CONTROL OF MECHATRONIC SYSTEMS
Theory and practice of control for packaging machines.

Davide Borghi
Tetra Pak Packaging Solutions

Cesare Fantuzzi
Università di Modena e Reggio Emilia
CONTENTS

Acronyms vii
Introduction ix

PART I THE ARCHITECTURE OF CONTROL SYSTEM OF AN AUTOMATIC MACHINE

1 Introduction to automatic machine 3

1.1 Automation system 3
1.2 Control patterns in industrial systems 4
1.3 Control functions in an automatic machine 7
1.4 A simple example: A handling system on rollers for handling of goods 8

2 Timing in Automation System 11

2.1 Nature of Computation for Industrial Automation 11
2.2 Real-Time Systems 12
    2.2.1 Real Time Operating System 13
    2.2.2 Task Preemption 15
2.3 Intertask communication 16
    2.3.1 Data Integrity in Communication Within a Process 16
    2.3.2 Design rules 18

3 Control Architecture 25
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Programmable Logic Controller (PLC)</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1 Rack</td>
<td>26</td>
</tr>
<tr>
<td>3.1.2 The CPU</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Execution of programs in PLC</td>
<td>27</td>
</tr>
<tr>
<td>References</td>
<td>28</td>
</tr>
<tr>
<td>4 PLC programming</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Principle of programming PLC</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Ladder Logic syntax</td>
<td>30</td>
</tr>
<tr>
<td>4.3 Examples</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Programming languages for PLC: International Standard IEC61131-3</td>
<td>32</td>
</tr>
<tr>
<td>References</td>
<td>33</td>
</tr>
<tr>
<td>5 The description a reactive behaviour of an mechatronic device</td>
<td>35</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>35</td>
</tr>
<tr>
<td>5.2 State Transition Logic</td>
<td>39</td>
</tr>
<tr>
<td>5.2.1 States and Transitions</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2 Transition Logic Diagrams</td>
<td>40</td>
</tr>
<tr>
<td>5.2.3 Example: Pulse-Width Modulation (PWM)</td>
<td>41</td>
</tr>
<tr>
<td>5.2.4 Transition Logic for the Process Control Example</td>
<td>43</td>
</tr>
<tr>
<td>5.2.5 Nonblocking State Code</td>
<td>44</td>
</tr>
<tr>
<td>5.2.6 State-Related Code</td>
<td>44</td>
</tr>
<tr>
<td>5.2.7 State Scanning: The Execution Cycle</td>
<td>45</td>
</tr>
<tr>
<td>5.2.8 Task Congruency: Universal Real-Time Solution</td>
<td>45</td>
</tr>
<tr>
<td>5.3 An example: the &quot;Gantry Crane&quot;</td>
<td>46</td>
</tr>
</tbody>
</table>
### ACRONYMS

<table>
<thead>
<tr>
<th>AC</th>
<th>Alternate Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>Motion Controller</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logical Controller</td>
</tr>
<tr>
<td>RMS</td>
<td>Root Mean Square</td>
</tr>
</tbody>
</table>
INTRODUCTION

Modern factories for the production and packaging of mass products, e.g. foodstuff and paper tissues, are organized in three main branches:

1. the processing branch, in which the product is made,

2. the packaging line, that prepares and wraps with packaging material every single piece of product to assure its sterility and integrity during the shipment and the final distribution to customers,

3. the factory logistic system, that handles the raw materials (packaging materials) and finished products in relation with the warehouse management, and for the truck loading and shipment.

Nowadays, product processing and packaging (i.e. manufacturing phases 1 and 2) have reached a high degree of automation, in which issues as energy consumption awareness, agile manufacturing and product customization are commonly fully addressed. For example, packaging lines are designed and deployed to allow a fast change in packaging size and pattern in order to respond promptly to production flexibility requirements. Notably, optimization in the electrical motors control of the machines in the packaging line has permitted a more efficient use of the energy during production.

In particular packaging machines have built to make and wrap with protective material goods, often foodstuff, for mass distribution and consume. Market of packaging machines was steadily increasing in the latter years, pushed by the request for production speed, product quality and lower cost of goods.

Such machinery employs electrical, mechanical, computer, and communication equipment in its design. An engineer who designs and maintains packaging machinery must be knowledgeable in both mechanical and electrical fields. In the past, packaging machinery manufacturers trained their own engineers in house, which took several years and many trial and error processes. Today’s rapid growth in the packaging machinery sector requires a more systematic and scientific approach in training engineers. Training engineers in mechatronics is that systematic and scientific approach. It incorporates both electrical and mechanical engineering knowledge under one umbrella. Training in mechatronics lets engineers
understand the mechanical as well as electrical requirements for design, operation, and maintenance of complicated machinery.

Packaging machinery requires precise, fast, and repeatable operation. Most new packaging machines utilize servo motors, in particular AC servo motor. The purpose of the motor is to provide controlled torque to move to a precise position the mechanical part of the machine, to reach the productive goal.

Servo motors employ digital control, which is versatile, repeatable, and reliable. In recent years servo motors are becoming more cost effective compared to other types of drives.

As servo motors continue to gain popularity in the world of packaging machinery, packaging professionals on the machinery side of things would do well to bone up on mechatronics. And if you’re a packaging machinery builder, as you consider new hires in your engineering department, lean toward those who can demonstrate that they’ve been subjected to a systematic and scientific approach to mechatronics training.

The aim of this book is to describe the process of developing a mechatronic approach to the development of the automatic machine control system.
PART I

THE ARCHITECTURE OF THE CONTROL SYSTEM OF AN AUTOMATIC MACHINE
CHAPTER 1

INTRODUCTION TO AUTOMATIC MACHINE

1.1 Automation system

The term Automation System identifies the technology that uses control systems to manage machines and processes, reducing the need for human intervention.

The automation is introduced to perform repetitive, complex or heavy operations where the environment is unsafe or unsuitable for human operators. Moreover, automation is employed to obtain a high quality and fast production process.

The automatic machine is a mechatronic device that performs automatic operations (i.e. where it is not required for the direct intervention of man in the processing cycle) to transform a raw material in a finished product through the use of energy.

Figure 1.1 shows a conceptual scheme of the parts of an automatic machine, that can be described as follows:

- **The Operational part**: it’s the part of the automatic machine that performs the required processing. It consists of mechanical components designed to interact with the product.

- **Control Unit Part**, that is the unit that governs the machine. Usually it consists of an electronic computer with special functions specifically designed to acquire signals from the sensors and provide control signals to the actuators that are located in the operational part of the machine (the electronic hardware, or simply the hardware, of the control unit). The control unit hardware runs the control software which implements the algorithms that supervises the operations of automatic machine.

- **Raw materials**, which are the rough parts that must be worked out by the machine to obtain the final product. Often the rough parts can be semi–finished products (e.g. intermediate products, such as preformed boxes to be filled) produced by other machines. In this case, as often happens, the machine is inserted in a production line.
INTRODUCTION TO AUTOMATIC MACHINE

Figure 1.1: The conceptual scheme of a automatic machine.

- **Finished product.** The products are processed by the machine automatically. These products can also be semi-finished products that are taken into input as crude product by another automatic machine.

- **sensors** are sensitive electronic components that allow the measurement of process variables within the operational part of the machine.

- **Energy.** The operational part requires sources of energy to carry out the necessary mechanical movements. The energy is always linked to a mechanical movement, and usually originates electrical, pneumatic or hydraulic, and subsequently processed by an actuation device (actuator).

- **Actuator.** It's a device that converts a primary energy (e.g. electric, pneumatic, etc.) into a mechanical energy. Electric motors are very often used as actuators for their flexibility and power efficiency.

1.2 Control patterns in industrial systems

In industrial processes, such as the filling of packages with liquid foodstuff, there are standard ways to treat particular control functions. We name them as *Control Patterns*, and they can be listed as:

- **Control of Batch Productions.** In Batch Production the product is processed stage by stage over a series of working area, such as ovens, mixing tanks, etc. (see Figure 1.2).

  Batch production is common in bakeries and in the manufacture of sports shoes, pharmaceutical ingredients, purifying water, inks, paints and adhesives.

  An example of industrial batch process is baking bread. A recipe in a cookbook states the following steps for making delicious loaf of bread:
1. Mix flour, water and yeast in a bowl.
2. Form the dough to a loaf.
3. Let the dough to rise for thirty minutes.
4. Bake the bread in an oven at 225°C for 45 minutes.
5. Let the loaf cool down.
6. Put the loaf in a plastic bag or eat it right away.

Batch processes are becoming more and more important in the chemical process industry. Batch processes are used in the manufacture of specialty materials, which are highly profitable. Some examples where batch processes are important are the manufacturing of pharmaceuticals, polymers, and semiconductors. In continuous processes grade and product changes, as well as start-up and shut-down phases can also be seen as batch processes.

As a second example, we can consider the manufacture of inks and paints.

A technique called colour-run is used in this industry. A colour-run is where one manufactures the lightest colour first, such as light yellow followed by the next increasingly darker colour such as orange, then red and so on until reaching black and then starts over again.

This minimizes the cleanup and reconfiguring of the machinery between each batch. White (by which is meant opaque paint, not transparent ink) is the only colour that cannot be used in a colour-run because a small amount of white pigment can adversely affect the medium colours.

The formal definition of a batch process in the batch control standard S88.01 is:

A process that leads to the production of finite quantities of material by subjecting quantities of input material to an ordered set of processing activities over a finite period of time using one or more equipment units.

