LECTURE NOTES

ON

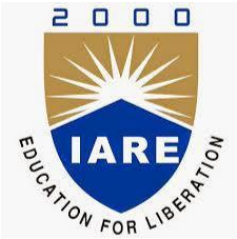
PRINCIPLES OF DISTRIBUTED EMBEDDED SYSTEMS

M.TECH I-semester EMBEDDED SYSTEMS

(AUTONOMOUS-R18)

Dr. S.Vinoth

(Associate Professor)



ELECRTONICS AND COMMUNICATION ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500043

UNIT-I

REAL TIME ENVIRONMENT

Real time environment

A solid understanding of the technical and economic factors that characterize a real-time application helps to interpret the demands that the system designer must cope with. The chapter starts with the definition of a real-time system and with a discussion of its functional and meta-functional requirements. Particular emphasis is placed on the temporal requirements that are derived from the well-understood properties of control applications. The objective of a control algorithm is to drive a process such that a performance criterion is satisfied. Random disturbances occurring in the environment degrade system performance and must be taken into account by the control algorithm. Any additional uncertainty that is introduced into the control loop by the control system itself, e.g., a non-predictable jitter of the control loop, results in a degradation of the quality of control. In the Sects. 1.2 to 1.5 real-time applications are classified from a number of viewpoints. Special emphasis is placed on the fundamental differences between hard and soft real-time systems. Because soft real-time systems do not have severe failure modes, a less rigorous approach to their design is often followed. Sometimes resource-inadequate solutions that will not handle the rarely occurring peak-load scenarios are accepted on economic arguments. In a hard real-time application, such an approach is unacceptable because the safety of a design in all specified situations, even if they occur only very rarely, must be demonstrated vis-a-vis a certification agency. In Sect. 1.6, a brief analysis of the real-time system market is carried out with emphasis on the field of embedded real-time systems. An embedded real-time system is a part of a self-contained product, e.g., a television set or an automobile. Embedded real-time systems, also called cyber-physical (CPS) systems, form the most important market segment for real-time technology and the computer industry in general.

When Is a Computer System Real-Time?

A real-time computer system is a computer system where the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced. By system behavior we mean the sequence of outputs in time of a system. We model the flow of time by a directed time line that extends from the past into the future. A cut of the time line is called an instant. Any ideal occurrence that happens at an instant is called an event. Information that describes an event (see also Sect. 5.2.4 on event observation) is called event information. The present point in time, now, is a very special event that separates the past from the future (the presented model of time is based on Newtonian physics and disregards relativistic effects). An interval on the time line, called a duration, is defined by two events, the start event and the terminating event of the interval. A digital clock partitions

the time line into a sequence of equally spaced durations, called the granules of the clock, which are delimited by special periodic events, the ticks of the clock. A real-time computer system is always part of a larger system – this larger system is called a real-time system or a cyber-physical system. A real-time system changes as a function of physical time, e.g., a chemical reaction continues to change its state even after its controlling computer system has stopped. It is reasonable to decompose a real-time system into a set of self-contained subsystems called clusters.

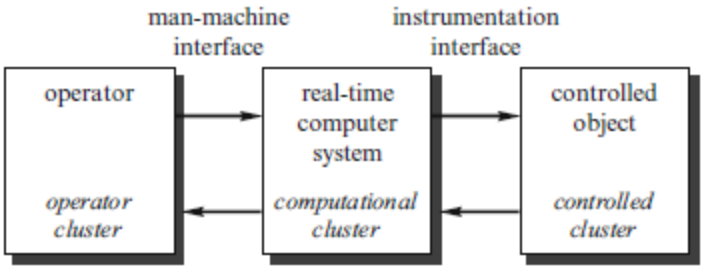


Fig. 1.1 Real-time system

Examples of clusters are (Fig. 1.1): the physical plant or machine that is to be controlled (the controlled cluster), the real-time computer system (the computational cluster;) and, the human operator (the operator cluster). We refer to the controlled cluster and the operator cluster collectively as the environment of the computational cluster (the real-time computer system). If the real-time computer system is distributed (and most of them are), it consists of a set of (computer) nodes interconnected by a real-time communication network. The interface between the human operator and the real-time computer system is called the man–machine interface, and the interface between the controlled object and the real-time computer system is called the instrumentation interface. The man–machine interface consists of input devices (e.g., keyboard) and output devices (e.g., display) that interface to the human operator. The instrumentation interface interface consists of the sensors and actuators that transform the physical signals (e.g., voltages, currents) in the controlled cluster into a digital form and vice versa. A real-time computer system must react to stimuli from its environment (the controlled cluster or the operator cluster) within time intervals dictated by its environment. The instant when a result must be produced is called a deadline. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If severe consequences could result if a firm deadline is missed, the deadline is called hard.

Example: Consider a traffic signal at a road before a railway crossing. If the traffic signal does not change to red before the train arrives, an accident could result.

A real-time computer system that must meet at least one hard deadline is called a hard real-time computer system or a safety-critical real-time computer system. If no hard deadline exists, then the system is called a soft real-time computer system. The design of a hard real-time system is fundamentally different from the design of a soft real-time system. While a hard real-time computer system must sustain a guaranteed temporal behavior under all specified load and fault conditions, it is permissible for a soft real-time computer system to miss a deadline occasionally. The differences between soft and hard real-time systems will be discussed in detail in the following sections. The focus of this book is on the design of hard real-time systems.

Functional Requirements

The functional requirements of real-time systems are concerned with the functions that a real-time computer system must perform. They are grouped into data collection requirements, direct digital control requirements, and man–machine interaction requirements.

Data Collection

A controlled object, e.g., a car or an industrial plant, changes its state as a function of time (whenever we use the word time without a qualifier, we mean physical time as described in Sect. 3.1). If we freeze the time, we can describe the current state of the controlled object by recording the values of its state variables at that moment. Possible state variables of a controlled object car are the position of the car, the speed of the car, the position of switches on the dashboard, and the position of a piston in a cylinder. We are normally not interested in all state variables, but only in the subset of state variables that is significant for our purpose. A significant state variable is called a real-time (RT) entity consists of the sensors and actuators that transform the physical signals.

Every RT entity is in the sphere of control (SOC) of a subsystem, i.e., it belongs to a subsystem that has the authority to change the value of this RT entity. Outside its sphere of control, the value of an RT entity can be observed, but its semantic content cannot be modified. For example, the current position of a piston in a cylinder of the engine is in the sphere of control of the engine. Outside the car engine the current position of the piston can only be observed, but we are not allowed to modify the semantic content of this

observation (the representation of the semantic content can be changed!). The first functional requirement of a real-time computer system is the observation of the RT entities in a controlled cluster and the collection of these observations. An observation of an RT entity is represented by a real-time (RT) image in the computer system. Since the state of a controlled object in the controlled cluster is a function of real time, a given RT image is only temporally accurate for a limited time interval. The length of this time interval depends on the dynamics of the controlled object. If the state of the controlled object changes very quickly, the corresponding RT image has a very short accuracy interval.

Example: Consider the example of Fig. 1.2, where a car enters an intersection controlled by a traffic light. How long is the observation the traffic light is green temporally accurate? If the information the traffic light is green is used outside its accuracy interval, i.e., a car enters the intersection after the traffic light has switched to red, an accident may occur. In this example, an upper bound for the accuracy interval is given by the duration of the yellow phase of the traffic light.

The set of all temporally accurate real-time images of the controlled cluster is called the real-time database. The real-time database must be updated whenever an RT entity changes its value. These updates can be performed periodically, triggered by the progression of the real-time clock by a fixed period (time-triggered (T T) observation), or immediately after a change of state, which constitutes an event, occurs in the RT entity (event-triggered (ET) observation). A more detailed analysis of time-triggered and event-triggered observations will be presented in Chaps. 4 and 5.

Signal Conditioning. A physical sensor, e.g., a thermocouple, produces a raw data element (e.g., a voltage). Often, a sequence of raw data elements is collected and an averaging algorithm is applied to reduce the measurement error. In the next step the raw data must be calibrated and transformed to standard measurement units.

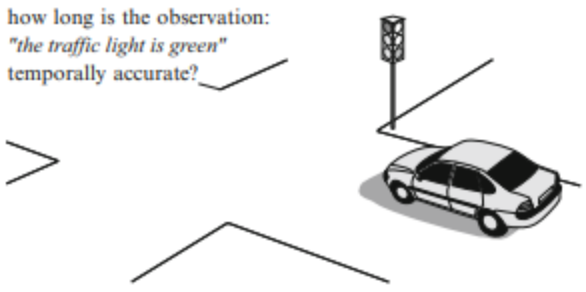


Fig. 1.2 Temporal accuracy of the traffic light information

The term signal conditioning is used to refer to all the processing steps that are necessary to obtain meaningful measured data of an RT entity from the raw sensor data. After signal conditioning, the measured data must be checked for plausibility and related to other measured data to detect a possible fault of the sensor. A data element that is judged to be a correct RT image of the corresponding RT entity is called an agreed data element.

Alarm Monitoring. An important function of a real-time computer system is the continuous monitoring of the RT entities to detect abnormal process behaviors.

Example: The rupture of a pipe, a primary event, in a chemical plant will cause many RT entities (diverse pressures, temperatures, liquid levels) to deviate from their normal operating ranges, and to cross some preset alarm limits, thereby generating a set of correlated alarms, which is called an alarm shower.

The real-time computer system must detect and display these alarms and must assist the operator in identifying a primary event that was the initial cause of these alarms. For this purpose, alarms that are observed must be logged in a special alarm log with the exact instant when the alarm occurred. The exact temporal order of the alarms is helpful in identifying the secondary alarms, i.e., all alarms that can be a causal consequence of the primary event. In complex industrial plants, sophisticated knowledge-based systems are used to assist the operator in the alarm analysis.

Example: In the final report on the August 14, 2003 power blackout in the United States and Canada the following statement: A valuable lesson from the August 14 blackout is the importance of having time-synchronized system data recorders. The Task Force's investigators labored over thousands of data items to determine the sequence of events much like putting together small pieces of a very large puzzle. That process would have been significantly faster and easier if there had been wider use of synchronized data recording devices.

A situation that occurs infrequently but is of utmost concern when it does occur is called a rare-event situation. The validation of the performance of a real-time computer system in a rare event situation is a challenging task and requires models of the physical environment

Example: The sole purpose of a nuclear power plant monitoring and shutdown system is reliable performance in a peak-load alarm situation (a rare event). Hopefully, this rare event will never occur during the operational life of the plant.

Direct Digital Control

Many real-time computer systems must calculate the actuating variables for the actuators in order to control the controlled object directly (direct digital control – DDC), i.e., without any underlying conventional control system. Control applications are highly regular, consisting of an (infinite) sequence of control cycles, each one starting with sampling (observing) of the RT entities, followed by the execution of the control algorithm to calculate a new actuating variable, and subsequently by the output of the actuating variable to the actuator. The design of a proper control algorithm that achieves the desired control objective, and compensates for the random disturbances that perturb the controlled object, is the topic of the field of control engineering. In the next section on temporal requirements, some basic notions of control engineering will be introduced.

Man–Machine Interaction

A real-time computer system must inform the operator of the current state of the controlled object, and must assist the operator in controlling the machine or plant object. This is accomplished via the man–machine interface, a critical subsystem of major importance. Many severe computer-related accidents in safety-critical realtime systems have been traced to mistakes made at the man–machine interface [Lev95].

Example: Mode confusion at the man–machine interface of an aircraft has been identified to be the cause of major aircraft accidents [Deg95].

Most process-control applications contain, as part of the man–machine interface, an extensive data logging and data reporting subsystem that is designed according to the demands of the particular industry.

Example: In some countries, the pharmaceutical industry is required by law to record and store all relevant process parameters of every production batch in an archival storage in order that the process conditions prevailing at the time of a production run can be reexamined in case a defective product is identified on the market at a later time.

Man–machine interfacing has become such an important issue in the design of computer-based systems that a number of courses dealing with this topic have been developed. In the context of this book, we will

introduce an abstract man–machine interface in Sect. 4.5.2, but we will not cover its design in detail. The interested reader is referred to standard textbooks on user interface design.

Temporal Requirements

Where Do Temporal Requirements Come from? The most stringent temporal demands for real-time systems have their origin in the requirements of control loops, e.g., in the control of a fast process such as an automotive engine. The temporal requirements at the man–machine interface are, in comparison, less stringent because the human perception delay, in the range of 50–100 ms, is orders of magnitude larger than the latency requirements of fast control loops.

A Simple Control Loop. Consider the simple control loop depicted in Fig. 1.3 consisting of a vessel with a liquid, a heat exchanger connected to a steam pipe, and a controlling computer system. The objective of the computer system is to control the valve (control variable) determining the flow of steam through the heat exchanger such that the temperature of the liquid in the vessel remains within a small range around the set point selected by the operator. The focus of the following discussion is on the temporal properties of this simple control loop consisting of a controlled object and a controlling computer system.

The Controlled Object. Assume that the system of Fig. 1.3 is in equilibrium. Whenever the steam flow is increased by a step function, the temperature of the liquid in the vessel will change according to Fig. 1.4 until a new equilibrium is reached. This response function of the temperature in the vessel depends on the environmental conditions, e.g., the amount of liquid in the vessel, and the flow of steam through the heat exchanger, i.e., on the dynamics of the controlled object. (In the following section, we will use d to denote a duration and t to denote an instant, i.e., a point in time).

