CHAPTER 1 INTRODUCTION

LEARNING OUTCOMES

By the end of this chapter, you should be able to:

1. Explain the real-time systems.
2. Discuss objet-oriented methods and deduces the rationale for developing the OCTOPUS method

INTRODUCTION

Software is used to implement more complex behaviors than other technologies, and its development is contrained more b our mental capabilities than physical processes. Development of software for real-time systems make even harder demands because of their puzzling interaction with the physical world, and the timing cinstraints and continuous service required.

This chapter briefly introduces real-rime systems, discusses object-oriented methids and deduces the rationale for developing the OCTOPUS method.

1.1 REAL-TIME SYSTEMS

Embedded real-time systems support varuous aspects of modern life. The increased power of micrprocessors and steadily falling prices have made digital control systems technically attractive and highly cost-effective. Television sets, cars, instruments and telecommunication equipment are controlled by microprocessors and the related software, which is emebedded into the product itself. The useer does not see or feel this software. The only indication of its existence is the large set of operations provided by the product.

Embeded software has a special characteristics that separates it from other types of software: it is tightly connected to its physical environment. Through sensors and actuators, the control software senses and changes the state of its environment.

Embedded software is normally real-time software, because it has to react to many events wothin the specified time limits.

Real-time software controls the behaviour of a real-time system, which must perform the main part of its operation within given time requirements.Accordingly, real-tiime systems are classified as *hard real-time systems* and *soft-real tme systems*. In hard real-time systems, a late response is treated as an error and may caise loss of life or property. Conversely, a response may also be erroneous if it is returne too early. Hard time constraints arm controllers in industrial robots, or automatics braking systems in cars. In soft real-time systems, a late response is normally acceptable. Typically examples of soft real-time ststems are communications equipment, such as digital telephone exchanges.

Performance issues are critical in all *real-time systems*. In the case of soft real-time systems, performance considerations are related to the capability to adequately handle the external load on the system. In hard real-time systems, performance requirements mean the ability to meet all the specified deadlines.

Typically, a real-time systems has to perform several different tasks. Some of these tasks are prriodic, that is, they are executed at regular time intervals. Other tasks are aperiodic, that is, the need to execute a task may occur at any arbitrary point in time. A real-time system that is able to react to aperiodic requests within given time limits is called reactive. Most embedded real-time systems are reactive; that is, they have to be able to react to new events, even if the system is still processing earlier tasks. Thus, competing requiests are processed concurrently.

Since embedded systems are connected to the surrounding real world, processor overload may occur. In an overload situation, the performance degradation of the system should take place gracefully. During the shortage of resources caused by an

overload situation, some tasks will have to wait for processing. Tasks have been classified into critical, essential and nonessential. Critical tasks have deadlines that must be met. Essential tasks also have deadlines, but failure to meet them will not cause severe problems. Nonessential tasks are allowed to wait without any specified time limit.

1.2 OBJECT-ORIENTED (OO) METHODS

This object-oriented approach to building a system based on the definition of a set of communicating entities called *objects*. The object-oriented technology originates from simulation applications [Dahl `69], where real systems are naturally modelled by the simulation software. For some time, object-oriented technology was seen as an implementation technique, and it was developed alongside programming languages.

OO technology has increasingly gained wider interest and the need to change the focus from implementation issues to software design has become evident. The first step in this direction was taken by the Ada-language community. Booch introduced his method for software design [Booch '87]. On the basis of that work and the ideas behind structured analysis [DeMarco '78], Sally Shlaer and Stephen Mellor introduced their object-oriented analysis (OOA) method in 1985 [Shlaer '85].

The above-mentioned OOA method stimulated a collection of other methods in which the system is based on a conceptual model of the application domain. Such an abstract model clarifies the application by formally organizing and structuring all the relevant nformation. This model consists of entities, attributes, and relationships. Accordingly, it is called an entity-relationships (ER) graph [Chen '76]. These ER graphs are capable of representingstatic relationships. However, they are weak in describing other aspects of the application domain. In order to overcome

this difficulty, the OOA uses state automatons to describe the dynamic behaviour of the system, and data flow diagrams to describe the functionality. All three of the models are similar to structured analysis, although the nodelling proceeds in a reverse order.

Among the many derivatives of OOA, the Object Modelling Techniques (OMT) [Rumbaugh '91] has been very successful. The OMT relaxes the completeness requirements of OOA and covers the whole life cycle of the system developed by giving guidelines for design and implementation.