From a control point of view a batch process combines the characteristics of continuous flow production with those of discrete, part based production. A batch can be viewed as a discrete entity, which during production moves between different production units. However, during the transition phase from one unit to the next unit and during certain operations within a unit, e.g. fed-batch operations, the batch is more naturally described as a continuous process.

Usually the batch process is at the beginning of a production line formed by automatic packaging machines, as it produces the food (pasteurized milk, chocolate bars, etc.) to be packed.
- **Control of discrete event system.** Discrete event dynamic system (DEDS) is a discrete-state, event-driven system of which the state evolution depends entirely on the occurrence of asynchronous discrete events over time, which are often unknown.

  Systems in general can be seen as compositions of elements, whose relations and interactions are governed by known laws (e.g. a mechanical system governed by Newton’s laws). A system can be described as dynamic when some external forces act on it. The effect of an external force on the system causes a change of the system state, which can be expressed in terms of internal parameters of the system, not necessarily observable (measurable), but causing possibly at a later time the change of output parameters.

  Based on the characteristics of the system’s components and their interactions, systems can be classified as discrete or continuous, time-invariant or time-varying, linear or nonlinear, deterministic or nondeterministic etc.

  The class of dynamic systems are so called Continuous Variable Dynamic Systems (CVDS), where the physical world is described by differential equations, and the variable that represents the state of the system can take a continuous range of values.

  On contrary in DEDS, the state is represented by a variable that can take a finite and number of values (i.e. represented by a discrete value) that can change at discrete instants of time, usually unknown in advance.

  In discrete event system the control actions depend by the occurrence of events. Example of event are:

  1. continuous value of process variable which is compared with some threshold (e.g. an analogic real value, such as the level of liquid in a Tank),
  2. an integer value that counts the number of occurrences of an event (e.g. the number of coins to be introduced into a distributor of soft drinks, the quantity of pieces in one cardboard pattern, etc.)
  3. a boolean logic value (e.g. mechanical end-stroke detected by an electrical switch, the presence of pieces in a basket).
  4. the expiration of a period of time set in a timer.

  Although similar to continuous-variable dynamic systems (CVDS), DEDS contains solely of discrete state spaces and event-driven state transition mechanisms.

- **Control of Continuous Systems.** A continuous-variable dynamic systems (CVDS) evolves in time with continuity. Usually the system dynamics is described using differential equations. In case the process evolves at discrete time (such as animal specimen evolution, that depends on annual breeding, or as computation of annual interest in a bank account), or it is discretize over time, the system dynamics is described using difference equations, or recurrence relations.

  It should be noted that *discrete time system* is a completely different concept that *discrete event system*. In the first, the time is the independent variable that takes into account in the system dynamics the evolution of the system versus the time, when the variable time is discretized because, for example, implementation of models into a computer. In the second case the evolution of the system happens only upon the event occurrence. In this latter case the time is only used to sign the precise moment of event occurrence.

  In control of continuous systems, the controller output can assume any real value in an interval assigned, or, more precisely for computer control systems, a set of quantized values in defined ranges of values.

  The control strategy is based on knowledge of a mathematical model of the process, carried out by means of differential or difference equations.
1.3 Control functions in an automatic machine

In general, the following components are present in an automatic machine:

- **The logical Control.** The control system of the machine is usually organized in an hierarchical way. Often the top level is implemented by a *discrete event system* that commands the activities of the machine split in operational steps. This part is often referred as the logical control of the machine.

  For example, the motion of a gripper carried by a cartesian robot (e.g. a robot that can move on the x–y–z spatial axes), can be described as a sequence of movements each of them done along an axis. A possible sequence is:

  1. Move the gripper along the axis x until the point A is reached.
  2. Move the gripper along the axis y until the point B is reached.
  3. Move the gripper along the axis z until the point C is reached.

  The supervision of this sequence, i.e. the move commands to the actuator and the acquisition of the position of gripper by means of sensors, is an example of logical control of operational sequences, and for this reason it is also called sequential control.

  The specifications of the logical control are often not defined in terms of performance time for a given variable, as in continuous control theory in terms of settling time of the position of the gripper. Those specifications are often written in terms of “if–then” rules:

  “If the user inserts the coin into the dispenser drinks and select the correct type of drink then the vending machine should deliver the soft drinks”

  Thus the logical controller should ensure the performing of one or more activity sequences, coordinating their evolution. The evolution of each sequence is dictated by occurrence of *events* associated with individual actions: an activation or the conclusion of an action, an occurrence of anomalies, the interactions with the machine operator (user), the result of a mathematical operation or the ending signal form a timer.

- **The continuous control of processes.** Example of continuous processes are: (a) temperature control of an oven or a sealing head, (b) level control of a filling machine, (c) the control of an electrical motor that drives a mechanical device, etc.

  The control of such continuous systems run in parallel tasks to the main operations of the logical control. Usually these routines are scheduled to run at fixed time, which is the sampling time of the digital control that they implement.

  In the area of packaging machine, often the control of a continuous process can be realized from a distinct control device from the device that hosts the logical control.

  In this case the the logical control demands the control supervision task to this device (e.g. the controller of a level of water in a tank) which works on a continuous control basis (i.e. the performance of the control system in measured in this case using the time, by example: the settling time of the output variable versus a step input of the reference variable).

  The continuous control and the logical control interact very closely, often using a hierarchical structure, in which the logical control is on the top level (i.e. supervises the process sequences), and the continuous control is on the bottom level (i.e. control loop to assure the performances of each sequence out the production steps).

  In summary, the control architecture from a logical point of view can be described using a top–down description:

  - The lower levels of the control system is often a continuous control, which interact strictly with the operational part of the machine, that modulates primary controls (level, temperature, ...) and interlocked controls (pumps, valves, ...).
– The upper levels (control cell) play logic control functions: supervision, startup and shutdown, control of the working sequences, management failures and emergencies.

- A particular case of continuous control is the **motion control** (MC). The machine operational procedures require that the product is handled by specific handling tools (SHT), such as grippers, cutters, pushers, etc. The motion of SHT is controlled by specific hardware called *Motion Controller* that handles the control of electrical motors, including the timing requirement concerning system dynamics, and the synchronization between different SHT that interacts to achieve the production target.

- **The user interface** (UI). The UI allows the machine operator to supervise the machine, more precisely he/she can: (a) start/stop the machine, (b) set production recipes (production parameters) and (c) identify and solve problems such as machine jam or breakdown (troubleshooting).

- **Communication to factory control level** (COM). Every automatic machine is included in a more complex process, a production line, that involve several machines at factory shopfloor level. This production line is supervised and monitored by a Manufacturing Execution Systems (MES), which is an information technology system that manage all the manufacturing operations in factories. A specific control function takes the responsability of communicate machine status, including production statistics, and receive production parameters and commands.

### 1.4 A simple example: A handling system on rollers for handling of goods

The automatic handling system is based on motorized rollers that through their rotation transporting parcels which must be selected according to the presence of a bar code applied on one side (see figure [1.3]).

Using a laser scanner is acquired the bar code applied, or it is detected the absence. This information is sent to a general control computer (supervision) which controls a series of mechanical traslo (urging devices) involved in pushing the parcel in correspondence of appropriate openings in the machine. In this way is performed the choice of parcels.

It should be emphasized that the position of the packs can be estimated from the calculation of the speed of rotation of the rollers of the roller conveyors according to the law kinematic \( p(t) = p_0 + v \cdot t \), where \( p_0 \) is the starting position detected by a photocell, \( v \) is the tangential speed of the rollers and \( t \) is the time measured internally by the computer via an internal clock, or timer.

We can describe the system using a block scheme using an hierarchical structure, in which the system is decomposed in modules. Each module correspond to a precise system functionality, which is studied and developed as a subsystem isolated from the rest of the components.

In this way, we apply a decomposition principle to the system development that permit to tackle the complexity of modern automatic machine.

A possible system decomposition is shown in figure [1.4] which components can be described as follows:

- the control system of rollers (one for each roller) has the task to drive each roller at a defined speed.
- A laser scanner that reads the bar code applied on the package.
- A traslo mechanical system that performs the thrust of the package in correspondence to the disposal branches.
- A photocell system that allows the detection of position of the package.
- a central computer for collecting information and for the supervisory control
A SIMPLE EXAMPLE: A HANDLING SYSTEM ON ROLLERS FOR HANDLING OF GOODS

Figure 1.3: The automatic handling system.

Figure 1.4: A functional decomposition of the automatic handling system.
2.1 Nature of Computation for Industrial Automation

A system for processing data for office application is designed to read input data, apply a particular algorithm and terminate. In this case the processing time is not critical to the effectiveness of the software application, but only to limit the waiting time of the user.

On the contrary, in an automation system the controller must take over continuously a physical process, therefore the computation is reactive in nature, that is, the computation is carried out with respect to real-world physical signals and events. To achieve this goal, industrial computers must have extensive input-output subsystems that interface it with physical signals, moreover, the controller repeats forever the following steps (see figure 2.1):

1. **Acquisition** of sensory signals from the physical process by means sensing devices (e.g. temperature sensors, position sensor, etc.)

2. **Execution** of the control algorithm, that computes the control actions on the basis of the sensorial signals.

3. **Application** of the control actions to the physical process through actuation devices (e.g. electrical motors, pneumatic actuators, etc.)