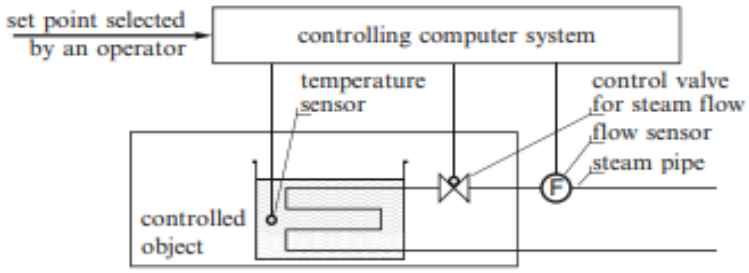


Fig. 1.3 A simple control loop

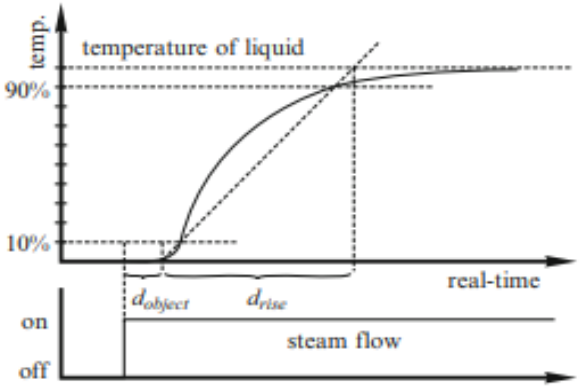


Fig. 1.4 Delay and rise time of the step response

There are two important temporal parameters characterizing this elementary step response function, the object delay d object (sometimes called the lag time or lag) after which the measured variable temperature begins to rise (caused by the initial inertia of the process and the instrumentation, called the process lag) and the rise time d rise of the temperature until the new equilibrium state has been reached. To determine the object delay d object and the rise time d rise from a given experimentally recorded shape of the step-response function, one finds the two points in time where the response function has reached 10% and 90% of the difference between the two stationary equilibrium values. These two points are connected by a straight line (Fig. 1.4). The significant points in time that characterize the object delay d object and the rise time d rise of the step response function are constructed by finding the intersection of this straight line with the two horizontal lines that denote the two liquid temperatures that correspond to the stable equilibrium states before and after the application of the step function.

Controlling Computer System. The controlling computer system must sample the temperature of the vessel periodically to detect any deviation between the intended value and the actual value of the

controlled variable temperature. The constant duration between two sampling points is called the sampling period d sample and the reciprocal 1/d sample is the sampling frequency, f sample. A rule of thumb says that, in a digital system which is expected to behave like a quasi-continuous system, the sampling period should be less than one-tenth of the rise time d rise of the step response function of the controlled object, i.e., d sample < (d rise/10). The computer compares the measured temperature to the temperature set point selected by the operator and calculates the error term. This error term forms the basis for the calculation of a new value of the control variable by a control algorithm. A given time interval after each sampling point, called the computer delay, the controlling computer will output this new value of the actuating variable to the control valve, thus closing the control loop. The delay d computer should be smaller than the sampling period d sample. The difference between the maximum and the minimum values of the delay of the computer is called the jitter of the computer delay, Dd computer. This jitter is a sensitive parameter for the quality of control. The dead time of the control loop is the time interval between the observation of the RT entity and the start of a reaction of the controlled object due to a computer action based on this observation. The dead time is the sum of the controlled object delay d object, which is in the sphere of control of the controlled object and is thus determined by the controlled object's dynamics, and the computer delay d computer, which is determined by the computer implementation. To reduce the dead time in a control loop and to improve the stability of the control loop, these delays should be as small as possible. The computer delay d computer is defined by the time interval between the sampling points, i.e., the observation of the controlled object, and the use of this information (see Fig. 1.5), i.e., the output of the corresponding actuator signal, the actuating variable, to the controlled object.

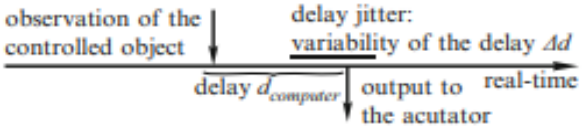


Fig. 1.5 Delay and delay jitter

Table 1.1 Parameters of an elementary control loop

Symbol	Parameter	Sphere of control	Relationships
d^{object}	Controlled object delay	Controlled object	Physical process
d^{rise}	Rise time of step response	Controlled object	Physical process
d^{sample}	Sampling period	Computer	$d^{sample} << d^{rise}$
$d^{computer}$	Computer delay	Computer	$d^{computer} < d^{sample}$
$\Delta d^{computer}$	Jitter of the computer delay	Computer	$\Delta d^{computer} << d^{computer}$
$d^{deadtime}$	Dead time	Computer and controlled object	$d^{computer} + d^{object}$

Apart from the necessary time for performing the calculations, the computer delay is determined by the time required for communication and the reaction time of the actuator.

Parameters of a Control Loop. Table 1.1 summarizes the temporal parameters that characterize the elementary control loop depicted in Fig. 1.3. In the first two columns we denote the symbol and the name of the parameter. The third column denotes the sphere of control in which the parameter is located, i.e., what subsystem determines the value of the parameter. Finally, the fourth column indicates the relationships between these temporal parameters.

Minimal Latency Jitter

The data items in control applications are state-based, i.e., they contain images of the RT entities. The computational actions in control applications are mostly time triggered, e.g., the control signal for obtaining a sample is derived from the progression of time within the computer system. This control signal is thus in the sphere of control of the computer system. It is known in advance when the next control action must take place. Many control algorithms are based on the assumption that the delay jitter Dd computer is very small compared to the delay d computer, i.e., the delay is close to constant. This assumption is made because control algorithms can be designed to compensate a known constant delay. Delay jitter brings an additional uncertainty into the control loop that has an adverse effect on the quality of control. The jitter Dd can be seen as an uncertainty about the instant when the RT-entity was observed. This jitter can be interpreted as causing an additional value error DT of the measured variable temperature T as shown in Fig. 1.6. Therefore, the delay jitter should always be a small fraction of the delay, i.e., if a delay of 1 ms is demanded then the delay jitter should be in the range of a few ms [SAE95].

Minimal Error-Detection Latency

Hard real-time applications are, by definition, safety-critical. It is therefore important that any error within the control system, e.g., the loss or corruption of a message or the failure of a node, is detected within a short time with a very high probability. The required error-detection latency must be in the same order of magnitude as the sampling period of the fastest critical control loop. It is then possible to perform some corrective action, or to bring the system into a safe state, before the consequences of an error can cause any severe system failure. Almost-no-jitter systems will have shorter guaranteed error-detection latencies

than systems that allow for jitter.

Dependability Requirements

The notion of dependability covers the meta-functional attributes of a computer system that relate to the quality of service a system delivers to its users during an extended interval of time. (A user could be a human or another technical system.) The following measures of dependability attributes are of importance,

Reliability

The Reliability R(t) of a system is the probability that a system will provide the specified service until time t, given that the system was operational at the beginning, i.e., t ¼ to. The probability that a system will fail in a given interval of time is expressed by the failure rate, measured in FITs (Failure In Time). A failure rate of 1 FIT means that the mean time to a failure (MTTF) of a device is 109 h, i.e., one failure occurs in about 115,000 years. If a system has a constant failure rate of l failures/h, then the reliability at time t is given by RðtÞ ¼ expðlðt toÞÞ; temp. jitter = Δd additional measurement error ΔT ΔT dT(t) dt real-time Fig. 1.6 The effect of jitter on the measured variable T 10 1 The Real-Time Environment where t to is given in hours. The inverse of the failure rate 1/l ¼ MTTF is called the Mean-Time-To-Failure MTTF (in hours). If the failure rate of a system is required to be in the order of 109 failures/h or lower, then we speak of a system with an ultrahigh reliability requirement.

Safety

Safety is reliability regarding critical failure modes. A critical failure mode is said to be malign, in contrast with a noncritical failure, which is benign. In a malign failure mode, the cost of a failure can be orders of magnitude higher than the utility of the system during normal operation. Examples of malign failures are: an airplane crash due to a failure in the flight-control system, and an automobile accident due to a failure of a computer-controlled intelligent brake in the automobile. Safety critical (hard) real-time systems must have a failure rate with regard to critical failure modes that conforms to the ultrahigh reliability requirement.

Example: Consider the example of a computer-controlled brake in an automobile. The failure rate of a computer-caused critical brake failure must be lower than the failure rate of a conventional braking

system. Under the assumption that a car is operated about 1 h per day on the average, one safety-critical failure per million cars per year translates into a failure rate in the order of 109 failures/h.

Similarly low failure rates are required in flight-control systems, train-signaling systems, and nuclear power plant monitoring systems.

Certification. In many cases the design of a safety-critical real-time system must be approved by an independent certification agency. The certification process can be simplified if the certification agency can be convinced that:

1. The subsystems that are critical for the safe operation of the system are protected by fault-containment mechanisms that eliminate the possibility of error propagation from the rest of the system into these safety-critical subsystems.
2. From the point of view of design, all scenarios that are covered by the given load- and fault-hypothesis can be handled according to the specification without reference to probabilistic arguments. This makes a resource adequate design necessary.
3. The architecture supports a constructive modular certification process where the certification of subsystems can be done independently of each other.

At the system level, only the emergent properties must be validated. [Joh92] specifies the required properties for a system that is designed for validation:
1. A complete and accurate reliability model can be constructed. All parameters of the model that cannot be deduced analytically must be measurable in feasible time under test.
2. The reliability model does not include state transitions representing design faults; analytical arguments must be presented to show that design faults will not cause system failure. 3. Design tradeoffs are made in favor of designs that minimize the number of parameters that must be measured.

Maintainability

Maintainability is a measure of the time interval required to repair a system after the occurrence of a benign failure. Maintainability is measured by the probability M (d) that the system is restored within a time interval d after the failure. In keeping with the reliability formalism, a constant repair rate m (repairs

per hour) and a Mean Time to Repair (MTTR) are introduced to define a quantitative maintainability measure. There is a fundamental conflict between reliability and maintainability. A maintainable design requires the partitioning of a system into a set of field replaceable units (FRUs) connected by serviceable interfaces that can be easily disconnected and reconnected to replace a faulty FRU in case of a failure. A serviceable interface, e.g., a plug connection, has a significantly higher physical failure rate than a non-serviceable interface. Furthermore, a serviceable interface is more expensive to produce. In the upcoming field of ambient intelligence, automatic diagnosis and maintainability by an untrained end user will be important system properties that are critical for the market success of a product.

Availability

Availability is a measure of the delivery of correct service with respect to the alternation of correct and incorrect service. It is measured by the fraction of time that the system is ready to provide the service. Example: Whenever a user picks up the phone, the telephone switching system should be ready to provide the telephone service with a very high probability. A telephone exchange is allowed to be out of service for only a few minutes per year. In systems with constant failure and repair rates, the reliability (MTTF), maintainability (MTTR), and availability (A) measures are related by A ¼ MTTF MTTF =ð Þ þ MTTR : The sum MTTF þ MTTR is sometimes called the Mean Time Between Failures (MTBF). Figure 1.7 shows the relationship between MTTF, MTTR, and MTBF.

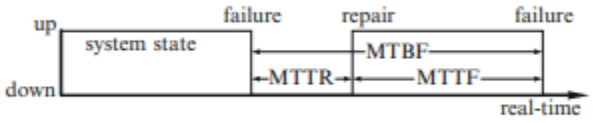


Fig. 1.7 Relationship between *MTTF*, *MTBF* and *MTTR*

A high availability can be achieved either by a long MTTF or by a short MTTR. The designer has thus some freedom in the selection of her/his approach to the construction of a high-availability system.

Security

A fifth important attribute of dependability – the security attribute – is concerned with the authenticity and integrity of information, and the ability of a system to prevent unauthorized access to information or services. There are difficulties in defining a quantitative security measure, e.g., the specification of a

standard burglar that takes a certain time to intrude a system. Traditionally, security issues have been associated with large databases, where the concerns are confidentiality, privacy, and authenticity of information. During the last few years, security issues have also become important in real-time systems, e.g., a cryptographic theft-avoidance system that locks the ignition of a car if the user cannot present the specified access code. In the Internet-of-Things (IoT), where the endpoints of the Internet are embedded systems that bridge the gap between the cyber world and physical word, security concerns are of crucial importance, since an intruder cannot only corrupt a data structure in a computer, but can cause harm in the physical environment.

Classification of Real-Time Systems

In this section we classify real-time systems from different perspectives. The first two classifications, hard real-time versus soft real-time (on-line), and fail-safe versus fail-operational, depend on the characteristics of the application, i.e., on factors outside the computer system. The second three classifications, guaranteed timeliness versus best effort, resource-adequate versus resource-inadequate, and event-triggered versus time-triggered, depend on the design and implementation of the computer application, i.e., on factors inside the computer system.

Hard Real-Time System Versus Soft Real-Time System

The design of a hard real-time system, which always must produce the results at the correct instant, is fundamentally different from the design of a soft-real time or an on-line system, such as a transaction processing system. In this section we will elaborate on these differences.

Table 1.2 Hard real-time versus soft real-time systems

Characteristic	Hard real-time	Soft real-time (on-line)
Response time	Hard-required	Soft-desired
Peak-load performance	Predictable	Degraded
Control of pace	Environment	Computer
Safety	Often critical	Non-critical
Size of data files	Small/medium	Large
Redundancy type	Active	Checkpoint–recovery
Data integrity	Short-term	Long-term
Error detection	Autonomous	User assisted

Table: compares the characteristics of hard real time systems versus soft real-time systems.

Response Time.