Behaviou-based methods have been developed as altrenatives to the methods based on conceptual models. The underlying idea behind these methods is simple : communication aspects are analysed first, because in the final stage the system will be composed of a collection of communicating objects. Among the behaviour-centered methods, Class-Responsibility-Colaboration (CRC) cards [Wirfs-Brock '90] is the most well known. Object-Oriented Software Engineering (OOSE) [Jacobson '92] can also be seen as a behaviour-centred method, because it strongly emphasizes use cases.

The Fusion method [Coleman '93] combines the above-mentioned aspects. It emphasizes the role of entity-relationship graphs in the analysis phase, and the behavious-centred view in the design phase.

The OMT and Fusion methods are the basis for the development of OCTOPUS. The object model notation of the OMT enables compact expression of all the necessary details, and the separation of structural, functional and dynamic aspects makes the models easier to build and understand. Basic separation between the analysis phase, concentrating on describing the internal behaviour of the application, is borrowed from the Fusion method.

The good-features of the OMT and Fusion are maintained in OCTOPUS as far as possible, and combines with techniques that are able to cope with the characteristic problems of software development for embedded real-time systems. In particular, the aspects related to reactive behaviour, time domain, and concurrency have been taken into account. OCTOPUS is not the only attempt to introduce object-oriented methods for real-time systems.

CODARTS has its roots in structured analysis and design. It has been complemented by a component called the *domain model* [Gomaa '95] which adds some more object-oriented flavour. The domain model defines several viewpoints for the analysis of the system. In one of these views, the data flow diagram notation is used to develop object communication diagrams. This shows the flow of data and control between nodes that are called *concurrent objects*. Although positioned in the domain model, the object communication diagram makes a design of the system. Consequently, the design phase starts immediately with structuring the system into tasks, for which CODARTS gives good guidelines. The focus on tasks, and separately on information-hiding modules, matches well with the features of the Ada programming language. Thus, an Ada-based architectural design is the natural result.

ROOM is based on a modelling language that allows building formal design models. A ROOM model consists of actors interacting using messages. Interaction takes place through ports. Each port defines a protocol: the input and output messages and their legal ordering. Since each actor can have multiples ports, ports can also be used to reflect different roles of actors. Compostition and layering support the development of more complex systems. ROOM is not a general-purpose method. It is most effective in control-centred application modelled with static structure.

1.3 CONCURRENCY IN A REAL-TIME SYSTEM

Concurrency aspects always occur in embedded real-time systems. The reason is that concurrency is an inherent feature of real-time applications, and it must included in every modelling effort. External events may occur at any point in time, even simultaneously, and they must be queued and handled within preset time limitations.

This can be demonstrated by a simple example, which has enough features to enable the demonstration of the concurrency requirements and also assist in the discussion of different ways to model it. Traditionally. Concurrent programming was developed for two reasons :

1. Concurrency enables an efficient utilization of the underlying hardware.
2. The concurrency approach helps the designer decompose a single program into separate activities, without being concerned with the exact sequencing of these activities.

These reasons are both important in practice. However, they defined in the solution domain and do not reflect the inherent concurrency of the application.

Our view of concurrency is demonstrated by the following example. Let us assume that a system S has to react to two different events called E1 and E2 (*Figure 1.1*).

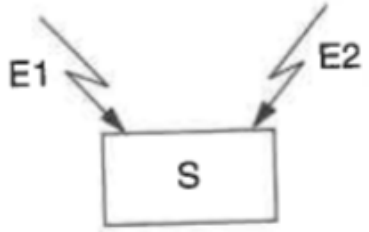

Figure 1.1 : System with two events

To be able to analyse the interleaving of simutaneuous tasks, it is necessary to make assumptions regarding the atomicity and duration of the operations performed by the system. The following assumptions are made :

a- System Sreacts to event E1 by performing a sequenece of three unbreakable operations. T1, T2 and T3, each of them requiring one time unit.

b- System S reacts to event E2 by performing a sequence of two unbreakable operations, T4 and T2, both of them requiring one time unit.

In addition, let us assume that in bith cases the first operation in the sequence is critical, but the succeeding ones are only essential.

- At the occurrence of each event, the system shall start processing the first operation within one time unit, and complete it no later than two time units after the occurrence.
- System S ust complete operation T3 no later than six time units after the occurrence of E1. Operation T2 must be completed no later than four time units after the occurrence of E2.