We call $T_{cycle}$ the total duration of the control cycle formed by the three above steps: **Acquisition**, **Execution** and **Application**. The length of $T_{cycle}$ depends on the speed of the control hardware and the complexity of the control
algorithm to be executed, and in general it should be limited after an analysis of the response time required to keep the system under control.

\[ p_{CPU} + p_s = 2 \cdot T_{cycle}v + 0.5\frac{v^2}{d} \]

therefore a long \( T_{cycle} \) affects the promptness of the system to react against the observation of an event.

When there are a large number of such tasks to be executed on a single processor, appropriate scheduling strategies are required (multitasking system). Since the number of tasks is mostly fixed, simple static scheduling policies are generally adequate.

Although computational demands are not generally very high, the computation is critical in nature. Indeed, the cost ratios of industrial controllers and the plant equipment they control can well be in excess of one thousand. This requirement is coupled with the fact that such controllers are often destined to work in harsher industrial environments. Thus, reliability requirement of industrial control computers are very high. In fact, this is one of the main reasons that decide the high price of such systems compared to commercial computers of comparable or even higher data processing capability.

\[ \triangle \]

### 2.2 Real-Time Systems

A control system behaves correctly if:
• It’s logically correct.
• Terminates its execution respecting timing constraints assigned.

Real-Time Systems (RTS) fail if timing constraints are not met. From a technical point of view the time interval in which the Real Time application is not so critical. As matter of fact a Real Time system must not necessarily be fast, but it must respond from external stimuli within a maximum time which is pre-defined.

For example, a control system for an electrical motor requires to acquire the actual position of each axis controlled 0.5 millisecond. If the real time application fails to acquire the position within the deadline of 0.5 millisecond, the control crashes as well. In this case a single failure of the cyclic application is critical for the system integrity.

We consider a second another example, in which a data–logging system acquires the circuit current of a brushless motor for diagnostic purposes. Such acquisition must take place with the period of 0.005 milliseconds.

However, in this second case, if some data are lost because the acquisition system fails to meet its deadline, the diagnostic system still survive, even with a performance decrease.

In this second case, the deadline is shorter than the previous one, but the criticality of the deadline is less stringent for the survival of the system.

Therefore, Real-time systems can be categorized as **Hard** or **Soft**. For a Hard Real Time, the system is taken to have failed if a computing deadline is not met. In a Soft Real Time, a limited extent of failure in meeting deadlines results in degraded performance of the system, but not in a catastrophic failure. The correctness and performance of an RTS is therefore not measured in terms of parameters such as, average number of transactions per second as in transactional systems such as databases.

In summary:

• **Hard Real Time**: Missing a deadline is a total system failure.

• **Soft Real Time**: The usefulness of a result degrades after its deadline, thereby degrading the system’s quality of service.

Thus, the goal of a hard real-time system is to ensure that all deadlines are met, but for soft real-time systems the goal becomes meeting a certain subset of deadlines in order to optimize some application specific criteria. The particular criteria optimized depends on the application, but some typical examples include maximizing the number of deadlines met, minimizing the lateness of tasks and maximizing the number of high priority tasks meeting their deadlines.

Hard real-time systems are used when it is imperative that an event be reacted to within a strict deadline. Such strong guarantees are required of systems for which not reacting in a certain interval of time would cause great loss in some manner, especially damaging the surroundings physically or threatening human lives (although the strict definition is simply that missing the deadline constitutes failure of the system). For example, a car engine control system is a hard real-time system because a delayed signal may cause engine failure or damage. Other examples of hard real-time embedded systems include medical systems such as heart pacemakers and industrial process controllers. Hard real-time systems are typically found interacting at a low level with physical hardware, in embedded systems. Early video game systems such as the Atari 2600 and Cinematronics vector graphics had hard real-time requirements because of the nature of the graphics and timing hardware.

Soft real-time systems are typically used where there is some issue of concurrent access and the need to keep a number of connected systems up to date with changing situations; for example software that maintains and updates the flight plans for commercial airliners. The flight plans must be kept reasonably current but can operate to a latency of seconds. Live audio-video systems are also usually soft real-time; violation of constraints results in degraded quality, but the system can continue to operate.

### 2.2.1 Real Time Operating System

A Real-Time Operating System (RTOS) is an Operating System with special features that make it suitable for building Real-Time computing applications, such as the control of automatic machines. In a RTOS, correctness of computing depends not only on the correctness of the logical result of the computation, but also on the result delivery time.
An RTOS is expected to respond in a timely, predictable way to unpredictable external stimuli.

A Good RTOS is one that enables bounded (predictable) behavior under all system load scenarios. Note however, that the RTOS, by itself cannot guarantee system correctness, but only is an enabling technology. That is, it provides the application programmer with facilities using which a correct application can be built. Speed, although important for meeting the overall requirements, does not by itself meet the requirements for an RTOS.

2.2.1.1 Programs, Processes, Tasks and Threads. Programs, Processes, Tasks and Threads are often found in literature on OS in similar contexts. All of them refer to a unit of computation. A program is a general term for a unit of computation and is typically used in the context of programming. A process refers to a program in execution. A process is an independently executable unit handled by an operating system. Sometimes, to ensure better utilisation of computational resources, a process is further broken up into threads. Threads are sometimes referred to as lightweight processes because many threads can be run in parallel, that is, one at a time, for each process, without incurring significant additional overheads. A task is a generic term, which, refers to an independently schedulable unit of computation, and is used typically in the context of scheduling of computation on the processor. It may refer either to a process or a thread.

2.2.1.2 Multitasking Very often, the operation of a mechatronic system has both hard and soft real time requirements:

- Implementation of the control system (Hard RT)
- Supervision and data logging (Soft RT)
- Communication with external devices (Soft or Hard RT)

In addition, the specifications on the deadline for the above processes may be different:

- Acquisition of the position of an electrical motor is within a time range of milliseconds.
- Acquisition of the current for the motor control is within a time range of microseconds.
- Acquisition of the temperature of a furnace is within a time range of seconds.

Because the deadlines and importance of several physical processes in an automatic machine, the control system must be structured in tasks that are executed in parallel which have specific different deadlines.

The Hardware and the software a control system must support the execution of parallel processes (multitasks system), which must be carried out logically in parallel. The parallelism is actually the simulated type, and the calculation time of the CPU is divided between the active processes (tasks).

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications.

Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilisation of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints.

In a multi–tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system. Cooperation will require that the tasks communicate with each other and share common data in an orderly an disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed.
2.2.2 Task Preemption

In computing, **preemption** is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such a change is known as a context switch. It is normally carried out by a privileged task or part of the system known as a **preemptive scheduler**, which has the power to preempt, or interrupt, and later resume, other tasks in the system.

In multitasking OS, there are 4 common states of tasks. The **Ready** means the task is running, the **Delayed** mean task waiting, for e.g., waiting for resource or waiting for higher priority task finished then the lower priority task resume its works. The **Suspended** for e.g., the task was created and it enter the suspended state before come to ready state. Finally, the **Pended** (blocked) mean the task was halted or stopped. Each task can move from one state to another state that’s depend on the Scheduler of CPU.

![Task state diagram](image)

**Figure 2.2:** The Task state diagram.

The term **preemptive multitasking** is used to distinguish a multitasking operating system, which permits preemption of tasks, from a **non–preemptive multitasking** system wherein processes or tasks must be explicitly programmed to yield when they do not need system resources.

In simple terms: Preemptive multitasking involves the use of an interrupt mechanism which suspends the currently executing process and invokes a scheduler to determine which process should execute next (see figure 2.3 for an example of preemptive multitasking). Therefore all processes will get some amount of CPU time at any given time.

![Preemptive task mechanism](image)

**Figure 2.3:** A preemptive task mechanism.
In preemptive multitasking, the operating system kernel can also initiate a context switch to satisfy the scheduling policy’s priority constraint, thus preempting the active task. In general, preemption means “prior seizure of”. When the high priority task at that instance seizes the currently running task, it is known as preemptive scheduling.

Nonpreemptive multitasking is a style of computer multitasking in which the operating system never initiates a context switch from a running process to another process (see figure 2.3 for an example of nonpreemptive multitasking). Such systems are either statically scheduled, most often periodic systems, or exhibit some form of cooperative multitasking, in which case the computational tasks can self-interrupt and voluntarily give control to other tasks. When non preemptive is used, a process that receives such resources can not be interrupted until it is finished.

Cooperative multitasking is a type of multitasking in which the process currently controlling the CPU must offer control to other processes. It is called “cooperative” because all programs must cooperate for it to work. In contrast, preemptive multitasking interrupts applications and gives control to other processes outside of an application’s control.

![Figure 2.4: A nonpreemptive task mechanism.](image)

### 2.3 Intertask communication

Communication among tasks is an inherent part of control software. Intertask communication is used both to make data generated in one task available to others and to synchronize the activities of different tasks.

A designer may choose to run all tasks in a single process but assign each task to a separate, asynchronously executing thread within that process. Since the tasks within different threads share the process’s address space, normal language syntax such as function calls or external variables can be used for data transfer between tasks. However, if any form of preemptive scheduling is used there is a risk of data corruption.

For example, if a task which is reading some data from a globally accessible variable in memory is preempted by a task which writes new data to that variable before the reading is complete, the data which is read may be corrupted. Methods must be specified to prevent such an occurrence. Different methods must be used depending on whether the tasks are running in different threads within the same process or within different processes.

