the development spiral provides a successively more complete version of the software.
- Model allows flexibility to manage requirements control changes
- Used in proprietary application
- Spiral Model – risk driven rather than document driven
- The "risk" inherent in an activity is a measure of the uncertainty of the outcome of that activity
- High-risk activities cause schedule and cost overruns
- Risk is related to the amount and quality of available information. The less information, the higher the risk

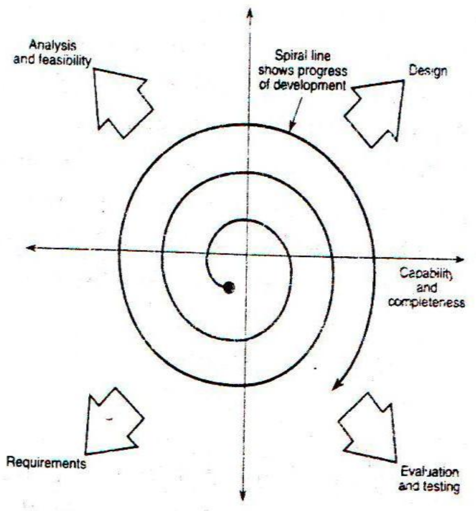
Analysis and feasibility
Spiral line shows progress of development
Design
Capability and completeness
Requirements
Evaluation and testing

FIG. 11.5 Spiral model of software development.

Spiral Model Strengths

– Introduces risk management
– Prototyping controls costs
– Evolutionary development
– Release builds for beta testing
– Marketing advantage

Spiral Model Weaknesses

– Lack of risk management experience
– Lack of milestones
– Management is dubious of spiral process
– Change in Management
– Prototype Vs Production

Metrics

- Objective understanding of the completion of the software at each stage or of its usefulness
- Software size, development time, personnel requirements, productivity, and number of defects all interrelate metric can define those relationships
- Cost and schedule are very poor metrics for producing quality software
- To rely on your estimates, you need to track the metrics honestly and record them carefully & consistently
- Good process model needs metrics to access performance and progress of software
    - Correctness, Reliability, Efficiency, Maintainability, Flexibility, Testability, Portability, Reuse, Utility, Size etc.

Process

- Process incorporate models of software development to generate useful, reliable and maintainable software
- Good process keeps statistics for feedback & improvement of the software development
- Software tools are integral to the software process; don't change them in midstream.
- Process maturity levels defined by the software engineering institute:
    1) Initial: Chaos or ad hoc process
    2) Repeatable: Design & Management defined
    3) Defined: Fully defined & enforced technical practices
    4) Managed Process: Feedback that detects and prevents problems, a control process
    5) Optimizing Process: Automating, monitoring & introducing new technologies

Software Limitations

- Not all problems can be solved
- Specifications cannot anticipate all possible uses and problems
- Errors creep into development in a number of ways
- Software simulation can predict only known outcomes
- Human error can occur in operating the software

9.3 Software Reliability

- Reliability is a broad concept.
    - It is applied whenever we expect something to behave in a certain way.
- Reliability is one of the metrics that are used to measure quality.
- It is a user-oriented quality factor relating to system operation.
    - Intuitively, if the users of a system rarely experience failure, the system is considered to be more reliable than one that fails more often.
- A system without faults is considered to be highly reliable.
    - Constructing a correct system is a difficult task.
    - Even an incorrect system may be considered to be reliable if the frequency of failure is "acceptable."
- Key concepts in discussing reliability:
    - Fault
    - Failure
    - Time

- Reliability develops complete plan
- Understands nature of bugs, their introduction and removal.

Guidelines to write reliable software

- Make each module independent
- Reduce the complexity of each task
- Isolate tasks from influences, both hardware and timing
- Communicate through a single well-defined interface between tasks

9.4 Fault Tolerance

- Fault tolerance concerns safety and operational uptime, not reliability.
- It defines how a system prevents or responds to bugs, errors, faults or failures.
- Use
  - Check sums on blocks of memory to detect bit flips
  - Watchdog timer: h/w that monitors a system characteristic to check the control flow and signals the processor with logic pulse when it detect fault.
  - Roll-Back-Recovery or Roll-Forward-Recovery
  - Careful design
  - Redundant architecture

9.5 Software Bugs and Testing

Phases of bugs

1. Intent
2. Translation
3. Execution
4. Operation

Intent

- Wrong assumption +misunderstanding
- Correctly solving wrong problems
- Viruses
- Slang-limits of operation to broadly or too narrowly defined

Translation

- Incorrect algorithm
- Incorrect analysis
- Misinterpretation

Execution

- Semantic error –does not know how command works
- Syntax error- rules of language
- Logic error- using wrong decision
- Range error-overflow /underflow error
- truncation error- incorrect rounding
- Data error –not initialing values, wrong error etc
- Language misuse-inefficient coding