The sequences of operations are assumed to be independent of each other.

Suppose that the following situation appears:

1. At time 0 event E1 occurs
2. At time 1.5 event E2 occurs
3. At time 3.5 event E1 occurs

Any system that reacts to this situation in the required manner is *concurrent*. It must simultaneously handle more than one sequence of operations. If the system is ideally concurrent, it is able to process any number of independent operation sequences. An ideally concurrent system reacts to the given conditions by performing the specified sequence of operations immediately after the occurrence of each event (*Figure 1.2*).

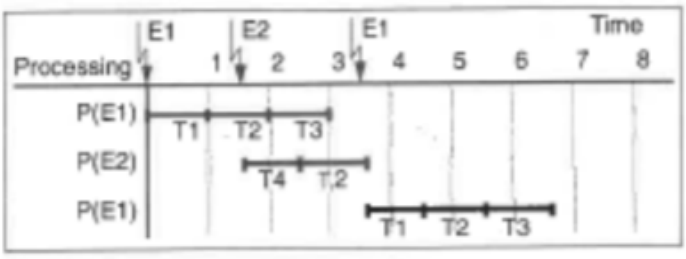

Figure 1.2 : Ideally concurrent system

However, an ideally concurrent system cannot be implemented. Even the approximation of an ideally concurrent system requires vast resources. Embedded systems do not normally have such overwhelming resources due to economic constraints. This, it is meaningful to assume that the systemcontains only one processor, which can oly perform one atomic operation at a time. In such a system, overlapped processing is not possible, and the operations must be performed sequentially. The order of the operations in this sequence totally determines the performance of the system. If the events are handled in a first-in first-out order, the previously given time requirements are not satisfied (*Figure 1.3*).

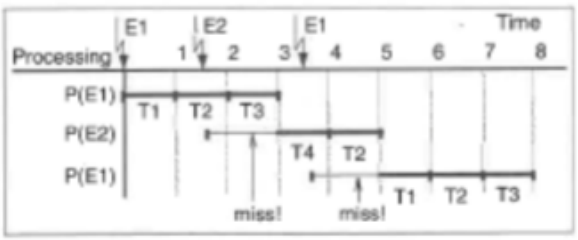

Figure 1.3 : No interleaving of processing

Thus, the system must be able to break the processing of a operation sequence at the completion of any single operation and start processing another sequence of operations, if necessary to meet the time requirements. If this break is made in the manner shown in *Figure 1.4*, the time requirements can be satisfied even with a single processor.

A system is called *quasi-concurrent* if it meets the concurrency requiremenst by postponing the processing of one operation sequence in order to process another sequence. In the example, quasi-concurrency of a single processor sufficiently solves the problem and fulfils the requirements.

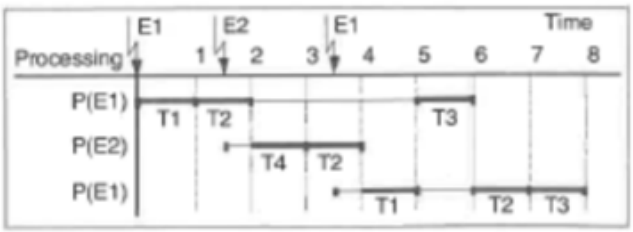

Figure 1.4 : Quasi-concurrent system

1.4 OBJECT-ORIENTED CONCURRENCY MODELS

Software development methods are based on modelling. Throughout the development phases a set of mdels are built, starting from the requirements specifications of the system gradually adding more details until the system is finally ready for use. In a good methods. Each model concentrates on the essential characteristics of the system.

Because concurrency in inherent in all real-time systems, it must be included in the model. The development of an object-oriented methods requires the modelling of objects. The basic difficuly of applying object-oriented methods to the development of real-time systems is how to combine the concepts of concurrency and object.

An object-oriented approach to concurrency modelling is to use either an *explicit concurrency model* or an *implicit concurrency model*. These two models differ in how and when concurrency is taken into account during the software development work.