#### 2.3.1 Data Integrity in Communication Within a Process

Data integrity must be examined at three levels:

- Assuring that individual data values are correct
• Assuring that sets of data values have consistent context
• Assuring that data used within extended calculations are consistent

**Integrity of individual data value**  The first problem, that of the integrity of individual data values, arises in single process systems that use preemptive dispatching.

Assume, for example, an assignment statement, \( x = y \); in a low priority task. Assume also that the variable \( y \) can be set in a task of high priority that is dispatched preemptively. Interrupts preempt computational operations at the machine instruction level.

That is, the interrupt allows the currently executing machine-language instruction to complete before taking effect. But the C language statement \( x = y \); may require several machine language operations to complete. There is a chance that the high priority task will interrupt and change the value of \( y \) at the point where only part of \( y \) has been copied to \( x \).

After the high priority task finishes, the remainder of the copy operation will take place, but now using the new value of \( y \). The result can be an \( x \) value that is corrupted and bears little resemblance to either the new or old values of \( y \). For the sake of illustration, consider \( x \) and \( y \) to be 8-bit signed integers (2’s-complement) in a computer that moves data 4 bits at a time.

The initial value of \( y \) is 1, or in binary, 00000001. Suppose that just after the transfer of data begins, an interrupt occurs and the high priority task changes \( y \) to -1, which in binary is 11111111. The transfer is then completed after the high priority task has finished. There are two possible results, depending on whether the first four or last four bits are transferred first.

If the first four bits (the high-order bits) are transferred first, the sequence of values taken on by \( x \) is:

\[
\text{XXXXXXXX} \rightarrow 0000\text{XXXX} \rightarrow 00001111
\]

Here we use Xs to represent the original bit values of \( x \). If the last four bits (the low-order bits) change first, the sequence is:

\[
\text{XXXXXXXX} \rightarrow \text{XXXX}0000 \rightarrow 11110001
\]

If there were no interrupt, \( y \) would be 1 throughout the transfer, and the value would be correctly transferred from \( y \) to \( x \). If the high-order bits were transferred first, the sequence would be

\[
\text{XXXXXXXX} \rightarrow 0000\text{XXXX} \rightarrow 00000001
\]

and if the low-order bits are transferred first, the sequence would be

\[
\text{XXXXXXXX} \rightarrow \text{XXXX}0001 \rightarrow 00000001
\]

In the preemptive case \( x \) ends up as either 15 (00001111) or -15 (11110001), depending on the order of transfer. In either case, the result is far removed from either the initial value of \( y \) or its changed value. If this error took place when our code was controlling a mechanical system, the result could easily be an actuator value that caused a violent motion of the controlled system, possibly damaging the machine or injuring someone.

**Consistency of data structure**  The second data integrity issue, assuring the consistency of data sets, arises from a similar cause. In this case, data associated with an array, table, or data structure is communicated across tasks.

Again, if an untimely interrupt occurs, some of the data in the set will be associated with an older version and some will be associated with the new version. In this case, although all of the data values may be correct, the set of data may not be.

For example, consider a control system whose gains are stored in an array. A lower priority task computes the gains and stores them in the array, and a higher priority task runs the control loop using these values. It may be the case that the arrays of gains which are created by the lower priority task allow stable control of the system, but the control gains which result from reading the array as it is being changed cause the controller to become unstable, causing damage or injury.
**Consistency of data in program**  
The third data integrity issue arises when a variable is used several times in a sequence of C (or other language) statements. For example, the following short sequence is commonly used to apply a limit to a variable:

```c
x = some_calculation ();
if (x > max) x = max;
```

If this code were in a low priority task and a high priority task used the value of \( x \), an interrupt occurring between these two lines of code would cause the high priority task to use the value of \( x \) computed before the limit was applied. This is an intermediate result; it was never intended that it be made available outside the function in which it resides. The preemption violates that intention and the high priority task gets an incorrect value with possibly serious consequences.

These potential problems will not always occur. It requires that the preemption happen at just the right (or wrong) moment. The interrupt responsible for the preemption, however, is usually asynchronous with respect to the low priority code that is originally executing—in other words, we can neither predict nor control which lines of code are running when that interrupt occurs. The probability for an error to occur might actually be quite small. Thus, it may be unlikely that the error will be caught in normal lab testing, but very likely that it will show up in a production system.

### 2.3.2 Design rules

**Temporarily masking/disabling interrupts**  
In systems programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing, the current thread. The processor responds by suspending its current activities, saving its state, and executing a small program called an interrupt handler (or interrupt service routine, ISR) to deal with the event. This interruption is temporary, and after the interrupt handler finishes, the processor resumes execution of the previous thread. There are two types of interrupts:

A **hardware interrupt** is an electronic alerting signal sent to the processor from an external device, either a part of the computer itself such as a disk controller or an external peripheral. 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. Unlike the software type (below), hardware interrupts are asynchronous and can occur in the middle of instruction execution, requiring additional care in programming. The act of initiating a hardware interrupt is referred to as an interrupt request (IRQ).

A **software interrupt** is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed. The former is often called a trap or exception and is used for errors or events occurring during program execution that are exceptional enough that they cannot be handled within the program itself.

For example, if the processor’s arithmetic logic unit is commanded to divide a number by zero, this impossible demand will cause a divide-by-zero exception, perhaps causing the computer to abandon the calculation or display an error message. Software interrupt instructions function similarly to subroutine calls and are used for a variety of purposes, such as to request services from low level system software such as device drivers. For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to the disk.

Each interrupt has its own **interrupt handler**. The number of hardware interrupts is limited by the number of interrupt request (IRQ) lines to the processor, but there may be hundreds of different software interrupts.

Interrupts are a commonly used technique for computer multitasking, especially in real-time computing, to implements the context switching between two different tasks to fulfill preemption request.

General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wishes. Some modern CPUs don’t allow user mode code to disable interrupts as such control is considered a key operating system resource. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.
On single-processor systems, if the application runs in kernel mode and can mask interrupts, this method is the solution with the lowest overhead to prevent simultaneous access to a shared resource. While interrupts are masked and the current task does not make a blocking OS call, then the current task has exclusive use of the CPU since no other task or interrupt can take control, so the critical section is protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few instructions and contains no loops. This method is ideal for protecting hardware bit-mapped registers when the bits are controlled by different tasks.

**Binary semaphores** In computer science, a semaphore is a variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes to a common resource in a parallel programming or multi-user environment.

A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to safely (i.e., without race conditions) adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available.

To better understand the idea behind the concept of the semaphore, let’s consider a library analogy. Suppose a library has 10 identical study rooms, intended to be used by one student at a time. To prevent disputes, students must request a room from the front counter if they wish to make use of a study room. When a student has finished using a room, the student must return to the counter and indicate that one room has become free. If no rooms are free, students wait at the counter until someone relinquishes a room.

The clerk at the front desk does not keep track of which room is occupied or who is using it, nor does she know if the room is actually being used, only the number of free rooms available, which she only knows correctly if all of the students actually use their room and return them when they’re done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. Once access to a room is granted, the room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario the front desk represents a semaphore, the rooms are the resources, and the students represent processes. The value of the semaphore in this scenario is initially 10. When a student requests a room he or she is granted access and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the resulting value of the semaphore is negative, they are forced to wait. When multiple people are waiting, they will wait in a queue.

Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores.

When the shared resource must be reserved without blocking all other tasks (such as waiting for Flash memory to be written), it is better to use mechanisms also available on general-purpose operating systems, such as semaphores and OS-supervised interprocess messaging. Such mechanisms involve system calls, and usually invoke the OS’s dispatcher code on exit, so they typically take hundreds of CPU instructions to execute, while masking interrupts may take as few as one instruction on some processors.

A binary semaphore is either locked or unlocked. When it is locked, tasks must wait for the semaphore to unlock. Typically a task will set a timeout on its wait for a semaphore. There are several well-known problems with semaphore based designs such as priority inversion and deadlocks.

In priority inversion a high priority task waits because a low priority task has a semaphore, but the lower priority task is not given CPU time to finish its work. A typical solution is to have the task that owns a semaphore run at (inhibit) the priority of the highest waiting task. But this simple approach fails when there are multiple levels of waiting: task A waits for a binary semaphore locked by task B, which waits for a binary semaphore locked by task C. Handling multiple levels of inheritance without introducing instability in cycles is complex and problematic.

In a deadlock, two or more tasks lock semaphores without timeouts and then wait forever for the other task’s semaphore, creating a cyclic dependency. The simplest deadlock scenario occurs when two tasks alternately lock two
semaphores, but in the opposite order. Deadlock is prevented by careful design or by having floored semaphores, which pass control of a semaphore to the higher priority task on defined conditions.

**Message passing** The other approach to resource sharing is for tasks to send messages in an organized message passing scheme. In this paradigm, the resource is managed directly by only one task. When another task wants to interrogate or manipulate the resource, it sends a message to the managing task. Although their real-time behavior is less crisp than semaphore systems, simple message-based systems avoid most protocol deadlock hazards, and are generally better-behaved than semaphore systems. However, problems like those of semaphores are possible. Priority inversion can occur when a task is working on a low-priority message and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its incoming message queue. Protocol deadlocks can occur when two or more tasks wait for each other to send response messages.

▽ Example 2.2: An example of priority inversion

The Mars Pathfinder mission was widely proclaimed as “flawless” in the early days after its July 4th, 1997 landing on the Martian surface. Successes included its unconventional “landing” – bouncing onto the Martian surface surrounded by airbags, deploying the Sojourner rover, and gathering and transmitting voluminous data back to Earth, including the panoramic pictures that were such a hit on the Web.