- Documentation-wrong/misleading comments

Operation

- Changing paradigm
- Interface error
- Performance
- Hardware error
- Human error

Debugging

- Print statement
- Break points and watch values
- In circuit emulators ,in circuit debugger
- Logic analyzer
- White box testing
- Black box testing
- Grey Box testing
    - Having knowledge of internal data structure & algorithm for purpose of designing test case but testing is black box
    - Used in reverse engineering to determine instance, boundary value

Testing Levels

- Unit test
    - To test functionality of specific section of code at functional level
    - Building blocks work independently of each other
    - E.g. Class level testing in OOP
- Integration test
    - To verify interface between components
- System test
    - To test the whole system which is to be used

9.6 Good Programming Practice

- For useful, reliable, maintainable program we must make them readable and understandable. Good design and programming practices can make programs more readable.

A) Style and format

- program- to do something
    - To communicate designer's intent to other structure of program and comment.

Design:

- Documentation form begin
- Pseudocode before program
- Keep routine short
- Write clearly: don't sacrifice clarity for efficiency
- Make routine right, clear, simple and correct before making it faster

Comments:

- Readable and clear
- Should not be paraphrase of code
- Should be correct (incorrect comments are worse than no comment)
- Comment more than you think you need

Variables:

- Name properly
- Minimize use of global variables
- Don't pass pointer
- Pass intact values

B) Structured Programming

- Establish framework for generating code that is more readable useful, reliable and maintainable.
- Framework based on clearly defined modules or procedures, each doing one task well in a variety of situations.
- Modules can isolate device dependent code for simplicity and reuse.
- Large modules: divide among team for more productive and parallel effort.
- Use of libraries of modules and procedures load faster and resist inadvertent changes
- Structure programming encourages the installation and testing of one module at a time to simplify the verification of the software.

Points to be noted

- 90% of processor time is spent in executing 10% of code. Identify this 1%.
- Listen to customer while developing specification
- Prototype complex task on host computer and investigate their behavior.
- Design architecture for debugging and testing.
- Code small modules so that you can test and forget
- Code single entry and exit in routine
- Document carefully

C) Coupling and Cohesion

- Define tasks and design the modules.
- Modules should have a minimum of coupling or communication. If two tasks or processes communicate heavily, they should reside in the same module.
- Cohesion means everything within a module should be closely related that is they should stick together.

Cohesion:

- A cohesive module performs a single task
- Modules should have maximum cohesion
- Different levels of cohesion
  - Coincidental, logical, temporal, procedural, communications, sequential, functional
- Coincidental Cohesion

  - Occurs when modules are grouped together for no reason at all
- Logical Cohesion
  - Modules have a logical cohesion, but no actual connection in data and control
- Temporal Cohesion
  - Modules are bound together because they must be used at approximately the same time
- Communication Cohesion
  - Modules grouped together because they access the same Input/Output devices
- Sequential Cohesion
  - Elements in a module are linked together by the necessity to be activated in a particular order
- Functional Cohesion
  - All elements of a module relate to the performance of a single function

Coupling:

- Coupling describes the interconnection among modules
- Modules should have minimum Communication or coupling
- Data coupling
  - Occurs when one module passes local data values to another as parameters
- Stamp coupling
  - Occurs when part of a data structure is passed to another module as a parameter
- Control Coupling
  - Occurs when control parameters are passed between modules
- Common Coupling
  - Occurs when multiple modules access common data areas such as Fortran Common or C extern
- Content Coupling
  - Occurs when a module data in another module
- Subclass Coupling
  - The coupling that a class has with its parent class

D) Documentation and Source Control

- Documentation describes the overall system function.
- Documentation: first to begin and last to finish ensuring completeness and veracity.
- Back up source files: disks, CD, tape drivers
- Store multiple copies in separate location
- File storage is cheap but reconstructing lost data is expensive and impossible.

E) Scheduling

- Should record all efforts expended in current jobs to estimate future job
- Timing of meeting, planning, designing, debugging testing should be properly planned
- Give more time than required to debugging and testing.

9.7 User Interface

- The user interface is a major concern from a system viewpoint, and software plays a principal role in the interface.
- Good user interfaces require extraordinary attention to detail.
- The use interface can make or break an instrument; for instance, half of all hospital accidents are due to improper use of correctly operating equipment.
- Strangely, the higher the cost of the equipment, the lower the quality of the human-interface design.
- Common design issues for a user interface include response time, error handling and help facilities.
- The response time should have a reasonable interval and consistent variation application to the task.
- Error handling should be clear and give remedial action.
- Help facilities should be on line and context sensitive.
- Command sequences should be useful and consistent.