The *implicit concurrency model* delays the design of concurrency. The object model as such is the basis of computation. As long as the implementation level is not approached, each object is comsidered as an autonomous unit, which performs specific actions concurrently with the actions of other objects. Thus, the system is first analysed and designed as if each object had its own processor providing a thread of execution (see the left side of *Fig. 1.5*). Each external event entering the system is seen as a processing request that broadcasts to some objects, which in turn may request further processing from other objects. Conceptually any number of objects may actually do some processing in response to a single request. The realization of concurreny is addressed later in the design phase and relies on scheduling of the operations of objects.

The *explicit concurreny model* addresses concurrency first and describes concurreny separately from the objects by using the notion of processes as supported by real-time operating systems (see the right side of *Fig.1.5*). This model contains two abstraction levels, objects and processes, addressing first the decomposition of the system into quasi-concurrent processes. Inside each process, object-oriented

technology is applied to replace the conventional functional decomposition and procedural programming techniques. The interaction of objects is seen and implemented as nested function calls. Explicit synchronization mechanisms, such as locks, monitors, and semaphores, have to be added to ensure the integrity of objects where necessary.

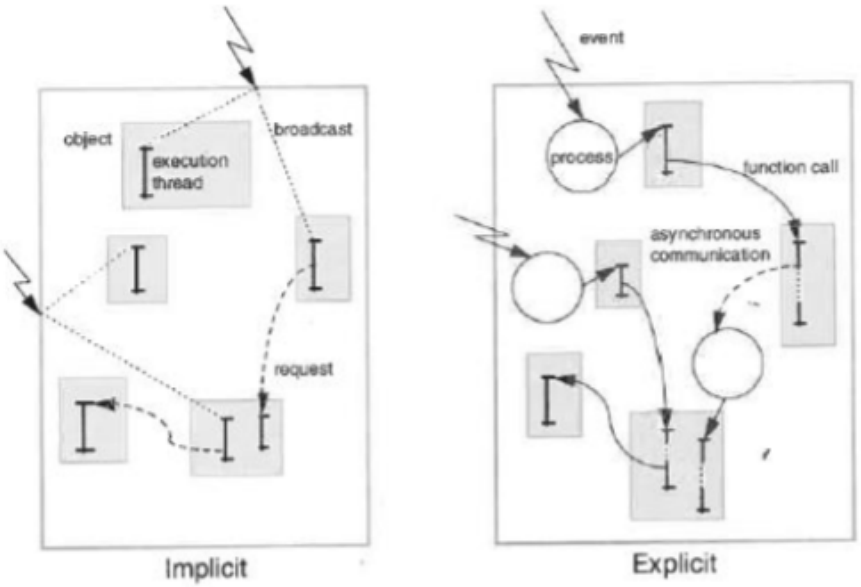

Figure 1.5 – Concurrency models

The *explicit concurrency model* can be applied to the previous example. For that purpose, the system myst be decomposed into a set of processes that are able to model the concurrency as required. Initially, two processes are taken to describe the concurrency.

- Any occurrence of event E1 invokes a new incarnation of a process P1, which processes the operations T1, T2 and T3.
- Any occurrence of event E2 invokes process P2, which processes the operations T4 and T2.

The operating system is assumed to be preemptive, and the processes are assumed to have fixed priorities. Process P2 is set at a higher priority level than P1. In addition, the operations must be made unbreakable by the explicit use of the synchronization means of the operating system. These assumptions are described in *Fig 1.6*.

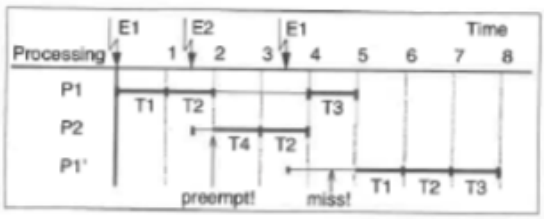

Figure 1.6 – A failed attempt to use process priorities to interleave processing

The first occurrence of E2 is handles correctly. However, when E1 occurs once more, the chosen approach fails, because process P1 cannot pre-empt its earlier occurrence having the same priority. This difficulty can be avoided by separating the first time-critical operation from the subsequent processing of event E1. This is accomplished by dividing P1 into two processes: P11, which processes operation T1, and P12, which processes T2 and T3. Similarly, P2 is divided into P21 for processing T4, and P22 for processing T2. If higher and eaual priorities are assigned to the start-up processes, P11 and P21, the system is able to correctly handle all scenarios of event occurrences (*Fig. 1.7*).