The demanding response time requirements of hard real-time applications, often in the order of milliseconds or less, preclude direct human intervention during normal operation or in critical situations. A hard real-time system must be highly autonomous to maintain safe operation of the process. In contrast, the response time requirements of soft real-time and on-line systems are often in the order of seconds. Furthermore, if a deadline is missed in a soft real-time system, no catastrophe can result.

Peak-load Performance.

In a hard real-time system, the peak-load scenario must be well defined. It must be guaranteed by design that the computer system meets the specified deadlines in all situations, since the utility of many hard real-time applications depends on their predictable performance during rare event scenarios leading to a peak load. This is in contrast to the situation in a soft-real time system, where the average performance is important, and a degraded operation in a rarely occurring peak load case is tolerated for economic reasons.

Control of Pace. A hard real-time computer system is often paced by the state changes occurring in the environment. It must keep up with the state of the environment (the controlled object and the human operator) under all circumstances. This is in contrast to an on-line system, which can exercise some control over the environment in case it cannot process the offered load.

Example: Consider the case of a transaction processing system, such as an airline reservation system. If the computer cannot keep up with the demands of the users, it just extends the response time and forces the users to slow down.

Safety. The safety criticality of many real-time applications has a number of consequences for the system designer. In particular, error detection and recovery must be autonomous such that the system can initiate appropriate recovery actions and arrive at a safe state within the time intervals dictated by the application without human intervention.

Size of Data Files. The real-time database that is composed of the temporally accurate images of the RT-entities is normally of small size. The key concern in hard real-time systems is on the short-term temporal accuracy of the real-time database that is invalidated by the flow of real-time. In contrast, in on-line transaction processing systems, the maintenance of the long-term integrity and availability of large data files is the key issue.

Redundancy Type. After an error has been detected in an on-line system, the computation is rolled back to a previously established checkpoint to initiate a recovery action. In hard real-time systems, roll-back/recovery is of limited utility for the following reasons:

1. It is difficult to guarantee the deadline after the occurrence of an error, since the roll-back/recovery action can take an unpredictable amount of time.
2. An irrevocable action that has been effected on the environment cannot be undone.
3. The temporal accuracy of the checkpoint data may be invalidated by the time difference between the checkpoint time and the instant now.

Fail-Safe Versus Fail-Operational

In many hard real-time systems one or more safe states, which can be reached in case of a system failure, can be identified. If such a safe state can be identified and quickly reached upon the occurrence of a failure, then we call the system fail-safe. Failsafeness is a characteristic of the controlled object, not the computer system. In failsafe applications the computer system must have a high error-detection coverage, i.e., the probability that an error is detected, provided it has occurred, must be close to one.

Example: In case a failure is detected in a railway signaling system, it is possible to set all signals to red and thus stop all the trains in order to bring the system to a safe state.

In many real-time computer systems a special external device, a watchdog, is provided to monitor the operation of the computer system. The computer system must send a periodic life-sign (e.g., a digital output of predefined form) to the watchdog. If this life-sign fails to arrive at the watchdog within the specified time interval, the watchdog assumes that the computer system has failed and forces the controlled object into a safe state. In such a system, timeliness is needed only to achieve high availability, but is not needed to maintain safety since the watchdog forces the controlled object into a safe state in case of a timing violation. There are, however, applications where a safe state cannot be identified, e.g., a flight control system aboard an airplane. In such an application the computer system must remain operational and provide a minimal level of service even in the case of a failure to avoid a catastrophe. This is why these applications are called fail-operational

Guaranteed-Response Versus Best-Effort If we start out with a specified fault- and load-hypothesis and deliver a design that makes it possible to reason about the adequacy of the design without reference to probabilistic arguments (even in the case of a peak load and fault scenario), then we can speak of a system with a guaranteed response. The probability of failure of a perfect system with guaranteed response is reduced to the probability that the assumptions about the peak load and the number and types of faults do not hold in reality. This probability is called assumption coverage [Pow95]. Guaranteed response systems require careful planning and extensive analysis during the design phase. If such an analytic response guarantee cannot be given, we speak of a best-effort design. Best-effort systems do not require a rigorous specification of the load- and fault-hypothesis. The design proceeds according to the principle best possible effort taken and the sufficiency of the design is established during the test and integration phases. It is difficult to establish that a best-effort design operates correctly in a rare-event scenario. At present, many non safety-critical real-time systems are designed according to the best-effort paradigm.

Resource-Adequate Versus Resource-Inadequate

Guaranteed response systems are based on the principle of resource adequacy, i.e., there are enough computing resources available to handle the specified peak load and the fault scenario. Many non safety-critical real-time system designs are based on the principle of resource inadequacy. It is assumed that the provision of sufficient resources to handle every possible situation is not economically viable, and that a dynamic resource allocation strategy based on resource sharing and probabilistic arguments about the expected load and fault scenarios is acceptable. It is expected that, in the future, there will be a paradigm shift to resource adequate designs in many applications. The use of computers in important volume based applications, e.g., in cars, raises both the public awareness as well as concerns about computer-related incidents, and forces the designer to provide convincing arguments that the design functions properly under all stated conditions. Hard real time systems must be designed according to the guaranteed response paradigm that requires the availability of adequate resources.

Event-Triggered Versus Time-Triggered

The distinction between event-triggered and time-triggered depends on the type of internal triggers and not the external behavior of a real-time system. A trigger is an event that causes the start of some action in the computer, e.g., the execution of a task or the transmission of a message. Depending on the triggering

mechanisms for the start of communication and processing actions in each node of a computer system, two distinctly different approaches to the design of the control mechanisms of real-time computer applications can be identified, event-triggered control and time-triggered control.

In event-triggered (ET) control, all communication and processing activities are initiated whenever a significant event other than the regular event of a clock tick occurs. In an ET system, the signaling of significant events to the central processing unit (CPU) of a computer is realized by the well-known interrupt mechanism. ET systems require a dynamic scheduling strategy to activate the appropriate software task that services the event.

In a time-triggered (TT) system, all activities are initiated by the progression of real-time. There is only one interrupt in each node of a distributed TT system, the periodic real-time clock interrupt. Every communication or processing activity is initiated at a periodically occurring predetermined tick of a clock. In a distributed TT real-time system, it is assumed that the clocks of all nodes are synchronized to form a global time that is available at every node. Every observation of the controlled object is time-stamped with this global time. The granularity of the global time must be chosen such that the time order of any two observations made anywhere in a distributed TT system can be established from their timestamps with adequate faithfulness [Kop09].

Example: The distinction between event-triggered and time-triggered can be explained by an example of an elevator control system. When you push a call button in the event triggered implementation, the event is immediately relayed to the interrupt system of the computer in order to start the action of calling the elevator. In a time-triggered system, the button push is stored locally, and periodically, e.g., every second, the computer asks to get the state of all push buttons. The flow of control in a time-triggered system is managed by the progression of time, while in an event-triggered system; the flow of control is determined by the events that happen in the environment or the computer system.

The Real-Time Systems Market

In a market economy, the cost/performance relation is a decisive parameter for the market success of any product. There are only a few scenarios where cost arguments are not the major concern. The total life-cycle cost of a product can be broken down into three rough categories: non-recurring development cost, production cost, and operation and maintenance cost. Depending on the product type, the distribution of the total life-cycle cost over these three cost categories can vary significantly. We will examine this life

cycle cost distribution by looking at two important examples of real-time systems, embedded real-time systems and plant-automation systems

Embedded Real-Time Systems

The ever-decreasing price/performance ratio of microcontrollers makes it economically attractive to replace conventional mechanical or electronic control system within many products by an embedded real-time computer system. There are numerous examples of products with embedded computer systems: cellular phones, engine controllers in cars, heart pacemakers, computer printers, television sets, washing machines, even some electric razors contain a microcontroller with some thousand instructions of software code. Because the external interfaces (particularly the man–machine interface) of the product often remain unchanged relative to the previous product generation, it is often not visible from the outside that a real-time computer system is controlling the product behavior.

Characteristics. An embedded real-time computer system is always part of a well-specified larger system, which we call an intelligent product. An intelligent product consists of a physical (mechanical) subsystem; the controlling embedded computer, and, most often, a man–machine interface. The ultimate success of any intelligent product depends on the relevance and quality of service it can provide to its users. A focus on the genuine user needs is thus of utmost importance.

Embedded systems have a number of distinctive characteristics that influence the system development process:

1. **Mass Production:** many embedded systems are designed for a mass market and consequently for mass production in highly automated assembly plants. This implies that the production cost of a single unit must be as low as possible, i.e., efficient memory and processor utilization are of concern.

2**. Static Structure:** the computer system is embedded in an intelligent product of given functionality and rigid structure. The known a priori static environment can be analyzed at design time to simplify the software, to increase the robustness, and to improve the efficiency of the embedded computer system. In many embedded systems there is no need for flexible dynamic software mechanisms. These mechanisms increase the resource requirements and lead to an unnecessary complexity of the implementation.

3. **Man–Machine Interface**: if an embedded system has a man–machine interface, it must be specifically designed for the stated purpose and must be easy to operate. Ideally, the use of the intelligent product should be self-explanatory, and not require any training or reference to an operating manual.

4. **Minimization of the Mechanical Subsystem:** to reduce the manufacturing cost and to increase the reliability of the intelligent product, the complexity of the mechanical subsystem is minimized.

5. **Functionality Determined by Software in Read-Only Memory (ROM):** the integrated software that often resides in ROM determines the functionality of many intelligent products. Since it is not possible to modify the software in a ROM after its release, the quality standards for this software are high.

6. **Maintenance Strategy:** many intelligent products are designed to be non maintainable, because the partitioning of the product into replaceable units is too expensive. If, however, a product is designed to be maintained in the field, the provision of an excellent diagnostic interface and a self-evident maintenance strategy is of importance.

7. **Ability to communicate:** many intelligent products are required to interconnect with some larger system or the Internet. Whenever a connection to the Internet is supported, the topic of security is of utmost concern.

8. **Limited amount of energy:** Many mobile embedded devices are powered by a battery. The lifetime of a battery load is a critical parameter for the utility of a system. A large fraction of the life-cycle cost of many intelligent products is in the production, i.e., in the hardware. The known a priori static configuration of the intelligent product can be used to reduce the resource requirements, and thus the production cost, and also to increase the robustness of the embedded computer system. Maintenance cost can become significant, particularly if an undetected design fault (software fault) requires a recall of the product, and the replacement of a complete production series.

Example: In we find the following laconic one-liner: General Motors recalls almost 300 K cars for engine software flaw.

Future Trends.

During the last few years, the variety and number of embedded computer applications have grown to the point that, by now, this segment is by far the most important one in the computer market. The embedded system market is driven by the continuing improvements in the cost/performance ratio of the semiconductor industry that makes computer-based control systems cost-competitive relative to their mechanical, hydraulic, and electronic counterparts. Among the key mass markets are the domains of consumer electronics and automotive electronics. The automotive electronics market is of particular interest, because of stringent timing, dependability, and cost requirements that act as technology catalysts.

Automotive manufacturers view the proper exploitation of computer technology as a key competitive element in the never-ending quest for increased vehicle performance and reduced manufacturing cost. While some years ago, the computer applications on board a car focused on non-critical body electronics or comfort functions, there is now a substantial growth in the computer control of core vehicle functions, e.g., engine control, brake control, transmission control, and suspension control. We observe an integration of many of these functions with the goal of increasing the vehicle stability in critical driving maneuvers. Obviously, an error in any of these core vehicle functions has severe safety implications. At present the topic of computer safety in cars is approached at two levels. At the basic level a mechanical system provides the proven safety level that is considered sufficient to operate the car. The computer system provides optimized performance on top of the basic mechanical system. In case the computer system fails cleanly, the mechanical system takes over. Consider, for example, an Electronic Stability Program (ESP). If the computer fails, the conventional mechanical brake system is still operational. Soon, this approach to safety may reach its limits for two reasons:

1. If the computer-controlled system is further improved, the magnitude of the difference between the performance of the computer-controlled system and the performance of the basic mechanical system is further increased. A driver who is used to the high performance of the computer-controlled system might consider the fallback to the inferior performance of the mechanical system a safety risk.
2. The improved price/performance of the microelectronic devices will make the implementation of fault-tolerant computer systems cheaper than the implementation of mixed computer/mechanical systems. Thus, there will be an economic pressure to eliminate the redundant mechanical system and to replace it with a computer system using active redundancy. The embedded system market is expected to grow significantly during the next 10 years. It is expected that many embedded systems will be connected to the Internet, forming the Internet of Things (IoT).

Plant Automation Systems Characteristics.

Historically, industrial plant automation was the first field for the application of real-time digital computer control. This is understandable since the benefits that can be gained by the computerization of a sizable plant are much larger than the cost of even an expensive process control computer of the late 1960s. In the early days, human operators controlled the industrial plants locally. With the refinement of industrial plant instrumentation and the availability of remote automatic controllers, plant monitoring and command facilities were concentrated into a central control room, thus reducing the number of operators required to

run the plant. In the 1970s, the next logical step was the introduction of central process control computers to monitor the plant and assist the operator in her/his routine functions, e.g., data logging and operator guidance. In the beginning, the computer was considered an add-on facility that was not fully trusted. It was the duty of the operator to judge whether a set point calculated by a computer made sense and could be applied to the process (open-loop control). In the next phase, Supervisory Control and Data Acquisition (SCADA) systems calculated the set-points for the programmable logic controllers (PLC) in the plant. With the improvement of the process models and the growth of the reliability of the computer, control functions have been increasingly allocated to the computer and gradually the operator has been taken out of the control loop (closed-loop control). Sophisticated non-linear control techniques, which have response time requirements beyond human capabilities, have been implemented.