User Interface Guidelines

Action or Concern	**Comments**
Tune dialogue to user	Make it smooth and consistent. Use logical rather than visual thinking.
Make error messages meaningful	Let the user know what is going on
Provide help facilities	Let the user know what is going on
Verify critical actions	Help user understand consequences
Permit reversal of actions	Forgive mistakes
Reduce memory load	Don't compromise simple operations by extending them for infrequent ones
Display only relevant information	Remove static or redundant information
Deactivate commands	Fade out or clear from screen unused
Use good layout techniques	Use a modular format

User Interface Development

- Storyboarding and rapid prototyping are particularly suited for developing user interfaces because they are informal and fast.
- Try to develop the user interface with a top-down approach prototype which can take any of three forms
  - A paper prototype depicting user interaction
  - A working prototype with a subset of functions
  - An existing program that has all features but needs modification
- Creeping featurism can sidetrack development because prototyping concentrates on short term results and can miss long term concerns that may require substantial reworking in the future.
- Finally, you need cooperation from both customer and management.
- Management must support the goal of prototyping and effect of development schedule
- Customer must be committed to both evaluation and refinement of the prototype.

9.8 Embedded and Real Time Software

- Most of the software that runs or controls instruments is embedded and real time software.
- Real-time software is code that responds to current events in a timely manner.
- Embedded software is hardware specific; often a user interacts with a portion of the software system but does not have complete control over the source code.
- Software design concerns with occurrence and loading.
- Occurrence is the timing of events in real-time software. Events occur either synchronously and asynchronously.
- Loading is the measure of processor capacity; two metrics are utilization (amount of processing) and throughput (no. of I/O operations).
- Real time software has two components → operating system and device driver.
- Real time operating system (RTOS) controls the flow of events in priority scheduling mechanism.
- Performance, fault tolerance, and reliability are major concerns for embedded software which has following metrics:
  - Execution speed of the processor
  - Response time of the system
  - Data transfer rate
  - Interrupt handling: context switching and interrupt latency
  - Memory size
- Performance can measure with time I/O bus signals with an oscilloscope, logic analyzer, or performance analyzer.
- Fault tolerance defines how the software deals with misused resources and outright errors with various degrees as:
  - Limit on downtime of the system
  - Absence of catastrophic errors
  - Predictableness
  - Robustness
- Some types of failure that affect reliability include missed or incomplete tasks, deadlock, spurious interrupts, and stack overflow.

Chapter - 10

Case Study

Examples chosen from local industrial situations with particular attention paid to the basic measurement requirements, accuracy, and specific hardware employed environmental conditions under which the instruments must operate, signal processing and transmission, output devices:

a) Instrumentation for a power station including all electrical and non-electrical parameters.
b) Instrumentation for a wire and cable manufacturing and bottling plant
c) Instrumentation for a beverage manufacturing and bottling plant
d) Instrumentation for a complete textile plant; for example, a cotton foil from raw cotton through to finished dyed fabric.
e) Instrumentation for a process; for example, an oil seed processing plant from raw seeds through to packaged edible oil product.
f) Instruments required for a biomedical application such as a medical clinic or hospital.
g) Other industries can be selected with the consent of the subject teacher.

Preliminary

1. All students must team up for the case study and it is recommended to form a group of four to six students in a group. Once formed, the group cannot be reshuffled.
2. The group will take a request letter from the department. However, before approaching to an organization, students need to bring the responsible person's name and post for issuing the letter. The letter must be addressed accordingly.
3. The duration for the case study is for a month from the date of presentation. You need to submit the report. Apart from the new recommended design, you need to present the cost benefit analysis of the project.

During Visit

1. You need to understand the current process control system of the visited organization and describe the same in your own word in the report. List all the variables that are included in the process control system.
2. The systematic approach to understand the system must be presented with necessary block and detailing diagrams, if it is required.
3. Interview managers and the personnel who are directly involved in the current system and get to know the merits and demerits of the system.
4. Learn more from users and consumers who are directly participating and using the product of the visited organization. Comment on the product and recommend better option for the product in the present context, if you feel its need.
5. List down all the requirements needed to go for the improvised system.
6. Mention the cost of the current system.
7. Compare it to the latest system available in the market.

After Visit

1. Think and recommend the extra mechanism to provide a better solution the current problem.
2. Draw the block diagram of the newly recommended system. How does the current system adjusts the demerits discussed in item no 3 of during visit.
3. Include how the cost varies and what additional benefit you get with the newly proposed system in place.
4. Did you face a difficulty to go for the case study? How do you relate this with the real life situation?
5. Recommend what you feel like.
6. On the basis of above prepare a report on the case study.

The final report should present the instrumentation requirements in terms of engineering specifications, the hardware solution suggested, a listing of the particular devices chosen to satisfy the requirements, appropriate system flow diagrams, wiring diagrams, etc. to show how the system would be connected and operated.