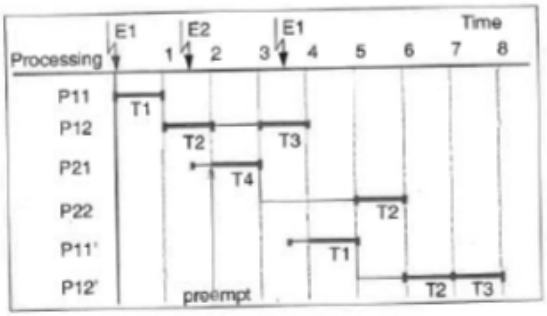

Figure 1.7 – Correct use of process priorities to interleave processing

The above example shows that the use of the explicitly concurrency model implies the division of the system into many processes in order to meet the external requirements. While this division may benefit functional decomposition, it is harmful for object-oriented anaylsis. Since the division is based on the occurrency requirements, not on the concepts of the application domain, object-oriented technology can be applied only to a limited extent.

What would this example look like if an *implicit concurrency model* were constructed? The necessary operations are already known. Each operation can naturally be encapsulated as an operation are already known. Each operation can nautrally be encapsulated as an operation of a single object, resulting in objects O1, O2, O3 and O4 that perform operations T1, T2, T3 and T4 respectively. Furthermore, event E1 requests object O1 to perform operation T1, and O1 in turn requests the execution of T2 from O2, which in turn requests T3 and O2. Similarly, event E2 requests T4 from object O4 which in turn requests T2 from O2. In this way, both events provoke a sequence of object interactions, as shown in *Fig.1*-8.

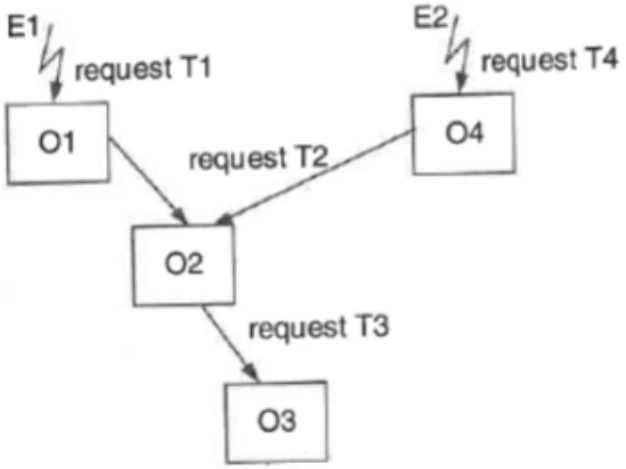

Figure 1.8 - Object interaction sequence

The *implicit concurremcy model* allows the processing prower of each object to be taken for granted. Therefore, concurrent actions between groups of objects do not create any problem. At an early development starge, the processing in the system can be envisioned as in *Fig.1.9*.

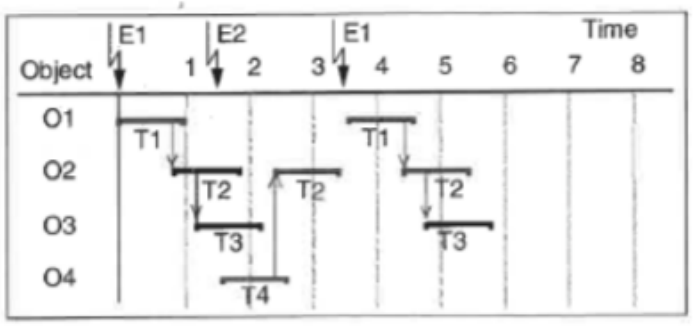

Figure 1.9 – Ideally concurrent object system

Later in the development cycle, one must admit, true parallel processing is not possible with only one processor. To solve this problem, it is necessary to determine how processor time is multiplexed between the objects. The most straightforward solution is to make each operation a unit of processor switching, and to switch the proceesing at the completion of each operation. In this way, the operations automatically become indivisible. If priorities are assigned so that O1 and O4 are on a higher level than O2 and O3, the processing scenario shown in *Fig. 1.10* can be deduced.