But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as “software glitches” and “the computer was trying to do too many things at once”.

David Wilner, Chief Technical Officer of Wind River Systems, explained what did happen in a keynote address at IEEE Real-Time Systems Symposium in 1997. Wind River makes VxWorks, the real-time embedded systems kernel that was used in the Mars Pathfinder mission. In his talk, he explained in detail the actual software problems that caused the total system resets of the Pathfinder spacecraft, how they were diagnosed, and how they were solved.

VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Pathfinder contained an “information bus”, which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue. The spacecraft also contained a communications task that ran with medium priority.

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

This scenario is a classic case of priority inversion.

The conflict of sharing resource and preemptive kernel are the reason lead to the priority inversion. There are 2 type of of priority inversion that are **Bounded** and **Unbounded** Inversion. We will examine step by step the process of using share resource then explain the Priority Inversion.
Identify the problem In the following example, there are 2 threads A and B running concurrently. They are trying to modify the sharing resource (global value X) by increase the value of X. Then the question is what is the value of X at the final? and how to determine the value of X at an specific time?

Figure 2.5: Multiple access to a single resource.

Then the solution: at the first time, each thread will try to lock the resource and only one thread success. The owning resource-thread will using resource while the others waiting for resource will be released. When the owning resource-thread release (unlock) one of the others locked the resource and continue running.

Figure 2.6: Access to a shared resource.

Bounded Inversion There is one low priority task (LPT) running and it locked the resource. The high priority task (HPT) appear. The LPT will be preemptive by the HPT, that is the property of preemptive kernel.
But the LPT not release the resource, the HPT require resource for it’s executing. In this situation, the scheduler have to put the HPT in waiting state and LPT in running state until the LPT release (unlock) the resource, then the HPT can continue running.

Unbounded Inversion  The problem becomes worst if during the time that the HPT waiting, LPT is running and LPT locking the resource, that’s appear the medium priority task(MPT).

MPT not request resource, and it’s will preemptive the LPT. The LPT have to waiting until MPT finished, and continues running then release the resource to HPT. But we don’t know exactly when the MPT finished and MPT can run forever. The situation now, the LPT waiting for MPT finished and cannot unlock resource for HPT, the HPT waiting fot resource from LPT, the MPT running unknown.

The Unbound inversion can lead to Chain of Nested Resource Lock and Deadlock. These thing will make the CPU waiting forever and lost data.

Avoid Priority Inversion  Most of RTOS uses a methodology called priority inheritance to solve Inversion Priority Problem, which can be described as follows:

1. When a low-priority task acquires a shared resource, the task continues running at its original priority level.
2. If a high-priority task requests ownership of the shared resource, the low-priority task is hoisted above the requesting task. The low-priority task can then continue executing until it releases the resource.
3. Once the resource is released, the task is dropped back to its original low-priority level, permitting the high-priority task to use the resource it has just acquired.
Figure 2.8: Unbounded priority inversion.

Figure 2.9: Task allocation

Figure 2.10: Deadlock
3.1 Programmable Logic Controller (PLC)

A programmable logic controller (PLC) or programmable controller is a digital computer used for automation of industrial processes, such as control of machinery on factory assembly lines. Unlike general-purpose computers, the PLC is designed for multiple inputs and output arrangements, extended temperature ranges, immunity to electrical noise, and resistance to vibration and impact. Programs to control machine operation are typically stored in battery-backed or non-volatile memory. A PLC is an example of a real time system since output results must be produced in response to input conditions within a bounded time, otherwise unintended operation will result.

Hence, a programmable logic controller is a specialized computer used to control machines and processes. It therefore shares common terms with typical PCs like central processing unit, memory, software and communications. Unlike a personal computer though the PLC is designed to survive in a rugged industrial atmosphere and to be very flexible in how it interfaces with inputs and outputs to the real world.

The components that make a PLC work can be divided into three core areas.

- The power supply and rack
- The central processing unit (CPU)
- The input/output (I/O) section

PLCs come in many shapes and sizes. They can be so small as to fit in your shirt pocket while more involved controls systems require large PLC racks. Smaller PLCs (a.k.a. “bricks”) are typically designed with fixed I/O points. For our consideration, we’ll look at the more modular rack based systems. It’s called “modular” because the rack can accept many different types of I/O modules that simply slide into the rack and plug in.
3.1.1 Rack

The rack is the component that holds everything together. Depending on the needs of the control system it can be ordered in different sizes to hold more modules. Like a human spine the rack has a backplane at the rear which allows the cards to communicate with the CPU. The power supply plugs into the rack as well and supplies a regulated DC power to other modules that plug into the rack. The most popular power supplies work with 120 VAC or 24 VDC sources.

3.1.2 The CPU

The brain of the whole PLC is the CPU module. This module typically lives in the slot beside the power supply. Manufacturers offer different types of CPUs based on the complexity needed for the system.

The CPU consists of a microprocessor, memory chip and other integrated circuits to control logic, monitoring and communications. The CPU has different operating modes. In programming mode it accepts the downloaded logic from a PC. The CPU is then placed in run mode so that it can execute the program and operate the process.

Since a PLC is a dedicated controller it will only process this one program over and over again. One cycle through the program is called a scan time and involves reading the inputs from the other modules, executing the logic based on
these inputs and then updated the outputs accordingly. The scan time happens very quickly (in the range of 1/1000th of a second). The memory in the CPU stores the program while also holding the status of the I/O and providing a means to store values.

Figure 3.3: The CPU of a PLC

3.2 Execution of program in PLC

The PLC processor has four major tasks executed repeatedly in the following order:

1. **Read** the physical inputs.
2. **Execute** the user programs (tasks).
3. **Write** the physical outputs.
4. **Housekeeping** tasks.

The processor repeats these tasks as long as it is running. The housekeeping tasks include communication with external devices and hardware diagnostics.

The time required to complete these four tasks is defined as the scan time and is typically ranges from a few milliseconds up to a few hundred milliseconds, depending on the length of the program. For very large programs, the scan time can be relatively long, causing the PLC program to miss transient events, especially if they are shorter than the scan time. In this situation, the possible solutions are:

1. Break program in program units (tasks, function blocks, or routines) that are executed at variable rate, computed on the time characteristics of the physical process which is controlled (Multitask).
2. Lengthen the time of the transient event so it is at least twice the maximum scan time (External hardware).
3. Partition long calculations (for example, array manipulation) into smaller parts so only a portion of the calculation is solved during a scan time (single task broken in execution “pieces”)
Modern PLC fully support the multitask paradigm, with powerful software primitives for inter-task communication, therefore the first solution is by far the most used to solve timing problems. Only if multitasking is a concern, and then only one task is available in the target control computer, the latter solution is an option.

![Figure 3.4: PLC scan.](image)

Normally, during the program scan, changes in physical inputs cannot be sensed, nor can physical outputs be changed at the output module terminals. However, some PLC processors have a function block that can read the current state of a physical input and another function block that can immediately set the current state of a physical output. However, this might cause inconsistency in the value of the same variable (which has a physical meaning) in the same PLC scan.

REFERENCES


CHAPTER 4

PLC PROGRAMMING

4.1 Principle of programming PLC

The first language which has been applied in PLC programming is the Ladder Logic Diagram language. An argument that aided the initial adoption of ladder logic was that a wide variety of engineers and technicians would be able to understand and use it without much additional training, because of the resemblance to familiar hardware systems. This argument has become less relevant given that most ladder logic programmers have a software background in more conventional programming languages, and in practice implementations of ladder logic have characteristics, such as sequential execution and support for control flow features, that make the analogy to hardware somewhat inaccurate.

Ladder logic is widely used to program PLCs, where sequential control of a process or manufacturing operation is required. Ladder logic is useful for simple but critical control systems or for reworking old hardwired relay circuits. As programmable logic controllers became more sophisticated it has also been used in very complex automation systems. Often the ladder logic program is used in conjunction with an HMI program operating on a computer workstation.

Manufacturers of programmable logic controllers generally also provide associated ladder logic programming systems. Typically the ladder logic languages from two manufacturers will not be completely compatible; ladder logic is better thought of as a set of closely related programming languages rather than one language. (The IEC 61131-3 standard has helped to reduce unnecessary differences, but translating programs between systems still requires significant work.) Even different models of programmable controllers within the same family may have different ladder notation such that programs cannot be seamlessly interchanged between models.

Ladder logic can be thought of as a rule-based language rather than a procedural language. A "rung" in the ladder represents a rule. When implemented with relays and other electromechanical devices, the various rules "execute" simultaneously and immediately. When implemented in a programmable logic controller, the rules are typically executed sequentially by software, in a continuous loop (scan). By executing the loop fast enough, typically many times per second,
the effect of simultaneous and immediate execution is relatively achieved to within the tolerance of the time required to execute every rung in the "loop" (the "scan time"). It is somewhat similar to other rule-based languages, like spreadsheets or SQL. However, proper use of programmable controllers requires understanding the limitations of the execution order of rungs.

### 4.2 Ladder Logic syntax

The language itself can be seen as a set of connections between logical checkers (contacts) and actuators (coils). If a path can be traced between the left side of the rung and the output, through asserted (true or “closed”) contacts, the rung is true and the output coil storage bit is asserted (1) or true.

If no path can be traced, then the output is false (0) and the “coil” by analogy to electro–mechanical relays is considered “de-energized”.