Usually, every plant automation system is unique. There is an extensive amount of engineering and software effort required to adapt the computer system to the physical layout, the operating strategy, the rules and regulations, and the reporting system of a particular plant. To reduce these engineering and software efforts, many process control companies have developed a set of modular building blocks, which can be configured individually to meet the requirements of a customer. Compared to the development cost, the production cost (hardware cost) is of minor importance. Maintenance cost can be an issue if a maintenance technician must be on-site for 24 h in order to minimize the downtime of a plant.

Future Trends. The market of industrial plant automation systems is limited by the number of plants that are newly constructed or are refurbished to install a computer control system. During the last 20 years, many plants have already been automated. This investment must pay off before a new generation of computers and control equipment is installed. Furthermore, the installation of a new generation of control equipment in a production plant causes disruption in the operation of the plant with a costly loss of production that must be justified economically. This is difficult if the plant's efficiency is already high, and the margin for further improvement by refined computer control is limited.

The size of the plant automation market is too small to support the mass production of special application-specific components. This is the reason why many VLSI components that are developed for other application domains, such as automotive electronics, are taken up by this market to reduce the system cost. Examples of such components are sensors, actuators, real-time local area networks, and processing nodes. Already several process-control companies have announced a new generation of process-control

equipment that takes advantage the of lowpriced mass produced components that have been developed for the automotive market, such as the chips developed for the Controller Area Network (CAN).

Multimedia Systems Characteristics. The multimedia market is a mass market for specially designed soft and firm real-time systems. Although the deadlines for many multimedia tasks, such as the synchronization of audio and video streams, are firm, they are not hard deadlines. An occasional failure to meet a deadline results in a degradation of the quality of the user experience, but will not cause a catastrophe. The processing power required to transport and render a continuous video stream is large and difficult to estimate, because it is possible to improve a good picture even further. The resource allocation strategy in multimedia applications is thus quite different from that of hard real-time applications; it is not determined by the given application requirements, but by the amount of available resources. A fraction of the given computational resources (processing power, memory, bandwidth) is allocated to a user domain. Quality of experience considerations at the end user determine the detailed resource allocation strategy. For example, if a user reduces the size of a window and enlarges the size of another window on his multimedia terminal, then the system can reduce the bandwidth and the processing allocated to the first window to free the resources for the other window that has been enlarged. Other users of the system should not be affected by this local reallocation of resources.

Future Trends. The marriage of the Internet with smart phones and multimedia personal computers leads to many new volume applications. The focus of this book is not on multimedia systems, because these systems belong to the class of soft and firm real-time applications.

Examples of Real-Time Systems In this section, three typical examples of real-time systems are introduced. These examples will be used throughout the book to explain the evolving concepts. We start with an example of a very simple system for flow control to demonstrate the need for end-to-end protocols in process input/output.

Controlling the Flow in a Pipe It is the objective of the simple control system depicted in Fig. 1.8 to control the flow of a liquid in a pipe. A given flow set point determined by a client should be maintained despite changing environmental conditions. Examples for such changing conditions are the varying level of the liquid in the vessel or the temperature sensitive viscosity of the liquid. The computer interacts with

the controlled object by setting the position of the control valve. It then observes the reaction of the controlled object by reading the flow sensor F to determine whether the desired effect, the intended change of flow, has been achieved. This is a typical example of the necessary end-to-end protocol [Sal84] that must be put in place between the computer and the controlled object (see also Sect. 7.1.2).

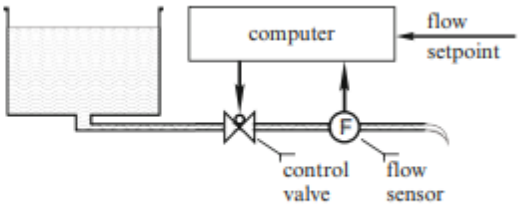


Fig. 1.8 Flow of liquid in a pipe

In a well-engineered system, the effect of any control action of the computer must be monitored by one or more independent sensors. For this purpose, many actuators contain a number of sensors in the same physical housing. For example, the control valve in Fig. 1.8 might contain a sensor, which measures the mechanical position of the valve in the pipe, and two limit switches, which indicate the firmly closed and the completely open positions of the valve. A rule of thumb is that there are about three to seven sensors for every actuator. The dynamics of the system in Fig. 1.8 is essentially determined by the speed of the control valve. Assume that the control valve takes 10 s to open or close from 0% to 100%, and that the flow sensor F has a precision of 1%. If a sampling interval of 100 ms is chosen, the maximum change of the valve position within one sampling interval is 1%, the same as the precision of the flow sensor. Because of this finite speed of the control valve, an output action taken by the computer at a given time will lead to an effect in the environment at some later time. The observation of this effect by the computer will be further delayed by the given latency of the sensor. All these latencies must either be derived analytically or measured experimentally, before the temporal control structure for a stable control system can be designed.

Engine Control

The task of an engine controller in an automobile engine is the calculation of the proper amount of fuel and the exact moment at which the fuel must be injected into the combustion chamber of each cylinder. The amount of fuel and the timing depend on a multitude of parameters: the intentions of the driver, articulated by the position of the accelerator pedal, the current load on the engine, the temperature of the

engine, the condition of the cylinder, and many more. A modern engine controller is a complex piece of equipment. Up to 100 concurrently executing software tasks must cooperate in tight synchronization to achieve the desired goal, a smoothly running and efficient engine with a minimal output of pollutants. The up- and downward moving piston in each cylinder of a combustion engine is connected to a rotating axle, the crankshaft. The intended start point of fuel injection is relative to the position of the piston in the cylinder, and must be precise within an accuracy of about 0.1 of the measured angular position of the crankshaft. The precise angular position of the crankshaft is measured by a number of digital sensors that generate a rising edge of a signal at the instant when the crankshaft passes these defined positions. Consider an engine that turns with 6,000 rpm (revolutions per minute), i.e., the crankshaft takes 10 ms for a 360 rotation. If the required precision of 0.1 is transformed into the time domain, then a temporal accuracy of 3 ms is required. The fuel injection is realized by opening a solenoid valve or a piezoelectric actuator that controls the fuel flow from a high-pressure reservoir into the cylinder. The latency between giving an open command to the valve and the actual point in time when the valve opens can be in the order of hundreds of ms, and changes considerably depending on environmental conditions (e.g., temperature). To be able to compensate for this latency jitter, a sensor signal indicates the point in time when the valve has actually opened. The duration between the execution of the output command by the computer and the start of opening of the valve is measured during every engine cycle. The measured latency is used to determine when the output command must be executed during the next cycle so that the intended effect, the start of fuel injection, happens at the proper point in time. This example of an engine controller has been chosen because it demonstrates convincingly the need for extremely precise temporal control. For example, if the processing of the signal that measures the exact position of the crankshaft in the engine is delayed by a few ms, the quality of control of the whole system is compromised. It can even happen that the engine is mechanically damaged if a valve is opened at an incorrect moment.

Rolling Mill

A typical example of a distributed plant automation system is the computer control of a rolling mill. In this application a slab of steel (or some other material, such as paper) is rolled to a strip and coiled. The rolling mill of Fig. 1.9 has three drives and some instrumentation to measure the quality of the rolled product. The distributed computer-control system of this rolling mill consists of seven nodes connected by a real-time communication system.

The most important sequence of actions – we call this a real-time (RT) transaction – in this application starts with the reading of the sensor values by the sensor computer. Then, the RT

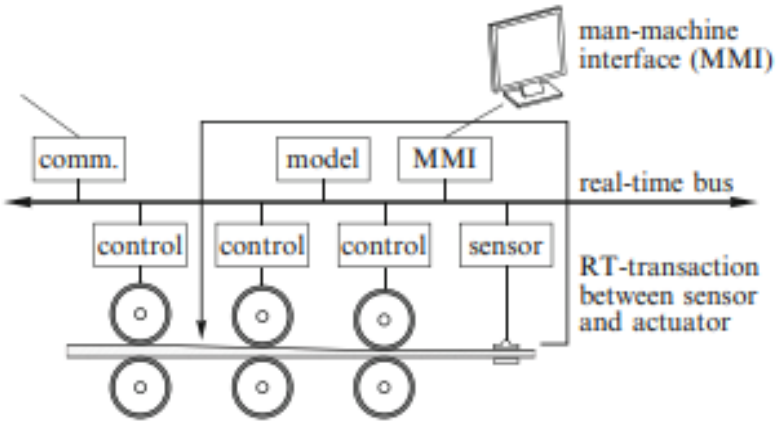


Fig. 1.9 An RT transaction

transaction passes through the model computer that calculates new set points for the three drives, and finally reaches the control computers to achieve the desired action by readjusting the rolls of the mill. The RT-transaction thus consists of three processing actions connected by two communication actions. The total duration of the RT transaction (bold line in Fig. 1.9) is an important parameter for the quality of control. The shorter the duration of this transaction, the better the control quality and the stability of the control loop, since this transaction contributes to the dead time of the critical control loop. The other important term of the dead time is the time it takes for the strip to travel from the drive to the sensor. A jitter in the dead time that is not compensated for will reduce the quality of control significantly. It is evident from Fig. 1.9 that the latency jitter in an event-triggered system is the sum of the jitter of all processing and communication actions that form the critical RT transaction. Note that the communication pattern among the nodes of this control system is multicast, not point-to-point. This is typical for most distributed real-time control systems. Furthermore, the communication between the model node and the drive nodes has an atomicity requirement. Either all of the drives are changed according to the output of the model, or none of them is changed. The loss of a message, which may result in the failure of a drive to readjust to a new position, may cause mechanical damage to the drive.

Simplicity

A recent report on Software for Dependable Systems: Sufficient Evidence? [Jac07] by the National Academies contains as one of its central recommendations: One key to achieving dependability at reasonable cost is a serious and sustained commitment to simplicity, including simplicity of critical functions and simplicity in system interactions. This commitment is often the mark of true expertise. We consider simplicity to be the antonym of cognitive complexity (in the rest of this book we mean cognitive complexity whenever we use the word complexity). In every-day life, many embedded systems seem to move in the opposite direction. The ever-increasing demands on the functionality, and the non-functional constraints (such as safety, security, or energy consumption) that must be satisfied by embedded systems lead to a growth in system complexity. In this chapter we investigate the notion of cognitive complexity and develop guidelines for building understandable computer systems. We ask the question: What does it mean when we say we understand a scenario? We argue that it is not the embedded system, but the models of the embedded system that must be simple and understandable relative to the background knowledge of the observer. The models must be based on clear concepts that capture the relevant properties of the scenario under investigation. The semantic content of a program variable is one of these concepts that we investigate in some detail. The major challenge of design is the building of an artifact that can be modeled at different levels of abstraction by models of adequate simplicity. This chapter is structured as follows. Section 2.1 focuses on the topic of cognition and problem solving and an elaboration of the two different human problem-solving subsystems, the intuitive-experiential subsystem and the analytic-rational subsystem. Concept formation and the conceptual landscape, that is the private knowledge base that a human develops over his lifetime, are discussed in Sect. 2.2. Section 2.3 looks at the essence of model building and investigates what makes a task difficult. Section 2.4 deals with the important topic of emergence in large systems.

Cognition

Cognition deals with the study of thought processes and the interpretation and binding of sensory inputs to the existing knowledge base of an individual [Rei10]. It is an interdisciplinary effort that stands between the humanities, i.e., philosophy, language studies, and social science on one side and the natural sciences, such as neural science, logic, and computer science on the other side. The study of model building, problem solving, and knowledge representation forms an important part of the cognitive sciences.

Problem Solving

Humans have two quite different mental subsystems for solving problems: the intuitive-experiential subsystem and the analytic-rational subsystem [Eps08]. Neuro-imaging studies have shown that these two subsystems are executed in two different regions of the human brain [Ami01]. Table 2.1 compares some of the distinguishing characteristics of these two subsystems.

Example: A typical task for the intuitive-experiential subsystem is face recognition, a demanding task that a baby at the age of 6 months can accomplish. A typical task for the analytic-rational subsystem is the confirmation of a proof of a mathematical theorem. The experiential subsystem is a preconscious emotionally-based subsystem that operates holistically, automatically, and rapidly, and demands minimal cognitive resources for its execution. Since it is nearly effortless, it is used most of the time.

Table Intuitive experiential versus analytic rational. Problem solving strategy

Intuitive experiential	Analytic rational
Holistic	Analytic
Emotional (what feels good)	Logical reason oriented (what is sensible?)
Unreflective associative connections	Cause and effect connections, causal chains
Outcome oriented	Process oriented
Behavior mediated by vibes from past experience	Behavior mediated by conscious appraisal of events
Encodes reality in concrete images, metaphors and narratives	Encodes reality in abstract symbols, words, and numbers
More rapid processing, immediate action	Slower processing, delayed action
Slow to change the fundamental structure: changes with repetitive or intense experience	Changes more rapidly, changes with the speed of thought
Experience processed passively and pre-consciously, seized by our emotions	Experience processed actively and consciously, in control of our thoughts
Self evidently valid: *seeing is believing*	Requires justification via logic and evidence

It is assumed that the experiential subsystem has access to a large coherent knowledge base that represents an implicit model of the world. This subjective knowledge base, which is one part of what we call the conceptual landscape of an individual, is mainly built up and maintained by experience and emotional events that are accumulated over the lifetime of an individual. Although this knowledge base is continually adapted and extended, its core structure is rather rigid and cannot be changed easily. Experiential reasoning is holistic and has the tendency to use limited information for general and broad classifications of scenarios and subjects (e.g., this is a good or bad person). The experiential system does assimilate the data about reality in a coherent stable conceptual framework. The concepts in this framework are mostly linked by unconscious associative connections, where the source of an association

is often unknown. The rational subsystem is a conscious analytic subsystem that operates according to the laws of causality and logic. Bunge [Bun08, p. 48] defines a causality relationship between a cause C and an event E as follows: If C happens, then (and only then) E is always produced by it. We try to get an understanding of a dynamic scenario by isolating a primary cause, suppressing seemingly irrelevant detail, and establishing a unidirectional causal chain between this primary cause and an observed effect. If cause and effect cannot be cleanly isolated, such as is the case in a feedback scenario, or if the relationship between cause and effect is nondeterministic (see also Sect. 5.6.1 on the definition of determinism), then it is more difficult to understand a scenario.