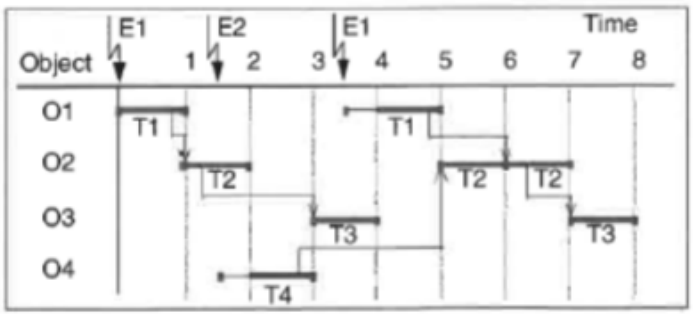

Figure 1.10 – Quasi-concurrent object interaction threads

In this case, only the distinct sequences of object interactions interleave with each other. Thus, these sequences can be regarded as *quasi-concurrent*. The actions of the objects, as such, are mutually exclusive (i.e., no single action can be pre-empted). Nevertheless, the resulting concurrency model meets the previously specified time requirements without requiring any additional synchronization means. It should be noted that the requests between the objects, represented by the thin arrows in *Fig. 1.10*, are asynchronous. Either they have to be implemented using the support functions of the underlying operating system, such as a capability of asynchronous message exchange, or the object themselves are enabled to receive and buffer requests, and to switch the processor between them.

The multiplexing of processor time between several operations is seldom as easy as in the previous example. The complexity of the scheduling depends on the objects that have been defined. For a given application problem, there may be many object designs that yield a correct description of the system. However, only a few of them have suitable processing characteristics.

In conclusion, the main deifference between the implicit and the explicit concurrency models is the timing of combining the objects and processes. The explicit model combines them at early stage. It removes concurrency from the object level by putting processes above objects. This keeps the object domain sequential. The disadvantage is an early need to partition the application into processes. The implicit model postpones the difficulty of implementing quasi-concurreny. This simplifies the analysis phase, but in the design phase, the process structure has to be formulated and the synchronization resolved.

The OCTOPUS method uses the implicit concurrency model in the analysis phase (Chapter 5). In the design phase, the concurrency is gradually made more explicit. This simplifies the analysis and still faciliatates an efficicnet implementation. A stepwise procedure is presented for transforming the implicit concurrency model into an explicit model (Chapter 6).

1.5 LEVELS OF CONCURRENCY

In addition to the general capability of modelling concurrency, object –oriented technology enables the definition of three distinct levels of concurrency :

- Low : *Object-interaction concurrency*
- Medium : *Interobject concurrency*
- High : *Intra-object concurrency*

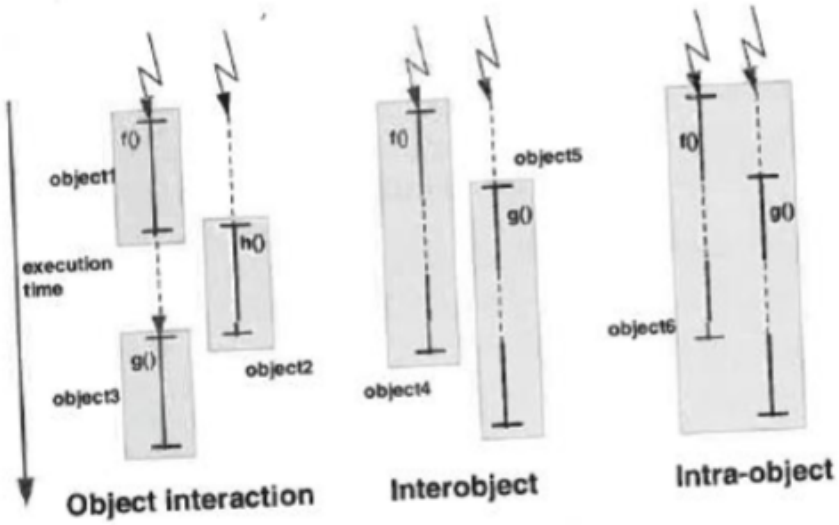

Figure 1.11 – Levels of Conccurrency

1.6 INTRODUCTION TO CASE STUDIES

Throughout this subject, we use examples from two different case studies : the Subscriber Line Test and Cruise Control System.

Both the case studies assume the typical software development situation for embedded system.

1.6.1 SUBSCRIBER LINE TEST

Telecommunication networks evolve continuously. This evolution creates projects where some functionality needs to be added into existing systems. This case study is an example of such a project.

A subscriber line is a pair of wires. In order to function properly, these wires must be without faults. The Subscriber Line Tester (SLT) is able to measure the electrical characteristics of subscriber lines and to determine their condition based on the measured values. Faulty lines can be disconnected and maintenance action called for.