Ladder logic has contacts that make or break circuits to control coils. Each coil or contact corresponds to the status of a single bit in the programmable controller’s memory.

So-called “contacts” may refer to physical (“hard”) inputs to the programmable controller from physical devices such as pushbuttons and limit switches via an integrated or external input module, or may represent the status of internal storage bits which may be generated elsewhere in the program.

Each rung of ladder language typically has one coil at the far right. Some manufacturers may allow more than one output coil on a rung.

1. -( )- A regular coil, energized whenever its rung is closed.
2. -\( \\
\) A “not” coil, energized whenever its rung is open.
3. -[ ]- A regular contact, closed whenever its corresponding coil or an input which controls it is energized.
4. -[\] A “not” contact, closed whenever its corresponding coil or an input which controls it is not energized.

The “coil” (output of a rung) may represent a physical output which operates some device connected to the programmable controller, or may represent an internal storage bit for use elsewhere in the program.

### 4.3 Examples

Some example of boolean equation in Ladder Diagram language:

The most simple instruction is:

\[
\begin{align*}
X & \text{ } S \\
\text{--[ ]--( )}
\end{align*}
\]

which is equivalent to the “C” code:

```c
if ( X ) {
    S=1;
} else {
    S=0;
}
```

It should be noted that the “C” code:
is not equivalent to the above code in Ladder Diagram, since in the latter “C” code, once the variable $X$ become true, then the variable $S$ become true, and still true even if the variable $X$ turns into false at the following PLC scan cycle.

Contacts in parallel implement the OR function (the result is true if either contact are true), while contact in series implement the AND function. For example the code

```
if ( X AND (Y OR Z) ) {
    S=1;
} else {
    S=0;
}
```

Typically, complex ladder logic is “read” left to right and top to bottom. As each of the lines (or rungs) are evaluated the output coil of a rung may feed into the next stage of the ladder as an input. In a complex system there will be many “rungs” on a ladder, which are numbered in order of evaluation.

```
if ( X AND (Y OR Z) ) {
    S=1;
} else {
    S=0;
}
```

that corresponds to:

```
if ( X AND (Y OR Z) ) {
    S=1;
} else {
    S=0;
}
if ( S AND X ) {
    T=1;

if ( S AND X ) {
    T=1;
```
This represents a slightly more complex system for rung 2. After the first line has been evaluated, the output coil (S) is fed into rung 2, which is then evaluated and the output coil T could be fed into an output device (buzzer, light etc.) or into rung 3 on the ladder. (Note that the contact X on the second rung serves no useful purpose, as X is already defined in the “AND” function of S from the 1st rung.)

This system allows very complex logic designs to be broken down and evaluated.

Often we have a green “start” button to turn on a motor, and we want to turn it off with a red “stop” button. The stop button itself is wired as a normally closed switch. This means that when the stop button is in its normal state (not pushed), the PLC input will be true. When the stop button is pushed, the input will go false. This will make the rung false and stop the “run” output. A normally open contact must be used in the ladder diagram, since this input is normally turned on through the normally closed pushbutton contact, and turns off when the button is pressed.

```plaintext
start stop run
--+----[ ]--+----[ ]---( )
|     |
|     run |
+-----[ ]---+

run motor
-------[ ]--------------( )
```

This latch configuration is a common idiom in ladder logic. In ladder logic it is referred to as seal-in logic. The key to understanding the latch is in recognizing that “start” switch is a momentary switch (once the user releases the button, the switch is open again). As soon as the “run” solenoid engages, it closes the “run” switch, which latches the solenoid on. The “start” switch opening up then has no effect.

Ladder notation is best suited to control problems where only binary variables are required and where interlocking and sequencing of binary is the primary control problem. Since execution of rungs is sequential within a program and may be undefined or obscure within a rung, some logic race conditions are possible which may produce unexpected results; complex rungs are best broken into several simpler steps to avoid this problem. Some manufacturers avoid this problem by explicitly and completely defining the execution order of a rung, however programmers may still have problems fully grasping the resulting complex semantics.

Analog quantities and arithmetical operations are clumsy to express in ladder logic and each manufacturer has different ways of extending the notation for these problems. There is usually limited support for arrays and loops, often resulting in duplication of code to express cases which in other languages would call for use of indexed variables.

### 4.4 Programming languages for PLC: International Standard IEC61131-3

Standardising the programming interface harmonises the way engineers design and operate industrial controls. A standard programming interface allows people with different backgrounds and skills to create different elements of a program during different stages of the software lifecycle: specification, design, implementation, testing, installation and maintenance [5].

Yet all pieces adhere to a common structure and work together harmoniously. Decomposition into logical elements, modularisation and modern software techniques result in programs that are structured, have increased re-usability, fewer errors and that are developed more efficiently.
Standardisation facilitates training by allowing users to learn on one model of hardware, but apply common techniques to all compatible hardware. This approach also facilitates maintainability and removes subjectivity from software development making code and is immediately familiar, even if not developed by the reader of that code.

The primary standard available for programmable logic controller (PLC) programming is defined in IEC 61131-3. This standard has been embraced by many other international standards organisations. For example, the American National Electrical Manufacturers Association (NEMA) standard IA 2.3-2005 is based on IEC 61131-3 as is the British Standard BS EN 61131-3.

IEC 61131-3 is the third part of the IEC 61131 family. This consists of:

- IEC 61131-1 General information establishes the definitions and identify the principal characteristics relevant to the selection and application of programmable controllers and their associated peripherals.
- IEC 61131-2 Equipment requirements and tests specifies equipment requirements and related tests for programmable controllers (PLC) and their associated peripherals.
- IEC 61131-3 Programming Languages providing the basis defines, as a minimum set, the basic programming elements, syntactic and semantic rules for the most commonly used programming languages, including graphical languages of Ladder Diagram and Functional Block Diagram, and Textual languages of Instruction List and structured Text; as well as major fields of application, applicable tests and means by which manufacturers may expand or adapt those basic sets to their own programmable controller implementations.
- IEC 61131-4 User Guidelines A technical report providing general overview information and application guidelines of the standard for the end user of programmable controllers.
- IEC 61131-4 User Guidelines. A technical report providing general overview information and application guidelines of the standard for the end user of programmable controllers.
- IEC 61131-5 Messaging service specification defines the data communication between programmable controllers and other electronic systems using the Manufacturing Message Specification (MMS, according to International Standard ISO/IEC 9506.
- IEC 61131-6 is reserved for future use.
- IEC 61131-7 Fuzzy control programming defines basic programming elements for fuzzy logic control as used in programmable controllers.
- IEC 61131-8 Guidelines for the application and implementation of programming languages provides a software developers guide for the programming languages defined in part 3 for the IEC overview and table of content.

REFERENCES

CHAPTER 5

THE DESCRIPTION A REACTIVE BEHAVIOUR OF AN MECHATRONIC DEVICE

5.1 Introduction

The approach to the development of the control system of a mechatronic device can follow two completely different philosophies:

- The input–output approach, in which the set of all the set of the sensor is analyzed in order to describe the control action (i.e. the reactive behavior of the control system).
- The input–state–output, in which the operational behavior of the machine is described by a set of operation state. In each state the control behaviour analyzes only a subset of sensor to describe the control action of the system.

To better understand the difference between the two approaches, we will describe a didactical example in which the control behavior is described using both the input–output approach and the input–state–output approach.

In the example, a stamping machine (see figure 5.1) to repeatedly go up and down to make stamps on envelopes that are placed on the lower support from another machine. When the control system of the stamping machine receives the signal envelope “present”, starts the motor in the direction of descent until the presence sensor indicates that the lower piston stamping has arrived at the end-stroke. At this point the motor has to reverse its stroke and traced back the piston until it is triggered the sensor upper stroke end. At this point the machine stamping machine indicates that the envelope can be removed from the place of processing.

At first stage, we describe the “input–output” approach to software development. To this end, they should be listed all the possible events diagnosable by the sensors (input signals):

- PRES.ENVELOPE
- PE.DOWN
- PE.UP

then it should be listed all the possible signal for the actuator:
finally, the activation logic is defined using a programming language, for example a latter diagram set of rungs:

```
| |
| PE_UP     M_STAMP_DOWN M_BELT
+--------| |--------------|/|----------( )
| |
| M_BELT    |
+--------| |--------|
| |
| PRES_ENVELOPE PE_UP PE_DOWN M_STAMP_DOWN
+--------|P|--------| |--------------|/|----------( )
| |
| M_STAMP_DOWN    |
+--------| |--------|
| |
| PE_DOWN     PE_UP M_STAMP_UP
+--------| |--------------|/|----------( )
| |
| M_STAMP_UP    |
+--------| |--------|
```

The example written in Ladder Diagram language assumes the use of the instruction `P` for the detection of rising edges of the corresponding signal. In addition, in order to keep the example very simple, this code fragment does not handle any exception or time-out control on the stroke of piston.

From this simple example in which we have three input signals, it’s easy to understand that the complexity of the control design will grow rapidly as well as the number of input sensors increases.

In fact, even if the various parts of the machine taken apart are easy to control with few lines of Ladder Diagram, it can be complex to combine together into a single program all “rung” relative to the various components.
These complications are mainly due to:

- It’s often necessary to insert rungs with contact (i.e., memory tag) to test the general conditions that ensure the operation of the machine (e.g., block of the product at the machine entrance, etc.).

- The separate assessment of the conditions for the activation of outputs (control of actuators) in the two operation modes “automatic” and “manual”, that are usually treated in a different way.