Example: Consider the analysis of a car accident that is caused by the skidding of a car. There are a number of conditions that must hold for skidding to occur: the speed of the car, the conditions of the road (e.g., icy road), the conditions of the tires, abrupt manoeuver by the driver, the non-optimal functioning of the computer based skid-control system, etc. In order to simplify the model of the situation (the reality is not simplified) we often isolate a primary cause, e.g., the speed, and consider the other conditions as secondary.

The rational subsystem is a verbal and symbolic reasoning system, driven by a controlled and noticeable mental effort to investigate a scenario. Adult humans have a conscious explicit model of reality in their rational subsystem, in addition to their implicit model of reality in the experiential subsystem. These two models of reality coincide to different degrees and form jointly the conceptual landscape of an individual. There seem to be a nearly unlimited set of resources in the experiential subsystem, whereas the cognitive resources that are available to the rational subsystem are limited [Rei10]. There are many subtle interrelationships between these two problem-solving subsystems, which form the extremes of a continuum of problem solving strategies where both systems cooperate to arrive at a solution. It is not infrequent that, after unsuccessful tries by the rational subsystem, at first a solution to a problem is produced unconsciously by the experiential subsystem. Afterwards this solution is justified by analytical and logical arguments that are constructed by the rational subsystem. Similarly, the significance of a new scenario is often recognized at first by the experiential subsystem. At a later stage it is investigated and analyzed by the rational subsystem and rational problem solving strategies are developed. Repeated encounters of similar problems – the accumulation of experience – effortful learning and drill move the problem-solving process gradually from the rational subsystem to the experiential subsystem, thus freeing

the cognitive resources that have previously been allocated to the problem solving process in the limited rational subsystem. There exist many practical examples that demonstrate this phenomenon: learning a foreign language, learning a new sport, or learning how to drive a car. It is characteristic for a domain expert that she/he has mastered this transition in her/his domain and mainly operates in the effortless experiential mode, where a fast, holistic and intuitive approach to problem solving dominates.

Example: A brain-imaging study of the chess-playing strategy of amateurs versus grand masters investigated the activity in different sections of the brain immediately after a chess move by the partner. The amateurs displayed the highest activity in the medial temporal lobe of the brain, which is consistent with the interpretation that their mental activity is focused on the rational analysis of the new move. The highly skilled grandmasters showed more activity in the frontal and parietal cortices, indicating that they are retrieving stored information about previous games from expert memory in order to develop an understanding of the scenario.

Definition of a Concept

In a changing world, knowledge about permanent and characteristic properties of objects and situations must be identified and maintained since such knowledge is of critical importance for survival. This knowledge is acquired by the process of abstraction, by which the particular is subordinated to the general, so that what is known about the general is applicable to many particulars. Abstraction is a fundamental task of the human cognitive system.

Example: Face recognition is an example for the powerful process of abstraction. Out of many particular images of the face of a person – varying angles of observation, varying distance, changing lighting conditions – characteristic permanent features of the face are identified and stored in order that they can be used in the future to recognize the face again. This demanding abstraction process is executed unconsciously, seemingly without effort, in the experiential subsystem. Only its results are delivered to the rational subsystem.

Abstraction forms categories, where a category is a set of elements that share common characteristic features. The notion of category is recursive: the elements of a category can themselves be categories. We thus arrive at a hierarchy of categories, going from the concrete to the abstract. At the lowest level we find immediate sensory experiences. A concept is a category that is augmented by a set of beliefs about its

relations to other categories [Rei10, pp. 261–300]. The set of beliefs relates a new concept to already existing concepts and provides for an implicit theory (a subjective mental model). As a new domain is penetrated, new concepts are formed and linked to the concepts that are already present in the conceptual landscape. A concept is a mental construct of the generalizable aspects of a known entity. It has an intension (What is 32 2 Simplicity the essence?) and an extension, answering the question as to which things and mental constructs are exemplars of the concept. A concept can also be considered as a unit of thought.

Cognitive Complexity

What do we mean when we say an observer understands a scenario? It means that the concepts and relationships that are employed in the representation of the scenario have been adequately linked with the conceptual landscape and the methods of reasoning of the observer. The tighter the links are, the better is the understanding. Understanding (and therefore simplicity) is thus a relation between an observer and a scenario, not a property of the scenario. We take the view of Edmonds [Edm00] that complexity can only be assigned to models of physical systems, but not to the physical systems themselves, no matter whether these physical systems are natural or man made. A physical system has a nearly infinite number of properties – every single transistor of a billion-transistor system-on-chip consists of a huge number of atoms that are placed at distinct positions in space. We need to abstract, to build models that leave out the seemingly irrelevant detail of the micro-level, in order to be able to reason about properties of interest to us at the macro-level. What then is a good measure for the cognitive complexity of a model? We are looking for a quantity that measures the cognitive effort needed to understand the model by a human observer. We consider the elapsed time needed to understand a model by a given observer a reasonable measure for the cognitive effort and thus for the complexity of a model relative to the observer. We assume that the given observer is representative for the intended user group of the model. According to the scientific tradition, it would be desirable to introduce an objective notion of cognitive complexity without reference to the subjective human experience. However, this does not seem to be possible, since cognitive complexity refers to a relation between an objective external scenario and the subjective internal conceptual landscape of the observer. The perceived complexity of a model depends on the relationship between the existing subjective conceptual landscape and the problem solving capability of the observer versus the concepts deployed in the representation of the model, the interrelations among these concepts and the notation used to represent these concepts. If the observer is an expert, such as the chess

grandmaster in the previous example, the experiential subsystem provides an understanding of the scenario within a short time and without any real effort. According to our metric, the scenario will be judged as simple. An amateur has to go through a tedious cause and-effect analysis of every move employing the rational subsystem that takes time and explicit cognitive effort. According to the above metric, the same chess scenario will be judged as complex.

There are models of behavior and tasks that are intrinsically difficult to comprehend under any kind of representation. The right column of Table 2.2 in Sect. 2.5 lists some characteristics of intrinsically difficult tasks. It may take a long time, even for an expert in the field, to gain an understanding of a model that requires the comprehension of the behavior of difficult tasks – if at all possible. According to the introduced metric, these models are classified as exceedingly complex. In order to gain an understanding of a large system we have to understand many models that describe the system from different viewpoints at different abstraction levels (see also Sect. 2.3.1). The cognitive complexity of a large system depends on the number and complexity of the different models that must be comprehended in order to understand the complete system. The time it takes to understand all these models can be considered as a measure for the cognitive complexity of a large system. Case studies about the understanding of the behavior of large systems have shown that the perceptually available information plays an important role for developing an understanding of a system [Hme04]. Invisible information flows between identified subsystems pose a considerable barrier to understanding. If every embedded system is one of its kind and no relationships between different instances of systems can be established, then there is hardly a chance that experience-based expert knowledge can be developed and the transition from the tedious and effortful rational subsystem to the effortless experiential subsystem can take place. One route to simplification is thus the development of a generic model of an embedded system that can be successfully deployed in many different domains at a proper level of abstraction. This model should contain few orthogonal mechanisms that are used recursively. The model must support simplification strategies and make public the internal information flow between identified subsystems, such that the process of gaining an understanding of the behavior is supported. By getting intimately acquainted with this model and gaining experience by using this model over and over again, the engineer can incorporate this model in the experiential subsystem and become an expert. It is one stated goal of this book to develop such a generic cross-domain model of embedded systems.

Simplification Strategies

The resources in the rational problem solving subsystem of humans, both in storage and processing capacity, are limited. The seminal work of Miller [Mil56] introduced a limit of five to seven chunks of information that can be stored in short-term memory at a given instant. Processing limitations are established by the relational complexity theory of Halford [Hal96]. Relational complexity is considered to correspond to the arity (number of arguments) of a relation. For example, binary relations have two arguments as in LARGER-THAN (elephant, mouse). The relational complexity theory states that the upper limits of adult cognition seem to be relations at the quaternary level. If a scenario requires cognitive resources that are beyond the given limits, then humans tend to apply simplification strategies to reduce the problem size and complexity in order that the problem can be tackled (possibly well, possibly inadequately) with the limited cognitive resources at hand. We know of four strategies to simplify a complex scenario in order that it can be processed by the limited cognitive capabilities of humans: abstraction, partitioning, isolation, and segmentation:

Abstraction refers to the formation of a higher-level concept that captures the essence of the problem-at-hand and reduces the complexity of the scenario by omitting irrelevant detail that is not needed, given the purpose of the abstraction. Abstraction is applied recursively.

Partitioning (also known as separation of concerns) refers to the division of the problem scenario into nearly independent parts that can be studied successfully in isolation. Partitioning is at the core of reductionism, the preferred simplification strategy in the natural sciences over the past 300 years. Partitioning is not always possible. It has its limits when emergent properties are at stake.

Isolation refers to the suppression of seemingly irrelevant detail when trying to find a primary cause. The primary cause forms the starting point of the causal chain that links a sequence of events between this primary cause and the observed effect. There is a danger that the simplification strategy of isolation leads to a too simplistic model of reality (see the example on skidding of a car in Sect. 2.1.1). l **Segmentation** refers to the temporal decomposition of intricate behavior into smaller parts that can be processed sequentially, one after the other. Segmentation reduces the amount of information that must be processed in parallel at any particular instant. Segmentation is difficult or impossible if the behavior is formed by highly concurrent processes, depends on many interdependent variables and is strongly non-linear, caused by positive or negative feedback loops.

The Conceptual Landscape

The notion of conceptual landscape, or the image [Bou61], refers to the personal knowledge base that is built up and maintained by an individual in the experiential and rational subsystem of the mind. The knowledge base in the experiential subsystem is implicit, while the knowledge base in the rational subsystem is explicit. The conceptual landscape can be thought of as a structured network of interrelated concepts that defines the world model, the personality, and the intentions of an individual. It is built up over the lifetime of an individual, starting from pre-wired structures that are established during the development of the genotype to the phenotype, and continually augmented as the individual interacts with its environment by exchanging messages via the sensory systems.

Concept Formation

The formation of concepts is governed by the following two principles [And01]:

The principle of utility states that a new concept should encompass those properties of a scenario that are of utility in achieving a stated purpose. The purpose is determined by the human desire to fulfill basic or advanced needs.

The principle of parsimony (also called Occam's razor) states that out of a set of alternative conceptualizations that are of comparable utility the one that requires the least amount of mental effort is selected.

There seems to be a natural level of categorization, neither too specific nor too general, that is used in human communication and thinking about a domain. We call the concepts at this natural level of categorization basic-level concepts.

Example: The basic level concept temperature is more fundamental than the sub-concept oil-temperature or the encompassing concept sensor data.

Studies with children have shown that basic-level concepts are acquired earlier than sub-concepts or encompassing concepts. As a child grows up it continually builds and adds to its conceptual landscape by observing regularities in the perceptions and utility in grouping properties of perceptions into new categories. These new categories must be interlinked with the already existing concepts in the child's mind to form a consistent conceptual landscape. By abstracting not only over perceptions, but also over already existing concepts, new concepts are formed. A new concept requires for its formation a number of experiences that have something in common and form the basis for the abstraction. Concept acquisition is

normally a bottom-up process, where sensory experiences or basic concepts are the starting point. Examples, prototypes and feature specification play an important role in concept formation. A more abstract concept is understood best bottom up by generalizations from a set of a suitable collection of examples of already acquired concepts. Abstract analysis and concrete interpretation and explanation should be intertwined frequently. If one remains only at a low-level of abstraction then the amount of non-essential detail is overwhelming. If one remains only at a high-level of abstraction, then relationships to the world as it is experienced are difficult to form. In the real world (in contrast to an artificial world), a precise definition of a concept is often not possible, since many concepts become fuzzy at their boundaries.

Example: How do you define the concept of dog? What are its characteristic features? Is a dog, which has lost a leg, still a dog?

Understanding a new concept is a matter of establishing connections between the new concept and already familiar concepts that are well embedded in the conceptual landscape.

Example: In order to understand the new concept of counterfeit money, one must relate this new concept to the following already familiar concepts: (1) the concept of money, (2) the concept of a legal system, (3) the concept of a national bank that is legalized to print money and (4) the concept of cheating. A counterfeit money bill looks like an authentic money bill. In this situation, examples and prototypes are of limited utility.

In the course of cognitive development and language acquisition, words (names) are associated with concepts. The essence of a concept associated with a word can be assumed to be the same within a natural language community (denotation), but different individuals may associate different shades of meaning with a concept (connotation), dependent on their individual existing conceptual landscape and the differing personal emotional experiences in the acquisition of the concept.

Example: If communicating partners refer to different concepts when using a word or if the concept behind a word is not well established in the (scientific) language community, (i.e., does not have a well-defined denotation), then effective communication among partners becomes difficult to impossible.