Subscriber lines used to be connected directly to telephone exchanges. In modern networks, many lines are often connected to remote digital transmission systems, which are digitally connected to telephone exchanges. The SLT described in this case study resides in such a remote digital transmission system (DTS) as shown in *Figure 1.12*. It is normally controlled remotely, but it can also be operated locally using a PC. The figure also shows a multiplexer unit (MUX) and channel units (CU).

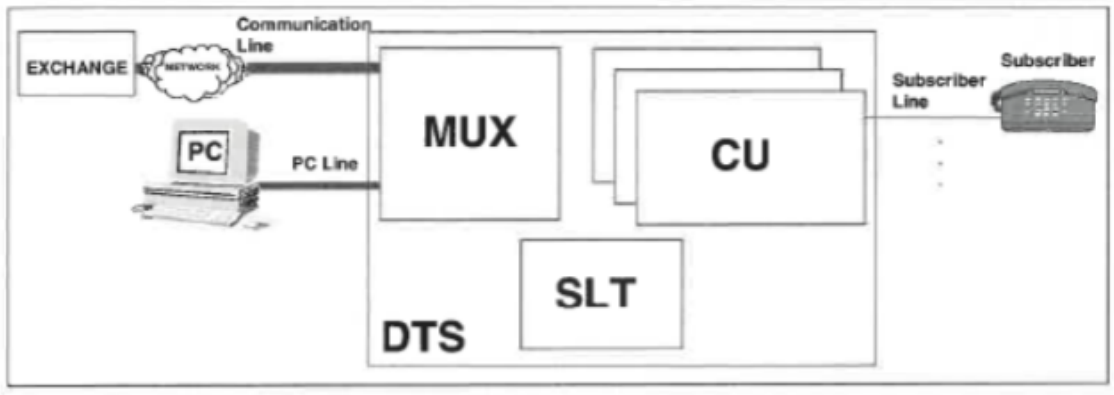

Figure 1.12 : A remote digital transmission system (DTS)

1.6.2 CRUISE CONTROL SYSTEM

A cruise control system maintains a car's speed, even over varying terrain, by controlling the throttle. The driver selects the desired speed or auto-accelerates to it but can regain manual control at any time (Fig. 1.13).

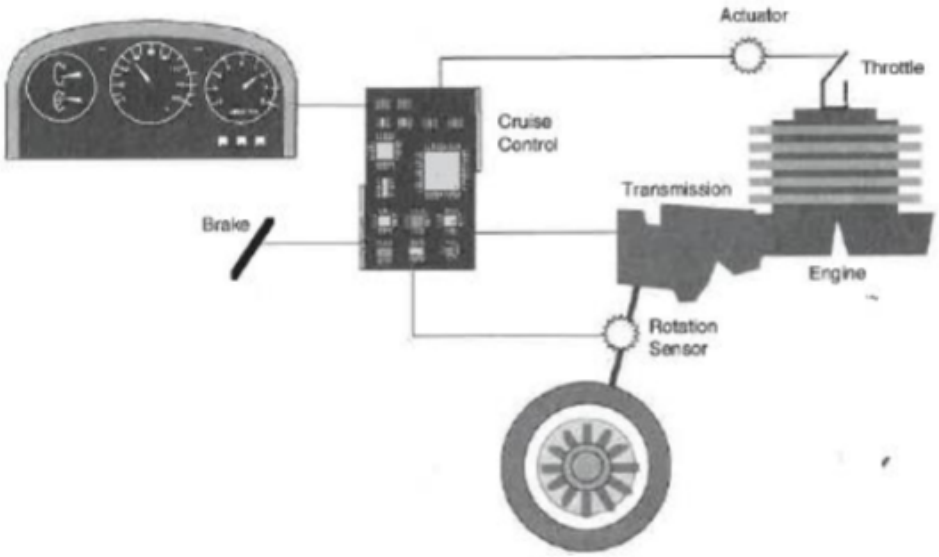

Figure 1.13 : Cruise Control

Cruise control is a standard example of an embedded real-time system that has been used many times in the literature. It is interesting to compare those with the application of OCTOPUS and to look at the differences.

This case study covers all phases and components of the OCTOPUS method and produces a complete software system design, including the complete development process at a level of detail that sometimes may seen overly complete for the given of complexity.