- The necessity of synchronize the components that interact in a direct way. In such a case, it’s necessary to insert rung contacts for the cross verification of the conditions (“interlocks” and “consensus”) to enable the command activation.

- It’s necessary to insert specific lines of code for the supervision of fault conditions, at any point of the machine may occur.

Now we describe the “input–state–output” methodology for the coding of the reactive control software. The project is set by analyzing the operational phases or steps of the machine, in the specific case:

The design according to the model based on the state passes through the analysis of the operating states of the machine, which we isolate according to the description of the operation of the machine.

In order to define an operational state we ask ourselves:

What are the actions that the machine performs at a given time? Are these actions different from those that run at other operational step in the production cycle of the machine?

In the specific case:

1. The stamping is awaiting the arrival of the envelope. The control don’t perform any action.
2. The stamp comes down to stamp the envelope. The current action consists in activating the motor in descendant direction.
3. The stamp is driven upward to reach the rest position (after having stamped)

At this stage we do not specify the implementation of the operations that the process must run in each phase, but only the logical sequence of operations. In other words, we specify “what is doing the machine” and “not how it’s doing”.

Then, we evaluate which are the conditions to be set corresponding to each state:

Subsequently it is evaluated which must be conditions which are to transit the machine from one operating state to the next one:
From the above didactical example it’s possible to outline which are the benefits of the approach to program development based on state. In particular a program which is based on the concept of state based on the rule allows you to organize a control sequence, isolating logic states of operation of the machine in which only a subset of sensors and actuators are of interest for the purposes of manufacturing execution. This allows you to have:

- **Decomposition of complexity ‘a machine.** In each logical state of operation only a subset of sensors and actuators are of interest, for which we can greatly simplify the control logic linking the was being processed at the moment.

- **Greatly simplify the functional analysis of the process.** The decomposition in states of operation facilitates the operation of analyze the operational phases of the process and thus facilitates the drafting of specific software.

- **Facilitates the process of verification of software.** It it easier to run the software verification, as it is sufficient to follow the “steps” of the program and the parallel states of operation of the machine.
• **intrinsic analysis of possible scenarios of operation special conditions.** Focus on an operating condition the machine also allows you to identify the possible behaviors abnormal automatism, in the event of failure or adverse events.

The approach *logic-based* can be more appropriate when:

• **Best management systems event operation non-sequential.** In the case of non-sequential processes, the approach that the direct management of events is simpler. The most important example of this concept is in the management of alarms on the process that feature an combinatorial than sequential behavior.

• **It’s easier to monitor and detect “asynchronous” alarms**, that can occur at any condition operation of the process. For example, the drive of an electrical motor continuously informs the logic controller of its own fault situation. This condition can occur regardless of what the operational state of the machine. In a state-based approach it is necessary to specify the monitoring actions of the fault in all states, with a significant increase in complexity of the control software.

• **The manual handling of the system is easier.** In the case in which the operator can be called to perform manual operations (eg. to solve the jam of material in the process, etc..), It may happen that the state of the machine is not congruent with that of the controller, because of its manipulations of the process took place both through the control system (buttons direct activation of the actuators) that through direct manipulation of the physical parts of the machine. In this case, the logic-based approach is more robust approach to states.

Therefore we can define a formal definition of operational state for an automatic machine:

<table>
<thead>
<tr>
<th>A state is a fundamental condition in the operation of a manufacturing plant that persists for a significant period of time and is distinguishable from any other operating condition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>An operating condition is said distinguishable from each other if it differs:</td>
</tr>
<tr>
<td>• In the event that the production process can generate in that state</td>
</tr>
<tr>
<td>• In the transitions that can change this operating condition.</td>
</tr>
<tr>
<td>• The actions to be performed on the machine.</td>
</tr>
<tr>
<td>A transition is a response to an event (of interest in a given state) that causes a change in the status of the plant itself.</td>
</tr>
</tbody>
</table>

Therefore, since the automatic machine can be described by sequences of production steps, which can be modeled with State and Transitions, the control systems must follow these properties, so that:

| The state memorized in the control system \(\iff\) corresponds to the state of the machine |

or, in other words, the state of the control system must always be congruent with that of the controlled process.

### 5.2 State Transition Logic

State transition logic is formally defined within *finite automata theory*. The state transition logic concept has been further specialized to mechatronic system control through the specification of a functional structure for each state. This structure specifies the organization of code at the state level so that it corresponds closely with the needs of control systems.

The use of transition logic has also been based on the very successful applications of PLCs. These devices, in their simplest form, implement Boolean logic equations, which are scanned continuously. The programming is done using ladder logic, a form of writing Boolean equations that mimics a relay implementation of logic. In basing real-time software design on transition logic, each state takes on the role of a PLC, greatly extending the scope of problems that can be tackled with the PLC paradigm.
5.2.1 States and Transitions

State specifies the particular aspect of its activity that a task is engaged in at any moment. It is the aspect of the design formalism that expresses duration. States are strictly sequential; each task is in one state at a time. Typical activities associated with states are:

- **Moving**: A cutting tool moving to position to start a cut, a carriage bringing a part into place, a vehicle moving at a constant velocity.
- **Waiting**: For a switch closure, for a process variable to cross a threshold, for an operator action, for a specified time.
- **Processing**: Thermal or chemical processes, material coating in webs.
- **Computing**: Where to go, how long to wait, results of a complex calculation.
- **Measuring**: The size of a part, the location of a registration mark, object features from vision input, the distance to an item.

Each state must be associated with a well-defined activity. When that activity ends, a transition to a new activity takes place. There can be any number of transitions to or from a state. Each transition is associated with a specific condition. For example, the condition for leaving a moving state could be that the end of the motion was reached, that a measurement indicated that further motion was not necessary, or that an exception condition such as a stall or excessively large motion error occurred.

5.2.2 Transition Logic Diagrams

State transition logic can be represented in diagrammatic form. Conventionally, states have been shown with boxes or circles, and transitions with lines connecting one state to another. Each line ends with an arrow that shows the direction of the transition. Each transition is labelled with the boolean conditions that specify the trigger event for that transition, as shown in figure 5.2.

This diagram shows a fragment of the transition logic for a task that controls the movement of a materials handling vehicle. The vehicle moves from one position to another, picking up parts in one position and dropping them off at another. The states are shown with rectangles; a description of the state is given inside each rectangle. The transitions are shown with arrows and the transition conditions are shown near the lines denoting transitions. The first “move-to” state shows a typical normal transition as well as an error transition, in this case based on a time-out condition.

Although these diagrams are not essential in using transition logic, they are an excellent visualization tool. If a task is compact enough to fit a logic diagram on a single page, the graphical description makes its function much easier to grasp.

The state representation of a task does not have to be graphical. Complicated tasks can have diagrams that are difficult to read and it can be difficult to automate the process of producing the diagrams from a computer.

Tabular representations can give the same information and, in some cases, are easier to deal with than diagrams. A general form for representing a state follows:

```
State Name [descriptive comment]
- transition target #1; reason for transition [comment]
- transition target #2; ...
```

For the transition logic fragment shown in figure 5.2, the tabular form could be shown as:

```
Move to Loading Position [move the cart]
- Wait to be Loaded; if at loading position
```
Figure 5.2: State transition diagram

- Error state; if time-out
- Wait to be Loaded [hold cart in position until loaded]
- Move cart; if load in place

The tabular format is equivalent to the diagram and more compact.

5.2.3 Example: Pulse-Width Modulation (PWM)

PWM is widely used as an actuation function where there is a need for a digital output to the actuating device, but continuous variation in the actuator output is desired. The need for a digital output usually arises due to concerns about cost and efficiency: Digital, or switching, amplifiers are both less expensive and more efficient than their analog, or linear counterparts. When the actuator plus target system constitute a low-pass filter, PWM can perform the function of a digital-to-analog converter by exploiting the temporal properties of the low-pass filtering. The PWM signal is usually a rectangular wave of fixed frequency with variable duty cycle (i.e., ratio of on-time to cycle time). The logic diagram in figure 5.3 shows a task to implement PWM and a typical PWM signal is shown in figure 5.4.

The PWM task has four states and will produce an effective low frequency PWM from software. The maximum frequency depends on the means of measuring time that is utilized and the timing latencies encountered in running the task. It would be suitable for actuation of a heater or perhaps a large motor, but could be too slow for a smaller sized motor whose mechanical time constant was too fast to allow effective filtering of the PWM signal.

The two main tasks,

PWM_ON

and

PWM_OFF

, turn the output on (or off) on entry and then just wait for the end of the specified time interval.

COMPUTE_TIMES
is active once per cycle to find the appropriate on and off times in the event that the duty cycle has been changed. The transition conditions take account of the two special cases—duty cycles of 0 (never on) and 1 (always on) in addition to the usual on–off switching operation.

This example shows the use of transition logic for a task that is quite low level computationally. Unlike a conventional implementation of such a task, the details are readily apparent from the transition logic and reference to the code (not shown here) need only be made to confirm that it is an accurate implementation of the transition logic.
5.2.4 Transition Logic for the Process Control Example

The diagram of the process system of the previous chapter is repeated in figure 5.5. The control objective is to maintain the mixture, temperature, and level in each of the tanks.

![Prototype process system](image)

**Figure 5.5: Prototype process system**

Of the tasks defined for this system, the level task is probably the most complex. Its job is to figure out which tank needs attention at any time, ask for carriage motion, and initiate the filling action. The transition logic for this task is shown in figure 5.6.