If we change the language community, the names of concepts will be changed, although the essence of the concept, its semantic content, remains the same. The names of concepts are thus relative to the context of discourse, while the semantic content remains invariant.

Example: The semantic content of the concept speed is precisely defined in the realm of physics. Different language communities give different names to the same concept: in German Geschwindigkeit, in French vitesse, in Spanish velocidad.

Scientific Concepts In the world of science, new concepts are introduced in many publications in order to be able to express new units of thought. Often these concepts are named by a mnemonic, leading to, what is often called, scientific jargon. In order to make an exposition understandable, new concepts should be introduced sparingly and with utmost care. A new scientific concept should have the following properties

Utility. The new concept should serve a useful well-defined purpose.

Abstraction and Refinement. The new concept should abstract from lower-level properties of the scenario under investigation. It should be clear which properties are not parts of the concept. In the case of refinement of a basic-level concept, it should be clearly stated what additional aspects are considered in the refined concept.

Precision. The characteristic properties of the new concept must be precisely defined.

Identity. The new concept should have a distinct identity and should be significantly different from other concepts in the domain.

Stability. The new concept should be usable uniformly in many different contexts without any qualification or modification.

Analogy. If there is any concept in the existing conceptual landscape that is, in some respects, analogous to the new concept, this similarity should be pointed out. The analogy helps to establish links to the existing conceptual landscape of a user and facilitates understanding. According to Analogical reasoning mechanisms are important to virtually every area of higher cognition, including language comprehension, reasoning and creativity. Human reasoning appears to be based less on an application of formal laws of logic than on memory retrieval and analogy. The availability of a useful, well defined, and stable set of concepts and associated terms that are generally accepted and employed by the scientific community is a mark for the maturity of a scientific domain. An ontology is a shared taxonomy that classifies terms in a way useful to a specific application domain in which all participants share similar levels of understanding of the meaning of the terms. Progress in a field of science is intimately connected with concept formation and the establishment of a well-defined ontology.

Example: The main contributions of Newton in the field of mechanics are not only in the formulation of the laws that bear his name, but also in the isolation and conceptualization of the abstract notions power, mass, acceleration and energy out of an unstructured reality. Clear concept formation is an essential prerequisite for any formal analysis or formal verification of a given scenario. The mere replacement of fuzzy concepts by formal symbols will not improve the understanding.

The Concept of a Message

We consider a message as a basic concept in the realm of communication. A message is an atomic unit that captures the value domain and the temporal domain of a unidirectional information transport at a level of abstraction that is applicable in many diverse scenarios of human communication and machine communication. A basic message transport service (BMTS) transports a message from a sender to one or a set of receivers. The BMTS can be realized by different means, e.g., biological or electrical. For example, the message concept can be used to express the information flow from the human sensory system to the conceptual landscape of an individual. The message concept can also model the indirect high-level interactions of a human with his environment that are based on the use of language.

Example: We can model the sensory perception, e.g. of temperature, by saying that a message containing the sensed variable (temperature) is sent to the conceptual landscape. A message could also contain verbal information about the temperature at a location that is outside the realm of direct sensory experience. The message concept is also a basic concept in the domain of distributed embedded computer systems at the architecture level. If the BMTS between encapsulated subsystems is based on unidirectional temporally predictable multicast messages, then the data aspect, the timing aspect, the synchronization aspect, and the publication aspect are integrated in a single mechanism. The BMTS can be refined at a lower level of abstraction by explaining the transport mechanism. The transport mechanism could be wired or wireless. The information can be coded by different signals. These refinements are relevant when studying the implementation of the message mechanism at the physical level, but are irrelevant at a level where the only concern is the timely arrival of the information sent by one partner to another partner. A protocol is an abstraction over a sequence of rule-based message exchanges between communicating partners. A protocol can provide additional services, such as flow control or error detection. A protocol can be understood by breaking it down to the involved messages without the need to elaborate on the concrete transport mechanisms that are used.

Semantic Content of a Variable The concept of a variable, a fundamental concept in the domain of computing, is of such importance for the rest of the book that it justifies some special elaboration. A variable can be considered as a language construct that assigns an attribute to a concept. If the point in real-time, the instant, when this assignment is valid, is of relevance, then we call the variable a state variable. As time progresses, the attribute of a state variable may change, while the concept remains the same. A variable thus consists of two parts, a fixed part, the variable name (or the identifier), and a variable part called the value of the variable that is assigned to the variable. The variable name designates the concept that determines what we are talking about. In a given context, the variable name – which is analogous to the name of a concept in a natural language community – must be unique and point to the same concept at all communicating partners. The meaning that is conveyed by a variable is called the semantic content of the variable. As we will show in the latter part of this section, the semantic content of a variable is invariant to a change in representation. The requirement of semantic precision demands that the concept that is associated with a variable name and the domain of values of the variable are unambiguously defined in the model of the given application.

Example: Consider the variable name engine-temperature that is used in an automotive application. This concept is too abstract to be meaningful to an automotive engineer, since there are different temperatures in an automotive engine: the temperature of the oil, the temperature of the water, or the temperature in the combustion chamber of the engine. The unambiguous definition of a concept does not only relate to the meaning of the concept associated with the variable, but also to the specification of the domain of values of the variable. In many computer languages, the type of a variable, which is introduced as an attribute of the variable name, specifies primitive attributes of the value domain of the variable. These primitive attributes, like integer or floating point number, are often not sufficient to properly describe all relevant attributes of the value domain. An extension of the type system will alleviate the problem.

The Essence of Model Building

Given the rather limited cognitive capabilities of the rational subsystem of the human mind we can only develop a rational understanding of the world around us if we build simple models of those properties that are of relevance and interest to us and disregard (abstract from) detail that proves to be irrelevant for the given purpose. A model is thus a deliberate simplification of reality with the objective of explaining a chosen property of reality that is relevant for a particular

purpose.

Example: The purpose of a model in Celestial Mechanics is the explanation of the movements of the heavenly bodies in the universe. For this purpose it makes sense to introduce the abstract concept of a mass point and to reduce the whole diversity of the world to a single mass point in space in order that the interactions with other mass points (heavenly bodies) can be studied without any distraction by unnecessary detail. When a new level of abstraction (a new model) is introduced that successfully conceptualizes the properties relevant for the given purpose and disregards the rest, simplicity emerges. Such simplicity, made possible by the formation of proper concepts, give rise to new insights that are at the roots of the laws of nature. As Popper points out, due to the inherent imperfection of the abstraction and induction process, laws of nature can only be falsified, but never be proven to be absolutely correct.

Emergence

We speak of emergence when the interactions of subsystems give rise to unique global properties at the system level that are not present at the level of the subsystems. Non-linear behavior of the subsystems, feedback and feed forward mechanisms, and time delays are of relevance for the appearance of emergent properties. Up to now, the phenomenon of emergence is not fully understood and a topic of intense study.

Irreducibility

Emergent properties are irreducible, holistic, and novel – they disappear when the system is partitioned into its subsystem. Emergent properties can appear unexpectedly or they are planned. In many situations, the first appearance of the emergent properties is unforeseen and unpredictable. Often a fundamental revision of state-of-the-art models is required to get a better understanding of the conditions that lead to the intended emergence. In some cases, the emergent properties can be captured in a new conceptualization (model) at a higher level of abstraction resulting in an abrupt simplification of the scenario.

Example: The emergent properties of a diamond, such as brilliance and hardness, which are caused by the coherent alignment of the Carbon-atoms, are substantially different from the properties of graphite (which consists of the same atoms). We can consider the diamond with its characteristic properties a new concept, a new unit of thought, and forget about its composition and internal structure. Simplicity comes out as a result of the intricate interactions among the elements that help to generate a new whole with its new emergent properties.

UNIT-II

REAL TIME OPERATING SYSTEMS

Real-Time Operating Systems

Overview In a component-based distributed real-time system we distinguish two levels of system administration, the coordination of the message-based communication and resource allocation among the components and the establishment, coordination, and control of the concurrent tasks within each one of the components. The focus of this chapter is on the operating system and middleware functions within a component. In case the software core image is not permanently residing in a component (e.g.,in read-only memory), mechanisms must be provided for a secure boot of the component software via the technology-independent interface. Control mechanisms must be made available to reset, start, and control the execution of the component software at run-time. The software within a component will normally be organized in a set of concurrent tasks. Task management and inter-component task interactions have to be designed carefully in order to ensure temporal predictability and determinism. The proper handling of time and time-related signals is of special importance in real-time operating systems. The operating system must also support the programmer in establishing new message communication channels at run time and in controlling the access to the message-based interfaces of the components. Domain specific higher-level protocols, such as a simple request-reply protocol, that consist of a sequence of rule-based message exchanges, should be implemented in the middleware of a component. Finally, the operating system must provide mechanisms to access the local process input/output interfaces that connect a component to the physical plant. Since the value domain and the time-domain of the RT-entities in the physical plant are dense, but the representation of the values and times inside the computer is discrete, some inaccuracy in the representation of values and times inside the computer system cannot be avoided. In order to reduce the effects of these representation inaccuracies and to establish a consistent (but not fully faithful) model of the physical plant inside the computer system, agreement protocols must be executed at the interface between the physical world and cyberspace to create a consistent digital image of the external world inside the distributed computer system. A real-time operating system (OS) within a component must be temporally predictable. In contrast to operating systems for personal computers, a real-time OS should be deterministic und support the implementation of fault-tolerance by active replication. In safety-critical applications, the OS must be certified. Since the certification of the behavior of a dynamic control structure is difficult, dynamic mechanisms should be avoided wherever possible in safety-critical systems.

Inter-Component Communication

The information exchange of a component with its environment, i.e., other components and the physical plant, is realized exclusively across the four message-based interfaces. It is up the generic middleware and the component's

operating system to manage the access to these four message interfaces for inter-component communication.

Technology Independent Interface

In some sense, the technology independent interface (TII) is a meta-level interfacethat brings a new component out of the core-image of the software, the job, and the given embodiment, the component hardware into existence. The purpose of the TII is the configuration of the component and the control of the execution of the software within a component. The component hardware must provide a dedicated TII port for the secure download of a new software image onto a component. Periodically, the g-state of the component should be published at the TII in order to be able to check the contents of the g-state by a dedicated diagnostic component. A further TII port directly connected to the component hardware must allow the resetting of the component hardware and the restart of the component software at the next reintegration point with a relevant g-state that is contained in the reset message. The TII is also used to control the voltage and frequency of the component hardware, provided the given hardware supports

voltage-frequency scaling. Since malicious TII messages have the potential to destroy the correct operation of a component, the authenticity and integrity of all messages that are sent to the TII interface must be assured.

Linking Interface

The linking interface (LIF) of a component is the interface where the services of the component are provided during normal operation. It is the most important interface from the point of view of operation and of composability of the components.

Technology Dependent Debug Interface

In the domain of VLSI design, it is common practice to provide a dedicated interface port for testing and debugging, known as the JTAG port that has been standardized in IEEE standard 1149.1. Such a debugging port, the technology dependent debug interface (TDI), supports a detailed view inside a

component that is needed by a component-designer to monitor and change the internal variables of a component that are not visible at any one of the other interfaces. The component-local OS should support such a testing and debugging interface.

Generic Middleware

The software structure within a component is depicted in Fig.. Between the local hardware-specific real-time operating system and the application software is the generic middleware (GM). The execution control messages that arrive at the TII (e.g., start task, terminate task, or reset the component hardware and restart the component with a relevant g-state) or are produced at the TII (e.g., periodic publication of the g-state) are interpreted inside a component by the standardized generic middleware (GM). The application software, written in a high-level language, accesses the operational message-based interfaces (the LIF and the local interface) by API system calls. The GM and the task-local operating system must manage the API system calls and the messages that arrive at the LIF and the commands that arrive via the TII messages. While the task-local operating system may be specific to a given component hardware, the GM layer provides standardized services, processes the standardized system control messages, and implements higher-level protocols.

Example: A high-level time-monitored request-reply protocol that is a unique concept at the level of the API requires two or more independent messages at the BMTS level and a set of local timer and operating system calls for its implementation. The GM implements this high-level protocol. It keeps track of all relevant messages and coordinates the timeouts and operating system calls.

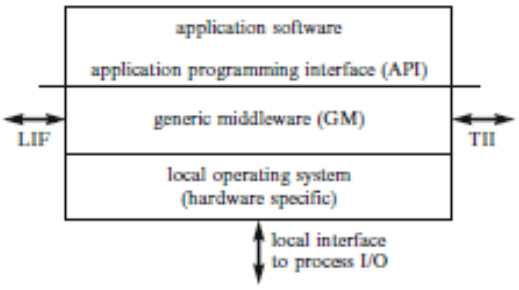


Fig. Software structure within a component

Task Management

In our model, the component software is assumed to be a unit of design, and a whole component is the smallest unit of fault-containment. The concurrent tasks within a component are cooperative and not competitive. Since the whole component is a unit of failure, it is not justified to design and implement resource intensive mechanisms to protect the component-internal tasks from each other. The component-internal operating system is thus a lightweight operating system that manages the task execution and the resource allocation inside a component. A task is the execution of a sequential program. It starts with reading of input data

and of its internal state and terminates with the production of the results and updated internal state. A task that does not have an internal state at its point of invocation is called a stateless task; otherwise, it is called a statefull task. Task management is concerned with the initialization, execution, monitoring, error handling, interaction, and termination of tasks.