![Transition logic for the level task](image)

**Figure 5.6: Transition logic for the level task**

This task functions largely in a supervisory role. It examines the current status of each of the tanks, and then sends out commands to provide the service it deems necessary. Other tasks are responsible for actually carrying out those commands. This state diagram is typical of the level of detail that is used. The Monitor state is identified as determining which tank needs service. How this is determined is not specified in the state diagram. That is a well-defined function,
and should be specified in the auxiliary documentation accompanying the state diagram (or tabular state representation). When code for this function is written, there will be a very clear path indicating when and how the code is used, and an explanation of what the code should be doing. Other states are simpler and probably do not need much if any auxiliary documentation.

5.2.5 Nonblocking State Code

A major feature of PLCs contributing to their success as industrial control components has been that the logic function is continually scanned. PLCs are programmed in a form of logic notation called ladder diagrams. In programming ladder logic, the programmer does not deal with program flow control, as must be done when using conventional programming languages. As long as the ladder is active, it is scanned repeatedly, so the user only has to be concerned with the fundamental performance issue of whether the scan rate is adequate for the particular control application.

Transition logic design is based on this same scanning principle for execution of state-related code. In order to achieve greater functional flexibility than is possible with ladder diagrams, however, standard sequential languages are used for coding. To implement a scanning structure with algorithmic languages requires the restriction that only non-blocking code can be used. Non-blocking code is a section of program that has predictable execution time.

The execution time for blocking code cannot be predicted. This definition does not say anything about how long that execution time is. For now, it is enough that it be predictable. How long the execution time is for any specific piece of code is a performance issue which will be dealt with later.

Examples of blocking code in the C language include, for example, the

```c
scanf()
```

function call used to get keyboard input from the user. This function only returns when the requested input values have been typed; if the user goes out for coffee, the function simply waits. Likewise, the commonly used construction to wait for an external event such as a switch closure,

```c
while (inbit (bitnum) == 0);
```

is also blocking. If the event never happens, the while loop remains hung.

This restriction to nonblocking code does not cause any loss of generality at all. Quite the contrary, the transition logic structure is capable of encoding any kind of desired waiting situations shown in the examples. By encoding the wait at the transition logic level rather than at the code level, system operations are documented in a medium that an engineer or technician involved in the project can understand without having to understand the intricacies of the program.

5.2.6 State-Related Code

The transition logic metaphor encourages the use of modular software by associating most of the user-written code with states. Although this level of organization is adequate for simple projects, an additional level of organization is necessary in most cases. To this end, an additional formal structure of functions is established for the state-related code. Two goals of modular code writing are thus fulfilled:

- Sections of code are directly connected to identifiable mechanical system operations.
- Individual functions are kept short and easily understood.

These functions are defined around the operational assumption of a scanned environment—the code associated with a state is scanned, that is, executed on a repeated basis, as long as that state is the current state for a task. Code associated with noncurrent (inactive) states is not executed at all.

For each state, the following functions are defined:
- **Entry**: Executed once on entry to the state.
- **Action**: Executed on every scan of the state.
- **Transition test**: Executed after the action function on every scan of the state.

In some cases there is code that is unique to a specific transition. This is subtly different from code that is associated with entry to a state, because the entry to the state could be via one of several different transitions. That code can be placed in the transition test function and only executed when a transition is activated, or it can be in a separate exit function that is associated with a specific transition.

This structure enforces programming discipline down to the lowest programming level. All of these functions must be nonblocking, so test functions, for example, never wait for transition conditions. They make the test, then program execution continues.

Relating code to design-level documentation is also enforced with this structure. Transition logic documentation for each task identifies states in terms of what the mechanical system is doing. Not only is the code relevant to that state immediately identified, the code is further broken into its constituent parts.

### 5.2.7 State Scanning: The Execution Cycle

The state scan is shown in figure 5.7. In addition to establishing the execution order for the state-related functions, it also provides the basis for parallel operation of tasks.

Each pass through the cycle executes one scan for one task. If this is the first time the scan has been executed for the current state, the entry function is executed. The action function is always executed. Then the transition test function is executed. If it determines that a transition should be taken, the associated exit function or transition–specific code is executed and a new state is established for the next scan.

Behind the execution details, there must be a database of task information. Each task must have data specifying its structural information—that is, all of the states and transitions, task parameters such as priority, sample time, and so on, and transient information such as present state and status.

### 5.2.8 Task Concurrency: Universal Real-Time Solution

Tasks, as noted earlier, must operate concurrently. This structure provides for parallel operation of tasks even in the absence of any specific multitasking operating system or dispatcher. Because all of the state functions are nonblocking, the scan cycle itself is also. It can therefore be used to scan each active task in succession. After finishing with all of the tasks, the first task is scanned again. This guarantees fully parallel operation of all tasks.

This method of dispatching, called cooperative multitasking, will be an adequate solution if the total scan time for all tasks is short enough to meet the system timing constraints. If cooperative multitasking is not adequate, a faster computer must be used, or other dispatching solutions must be found.

The methodology discussed thus far therefore presents a universal real-time solution. It is capable of solving all real-time problems, without any special real–time constructs or operating systems, if a fast enough computer is available. All of the usual real-time facilities such as semaphores, task synchronization, and event-based scheduling can be implemented using the formalism of transition logic, with all code nonblocking.

These various facilities of real-time operating systems serve two main functions: making sure that high priority activities are serviced before low priority activities, and providing protections for problems that arise because of the asynchronous nature of real-time control software. A common problem in control system software is that while there might be enough programming power overall, high priority activities do not get enough attention. This is an obvious potential problem with the cooperative multitasking described earlier because all tasks are treated equally. One solution

---

1 The term “scheduling” is often used synonymously with dispatching; in this material, “scheduling” will usually be used for a planning activity concerned with meeting execution time targets, while “dispatching” will be used for the part of the program that actively directs task execution.
to this dilemma is to use a faster computer. However, if the use of a fast enough computer is not practical, use of preemptive dispatching based on interrupts can be implemented. Interrupts do not make the computer any faster; in fact, interrupts have overhead that actually reduces total computing capability. What interrupts do accomplish, though, is to allow higher priority activities to preempt lower priority activities so as to avoid the use of a faster (and more expensive) computer when the real problem is honoring priority rather than total computing capability. To do this, the transition logic paradigm must be extended to include the designation of task types and library code must be included to allow integration of transition logic based code with a variety of real time and/or multiprocessor environments.

5.3 An example: the “Gantry Crane”

A gantry crane is a type of crane which lift objects by a hoist which is fitted in a hoist trolley and can move horizontally on a rail or pair of rails fitted under a beam. An overhead travelling crane, also known as an overhead crane or as a suspended crane, has the ends of the supporting beam resting on wheels running on rails at high level, usually on the parallel side walls of a factory or similar large industrial building, so that the whole crane can move the length of the building, while the hoist can be moved to and from across the width of the building. A gantry crane or portal crane has a similar mechanism supported by uprights, usually with wheels at the foot of the uprights allowing the whole crane
to traverse. Some portal cranes may have only a fixed gantry, particularly when they are lifting loads such as railway cargoes that are already easily moved beneath them.

Let’s suppose our target is to develop the control software of a overhead crane such as shown in figure 5.8.

![Figure 5.8: A gantry crane](image)

This system, the gantry crane, is controller to handle goods in the vertical and horizontal directions, for example to move pieces of goods from an incoming conveyor belt to an outcoming conveyor belt avoiding an obstacle placed between the two.

The development of the control software requires the accomplishment of the following steps:

1. The description of the operational working cycle of the system. This description does not enter into the technological details, but it only states the working functionalities that is required to reach the target.

2. The detailed development of the control logics. In this description the details of the control logics are stated, in particular the sensors and actuators signal are described.

3. The code development. In this step the control software is developed form the detailed description. Automatic generation of the code is an option of the modern software tools.

The description of the operational working cycle is depicted in figure 5.9 and it can be detailed as following:

**Pre-requisite:** The gantry crane is used in an automatic system to pick the product which is arriving transported by a conveyor belt and move to another conveyor belt. The product is picked by a gripper and the motion is controlled so that the gripper is initially moved upward, then rightward and finally downward to release the product on the second conveyor belt.

1. The control system keep the automation waiting the arrival of the product transported by a conveyor belt actuated by an electrical motor.
2. When the product is detected, the controller stops the conveyor belt, and then move down the gripper until it reaches the product.

3. The crane keep the product and moves upward and then rightward to reach the position of the second conveyor belt.

4. The the gripper moves down to reach the position in which the gripper opens and the product is softly dropped over the belt.

The control logics, however, doesn’t take into account the initial state of the gripper. In other words, it’s not assured that the initial state of the gripper correspond to the upper position, therefore it’s necessary to treat explicitly the control of the system in order to reach a known initial state. This problem is called the “homing”.

The “homing” operation puts the equipment at a specific starting point for operation. This starting point is called the home position. Typically, the homing operation is started when you reset the equipment for starting the operation (or re-starting).

In our example, assuming that the homing position corresponds to the upper and left position for the gripper, the home is reached by controlling the sequences of the two following steps:

1. move up the gripper.
2. move left the gripper.

This can be done with an explicit “home” procedure (see figure... ) in which it’s added a sequence reserved for reaching the home position.

Figure 5.9: State diagram of the control of the gantry crane.
Figure 5.10: The sequence of homing