Simple Tasks

If there is no synchronization point within a task, we call it a simple task (S-task), i.e., whenever an S-task is started, it can continue until its termination point is reached, provided the CPU is allocated to the task. Because an S-task cannot be blocked within the body of the task by waiting for an event external to the S-task, the execution time of an S-task is not directly dependent on the progress of other tasks in the node and can be determined in isolation. It is possible for the execution time of an S-task to be extended by indirect interactions, such as the preemption of the task execution by a task with higher priority. Depending on the triggering signal for the activation of a task, we distinguistime-triggered (TT) tasks and (ET) event-triggered tasks. A cycle is assigned to every TT-task and the task execution is started whenever the global time reaches the start of a new cycle. Event-triggered tasks are started whenever a start-event for the task occurs. A start event can be the completion of another task or an external event that is relayed to the operating system by an incoming message or by the interrupt mechanism.

In an entirely time-triggered system, off-line scheduling tools establish the temporal control structure of all tasks a priori. This temporal control structure is encoded in a Task-Descriptor List (TADL) that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time to guarantee mutual exclusion is not necessary.

Whenever the time reaches an entry point of the TADL, the dispatcher is activated. It performs the action that has been planned for this instant. If a task is started, the operating system informs the task of its activation time, which is synchronized within the cluster. After task termination, the operating system makes the results of the task available to other tasks. The application program interface (API) of an S-task in a TT system consists of three data structures and two operating system calls. The data structures are the input data structure, the output data structure, and the g-state data structure of the task (which is empty, in case the task is stateless). The system calls are TERMINATE TASK and ERROR. The TERMINATE TASK system call is executed by the task whenever the task has reached its normal termination point. In the case of an error that cannot be handled within the application task, the task
terminates its operation with the ERROR system call. In an event-triggered system, the evolving application scenario determines the sequence of task executions dynamically. Whenever a significant event happens, a task is released to the ready state, and the dynamic task scheduler is invoked. It is up to the scheduler to decide at run-time, which one of the ready tasks is selected for the next service by the CPU. Different dynamic algorithms for solving the scheduling problem are discussed in the following chapter. The WCET (Worst-Case Execution Time) of the scheduler contributes to the WCAO (Worst-Case Administrative Overhead) of the operating system. Significant events that cause the activation of an ET task can be:

1. An event from the node's environment, i.e., the arrival of a message or an interrupt from the controlled object, or
2. A significant event inside the component, i.e., the termination of a task or some other condition within a currently executing task, or
3. The progression of the clock to a specified instant. This instant can be specified either statically or dynamically.

An ET operating system that supports non-preemptive S-tasks will take a new scheduling decision after the currently running task has terminated. This simplifies task management for the operating system but severely restricts its responsiveness. If a significant event arrives immediately after the longest task has been scheduled, this event will not be considered until this longest task has completed. In an RT operating system that supports task preemption, each occurrence of a significant event can potentially activate a new task and cause an immediate interruption of the currently executing task. Depending on the outcome of the dynamic scheduling algorithm, the new task will be selected for execution or the interrupted task will be

continued. Data conflicts between concurrently executing S-tasks can be avoided if the operating system copies all input data required by this task from the global data area into a private data area of the task at the time of task invocation. If components are replicated, care must be taken that the preemption points at all replicas is at the same statement, otherwise replica determinism may be lost. The API of an operating system that supports event-triggered S-tasks requires more system calls than an operating system that only supports time-triggered tasks. Along with the data structures and the already introduced system calls of a TT system, the operating system must provide system calls to ACTIVATE (make ready) a new task, either immediately or at some future point in time. Another system call is needed to DEACTIVATE an already activated task.

Trigger Tasks

In a TT system, control always remains within the computer system. To recognize significant state changes outside the computer, a TT system must regularly monitor the state of the environment. A trigger task is a time-triggered S-task that evaluates a trigger condition on a set of temporally accurate state variables that reflect the current state of the environment. The result of a trigger task can be a control signal that activates another application task. Since the states, either external or internal, are sampled at the frequency of the trigger task, only those states with a duration longer than the sampling period of the trigger task are guaranteed to be observed.

Short-lived states, e.g., the push of a button, must be stored in a memory element (e.g., in the interface) for a duration that is longer than the sampling period of the trigger task. The periodic trigger task generates an administrative overhead in a TT system. The period of the trigger task must be smaller than the laxity (i.e., the difference between deadline and execution time) of an RT transaction that is activated by an event in the environment. If the laxity of the RT transaction is very small (<1 ms), the overhead associated with a trigger task can become in tolerable and the implementation of an interrupt is needed.

Complex Tasks

A task is called a complex task (C-Task) if it contains a blocking synchronization statement (e.g., a semaphore wait operation) within the task body. Such a wait operation may be required because the task must wait until a condition outside the task is satisfied, e.g., until another task has finished updating a common data structure or until input from a terminal has arrived. If a common data structure is

implemented as a protected shared object, only one task may update the data at any particular moment (mutual exclusion). All other tasks must be delayed by the wait
operation until the currently active task finishes its critical section. The worst-case execution time of a complex task in a node is therefore a global issue because it depends directly on the progress of the other tasks within the node or within the environment of the node.

The WCET of a C-task cannot be determined independently of the other tasks in the node. It can depend on the occurrence of an event in the node environment, as seen from the example of waiting for an input message. The timing analysis is not a local issue of a single task anymore; it becomes a global system issue. It is impossible to give an upper bound for the WCET of a C-task by analyzing the task code only. The application programming interface of a C-task is more complex than that of S-tasks. In addition to the three data structures already introduced, i.e., the input data structure, the output data structure, and the g-state data structure, the global data structures that are accessed at the blocking point must be defined. System calls must be provided that handle a WAIT-FOR-EVENT and a SIGNAL-EVENT. After the execution of the WAIT-FOR-EVENT, the task enters the blocked state and waits in the queue. The event occurrence releases the task from the blocked state. It must be monitored by a time-out task to avoid permanent blocking. The time-out task must be deactivated in case the awaited event occurs within the time-out period, otherwise the blocked task must be killed.

The Dual Role of Time

A real-time image must be temporally accurate at the instant of use. In a distributed system, the temporal accuracy can only be checked if the duration between the instant of observation of a RT-entity, observed by the sensor node, and the instant of use, determined by the actuator node, can be measured. This requires the availability of a global time base of proper precision among all involved nodes. If fault tolerance is required, two independent self-checking channels must be
provided to link an end system to the fault-tolerant communication infrastructure. The clock synchronization messages must be provided on both channels in order to tolerate the loss of any one of the channels. Every I/O signal has two dimensions, the value dimension and the temporal
dimension. The value dimension relates to the value of the I/O signal. The temporal dimension relates to the instant when the value was captured from the environment or released to the environment.

Example: In the context of hardware design, the value dimension is concerned with the contents of a register and the temporal dimension is concerned with the trigger signal, i.e., the control signal that determines when the contents of an I/O register are transferred to another subsystem.

An event that happens in the environment of a real-time computer can be looked upon from two different timing perspectives:

1. It defines the instant of a value change of an RT entity in the domain of time. The precise knowledge of this instant is an important input for the later analysis of the consequences of the event (time as data).
2. It may demand immediate action by the computer system to react as soon as possible to this event (time as control).

It is important to distinguish these two different roles of time. In the majority of situations, it is sufficient to treat time as data and only in the minority of cases, an immediate action of a computer system is required (time as control).

Example: Consider a computer system that must measure the time interval between start and finish during a downhill skiing competition. In this application it is sufficient to treat time as data and to record the precise time of occurrence of the start event and finish event. The messages that contain these two instants are transported to another computer that later calculates the difference. The situation of a train-control system that recognizes a red alarm signal, meaning the train should stop immediately, is different. Here, an immediate action isrequired as a consequence of the event occurrence. The occurrence of the event must initiate a control action without delay.

Time as Data

The implementation of time as data is simple if a global time-base of known precision is available in the distributed system. The observing component must include the timestamp of event occurrence in the observation message. We call a message that contains the timestamp of an event a timed message. The timed message can be processed at a later time and does not require any dynamic data dependent modification of the temporal control structure. Alternatively, if a field bus communication protocol with a known constant delay is used, the time of message arrival, corrected by this known delay, can be used to establish the send time of the message. The same technique of timed messages can be used on the output side. If an output signal must be invoked on the environment at a precise instant with a precision much

finer than the jitter of the output messages, a timed output message can be sent to the node controlling the actuator. This node interprets the time in the message and acts on the environment precisely at the intended instant. In a TT system that exchanges messages at a priori known instants with a fixed period between messages, the representation of time in a timed message can take advantage of this a priori information. The time value can be coded in fractions of the period of the message, thus increasing the data efficiency. For example, if an observation message is exchanged every 100 ms, a 7 bit time representation of time relative to the start of the period will identify the event with a granularity of better than 1 ms. Such a 7-bit representation of time, along with the additional bit to denote the event occurrence, can be packed into a single byte. Such a compact representation of the instant of event occurrence is very useful in alarm monitoring systems, where thousands of alarms are periodically queried by a cyclic trigger task.

The cycle of the trigger task determines the maximum delay of an alarm report (time as control), while the resolution of the timestamp informs about the exact occurrence of the alarm event (time as data) in the last cycle.

Example: In a single periodic TT-Ethernet message with a data field of 1,000 bytes and cycle time of 10 ms, 1,000 alarms can be encoded in a single message with a worst-case reaction time of 10 ms and an alarm resolution time of better than 100 ms. In a 100 Mbit/ s Ethernet system, these periodic alarm messages will generate a (background) system load of less than 1% of the network capacity. Such an alarm reporting system will not cause any increase in load if all 1,000 alarms occur at the same instant. If, in an event-triggered system, a 100 byte Ethernet message is sent whenever an alarm occurs, then the peak-load of 1,000 alarm messages will generate a load of 10% of the network capacity and a worstcase reaction time of 100 ms.

Time as Control

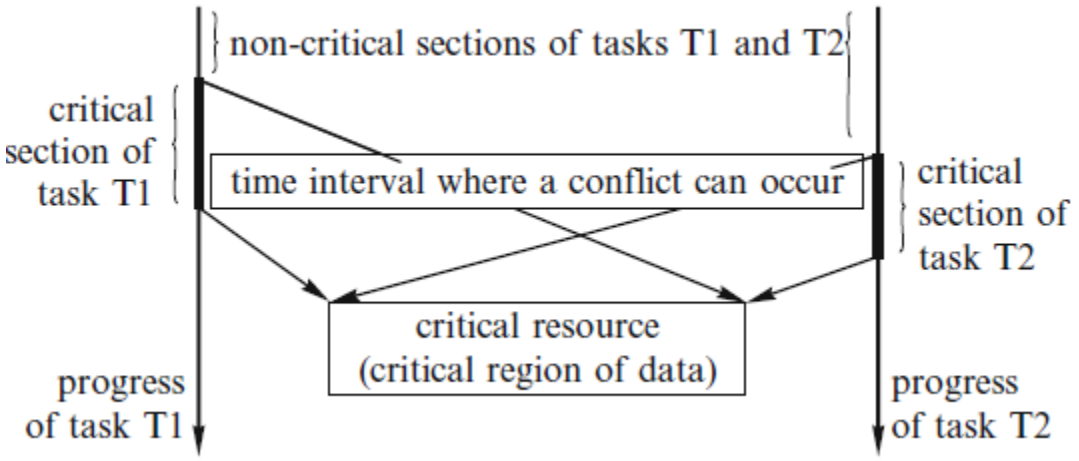
Time as control is more difficult to handle than time as data, because it may sometimes require a dynamic data-dependent modification of the temporal control structure. It is prudent to scrutinize the application requirements carefully to identify those cases where such a dynamic rescheduling of the tasks is absolutely necessary. If an event requires immediate action, the worst-case delay of the message transmission is a critical parameter. In an event-triggered protocol such as CAN,

the message priorities are used to resolve access conflicts to the common bus that result from nearly simultaneous events. The worst-case delay of a particular message can be calculated by taking the peak-load activation pattern of the message system into account [Tin95].

Example: The prompt reaction to an emergency shutdown request requires time to act as control. Assume that the emergency message is the highest priority message in a CAN system. In a CAN system, the worst-case delay of the highest priority message is bounded by the transmission duration of the longest message (which is about 100 bits), because a message transmission cannot be preempted.

Inter-task Interactions

Inter-task interactions are needed to exchange data among concurrently executing tasks inside a component such that progress towards the common goal can be achieved. There are two principal means to exchange data among a set of concurrently executing tasks: (1) by the exchange of messages and (2) by providing a shared region of data that can be accessed by more than one task. Within a component, shared data structures are widely used since this form of inter-task interaction can be implemented efficiently in a single component where the tasks cooperate. However, care must be taken that the integrity of data that is read or written concurrently by more than one task is maintained.



Figure, Critical task sections and critical data regions, depicts the problem. Two tasks, T1 and T2 access the same critical region of data. We call the interval during the program execution during which the

critical region of data is accessed the critical section of a task. If the critical sections of tasks overlap, bad things may occur. If the shared data is read by one task while it is modified by another task, then the reader may read inconsistent data. If the critical sections of two or more writing tasks overlap, the data may be corrupted.

The following three techniques can be applied to solve the problem:

1. Coordinated task schedules
2. The non-blocking write protocol
3. Semaphore operations

Coordinated Static Schedules

In a time-triggered system, the task schedules can be constructed in such a way that critical sections of tasks do not overlap. This is a very effective way to solve the problem, because:

1. The overhead of guaranteeing mutual exclusion is minimal and predictable.
2. The solution is deterministic.

Wherever possible, this solution should be selected.

The Non-blocking Write Protocol

If, however, the tasks with the critical sections are event-triggered, we cannot design conflict-free coordinated task schedules a priori. The non-blocking write (NBW) protocol is an example for a lock-free real-time protocol that ensures data integrity of one or more readers if only a single task is writing into the critical region of data. Let us analyze the operation of the NBW for the data transfer across the interface from the communication system to the host computer. At this interface, there is one writer, the communication system, and many readers, the tasks of the component. A reader does not destroy the information written by a writer, but a writer can interfere with the operation of the reader. In the NBW

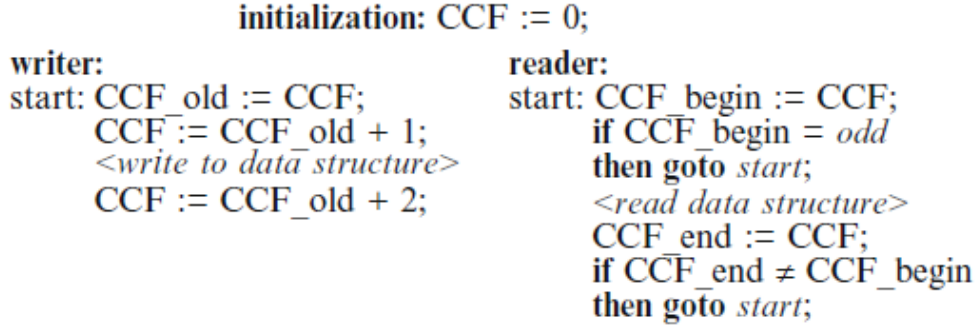


Fig. The non-blocking write (NBW) protocol

protocol, the real-time writer is never blocked. It will thus write a new version of the message into the critical data region whenever a new message arrives. If a reader reads the message while the writer is writing a new version, the retrieved message will contain inconsistent information and must be discarded. If the reader is able to detect the interference, then the reader can retry the read operation until it retrieves a consistent version of the data. It must be shown that the number of retries performed by the reader is bounded. The protocol requires a concurrency control field, CCF, for every critical data region. Atomic access to the CCF must be guaranteed by the hardware. The concurrency control field is initialized to zero and incremented by the writer before the start of the write operation. It is again incremented by the writer after the completion of the write operation. The reader starts by reading the CCF at the start of the read operation. If the CCF is odd, then the reader retries immediately because a write operation is in progress. At the end of the read operation, the reader checks whether the writer has changed the CCF during the read operation. If so, it retries the read operation again until it can read an uncorrupted version of the data structure. It can be shown that an upper bound for the number of read retries exists if the time between write operations is significantly longer than the duration of a write or read operation. The worst-case extension of the execution time of a typical real time task caused by the retries of the reader is only a few percent of the original worst-case execution time (WCET) of the task. Non-locking synchronization has been implemented in other real-time systems, e.g., in a multimedia system. It has been shown that systems with non locking synchronization achieve better performance than systems that lock the data.

Semaphore Operations

The classic mechanism to avoid data inconsistency is to enforce mutual exclusive execution of the critical task sections by a WAIT operation on a semaphore variable that protects the resource. Whenever one task is in its critical section, the other task must wait in a queue until the critical section is freed (explicit synchronization). The implementation of a semaphore-initialize operation is expensive, both regarding memory requirements and operating system processing overhead. If a process runs into a blocked semaphore, a context switch must be made. The process is put into a queue and is delayed until the other process finishes its critical section. Then, the process is dequeued and another context switch is made to reestablish the original context. If the critical region is very small (this is the case in many real time applications), the processing time for the semaphore operations can take significantly longer than the actual reading or writing of the common data. Both the NBW protocol and semaphore operation can lead

to a loss of replica determinism. The simultaneous access to CCF or a semaphore variable leads to a race condition that is resolved in an unpredictable manner in the replicas.

Process Input/Output

A transducer is a device that forms the interface between the plant (the physical world) and the computer (the cyber world). On the input side, a sensor transforms a mechanical or electrical quantity to a digital form, whereby the discreteness of the digital representation leads to an unavoidable error if the domain of the physical quantity is dense. The last bit of any digital representation of an analog quantity (both in the domain of value and time) is non-predictable, leading to potential inconsistencies in the cyber world representation if the same quantity is observed by two independent sensors. On the output side, a digital value is transformed to an appropriate physical signal by an actuator.

Analog Input/ Output

In a first step, many sensors of analog physical quantities produce analog signals in the standard 4–20 mA range (4 mA meaning 0% of the value range and 20 mA meaning 100% of the value range) that is then transformed to its digital form by an analog-to-digital (AD) converter. If a measured value is encoded in the 4–20 mA range, it is possible to distinguish a broken wire, where no current flows (0 mA), from a measured value of 0% (4 mA). Without special care, the electric-noise level limits the accuracy of any analog control signal to about 0.1%. Analog-to-digital (AD) converters with a resolution of more than 10 bits require a carefully controlled physical environment that is not available in typical industrial applications. A 16-bit word length is thus more than sufficient to encode the value of an RT entity measured by an analog sensor. The time interval between the occurrence of a value in the RT entity and the presentation of this value by the sensor at the sensor/computer interface is

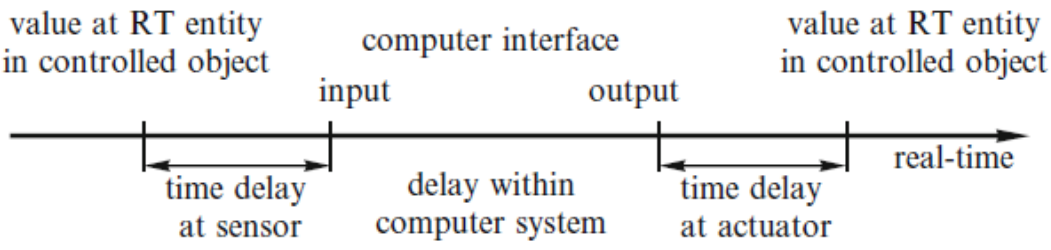


Fig. Time delay of a complete I/O transaction

determined by the transfer function of the particular sensor. The step response of a sensor, denoting the lag time and the rise time of the sensor, gives an approximation of this transfer function. When reasoning about the temporal accuracy of a sensor/actuator signal, the parameters of the transfer functions of the sensors and the actuators must be considered. They reduce the available time interval between the occurrence of a value at the RT entity and the use of this value for an output action by the computer. Transducers with short lag times increase the length of the temporal accuracy interval that is available to the computer system. In many control applications, the instant when an analog physical quantity is observed (sampled) is in the sphere of control of the computer system. In order to reduce the dead time of a control loop, the instant of sampling, the transmission of the sampled data to the control node and the transmission of the set-point data to the actuator node should be phase-aligned.

Digital Input/Output

A digital I/O signal transits between the two states TRUE and FALSE. In many applications, the length of the time interval between two state changes is of semantic significance. In other applications, the moment when the transition occurs is important. If the input signal originates from a simple mechanical switch, the new stable state is not reached immediately but only after a number of random oscillations called the contact bounce, caused by the mechanical vibrations of the switch contacts. This contact bounce must be eliminated either by an analog lowpass filter or, more often, within the computer system by software tasks, e.g., debouncing routines. Due to the low price of a microcontroller, it is cheaper to debounce a signal by software techniques than by hardware mechanisms (e.g., a low pass filter). A number of sensor devices generate a sequence of pulse inputs, where each pulse carries information about the occurrence of an event. For example, distance measurements are often made by a wheel rolling along the object that must be measured. Every rotation of the wheel generates a defined number of pulses that can be converted to the distance traveled. The frequency of the pulses is an indication of the speed.

Interrupts

The interrupt mechanism empowers a device outside the sphere of control of the computer to govern the temporal control pattern inside the computer. This is a powerful and potentially dangerous mechanism that must be used with great care. Interrupts are needed when an external event requires a reaction time from the computer (time as control) that cannot be implemented efficiently with a trigger task. A trigger task

extends the response time of an RT transaction that is initiated by an external event by at most one period of the trigger task. Increasing the trigger task frequency can reduce this additional delay at the expense of an increased overhead has analyzed this increase in the overhead for the periodic execution of a trigger task as the required response time approaches the WCET of the trigger task. As a rule of thumb, only if the required response time is less than ten times the WCET of the trigger task, the implementation of an interrupt should be considered. If information about the precise instant of arrival of a message is required, but no immediate action has to be taken, an interrupt-controlled time-stamping mechanism implemented in hardware should be used. Such a mechanism works autonomously and does not interfere with the control structure of tasks at the operating system level.

Example: In the hardware implementation of the IEEE 1,588 clock synchronization protocol, a hardware mechanism autonomously generates the time-stamp of an arriving synchronization message .

In an interrupt-driven software system, a transient error on the interrupt line may upset the temporal control pattern of the complete node and may cause the violation of important deadlines. Therefore, the time interval between the occurrence of any two interrupts must be continuously monitored, and compared to the specified minimum duration between interrupting events.

Interrupts

The interrupt mechanism empowers a device outside the sphere of control of the computer to govern the temporal control pattern inside the computer. This is a powerful and potentially dangerous mechanism that must be used with great care. Interrupts are needed when an external event requires a reaction time from the computer (time as control) that cannot be implemented efficiently with a trigger task. A trigger task extends the response time of an RT transaction that is initiated by an external event by at most one period of the trigger task. Increasing the triggertask frequency can reduce this additional delay at the expense of an increased overhead. has analyzed this increase in the overhead for the periodic execution of a trigger task as the required response time approaches the WCET of the trigger task. As a rule of thumb, only if the required response time is less than ten times the WCET of the trigger task, the implementation of an interrupt should be considered. If information about the precise instant of arrival of a message is required, but no immediate action has to be taken, an interrupt-controlled time-stamping mechanism implemented

in hardware should be used. Such a mechanism works autonomously and does not interfere with the control structure of tasks at the operating system level.

Example: In the hardware implementation of the IEEE 1,588 clock synchronization protocol, a hardware mechanism autonomously generates the time-stamp of an arriving synchronization message .

In an interrupt-driven software system, a transient error on the interrupt line may upset the temporal control pattern of the complete node and may cause the violation of important deadlines. Therefore, the time interval between the occurrence of any two interrupts must be continuously monitored, and compared to the specified minimum duration between interrupting events.

Raw Data, Measured Data, and Agreed Data

The concepts of raw data, measured data, and agreed data have been introduced: raw data are produced at the digital hardware interface of the physical sensor. Measured data, presented in standard engineering units, are derived from one or a sequence of raw data samples by the process of signal conditioning. Measured data that are judged to be a correct image of the RT entity, e.g., after the comparison with other measured data elements that have been derived by

diverse techniques, are called agreed data. Agreed data form the inputs to control actions. In a safety critical system where no single point of failure is allowed to exist, an agreed data element may not originate from a single sensor. The challenge in the development of a safety critical input system is the selection and placement of the redundant sensors and the design of the agreement algorithms. We distinguish two types of agreement, syntactic agreement and semantic agreement.

Syntactic Agreement

Assume that a two independent sensors measure a single RT entity. When the two observations are transformed from the domain of analog values to the domain of discrete values, a slight difference between the two raw values caused by a measurement error and digitalization error is unavoidable. These different raw data values will cause different measured values. A digitalization error also occurs in the time domain when the time of occurrence of an event in the controlled object is mapped into the discrete time of the computer. Even in the fault-free case, these different measured values must be reconciled in

some way to present an agreed view of the RT entity to the possibly replicated control tasks. In syntactic agreement, the agreement algorithm computes the agreed value without considering the context of the measured values. For example, the agreement algorithm always takes the average of a set of measured data values. If a Byzantine failure of one of the sensors must be tolerated, three additional sensors are needed.

Semantic Agreement

If the meanings of the different measured values are related to each other by a process model based on a priori knowledge about the relationships and the physical characteristics of the process parameters of the controlled object, we speak of semantic agreement. In semantic agreement, it is not necessary to duplicate or triplicate every sensor. Different redundant sensors observe different RT-entities. A model of the physical process relates these redundant sensor

readings to each other to find a set of plausible agreed values and to identify implausible values that indicate a sensor failure. Such an erroneous sensor value must be replaced by a calculated estimate of the most probable value at the given point in time, based on the inherent semantic redundancy in the set of measurements.

Example: A number of laws of nature govern a chemical process: the conservation of mass, the conservation of energy, and some known maximum speed of the chemical reaction. These fundamental laws of nature can be applied to check the plausibility of the measured data set. In case one sensor reading deviates significantly from all other sensors, a sensor failure is assumed and the failed value is replaced by an estimate of the correct value at this instant, to be able to proceed with the control of the chemical process.

Semantic agreement requires a fundamental understanding of the applied process technology. It is common that an interdisciplinary team composed of process technologists, measurement specialists, and computer engineers cooperates to find the RT entities that can be measured with good precision at reasonable cost. Typically, for every output value, about three to seven input values must be observed, not only to be able to diagnose erroneous measured data elements, but also to check the proper operation of the actuators. Independent sensors that observe the intended effect of the actuator must monitor the proper operation of every actuator. In engineering practice, semantic agreement of measured data values is more

important than syntactic agreement. As a result of the agreement phase, an agreed (and consistent) set of digital input values is produced. These agreed values, defined in the value domain and in the time domain, are then used by all (replicated) tasks to achieve a replica-determinate behavior of the control system.

Error Detection

A real-time operating system must support error detection in the temporal domain and error detection in the value domain by generic methods. Some of these generic methods are described in this section.

Monitoring Task Execution Times

A tight upper bound on the worst-case execution time (WCET) of a real-time task must be established during software development. This WCET must be monitored by the operating system at run time to detect transient or permanent hardware errors. In case a task does not terminate its operation within the WCET, the execution of the task is terminated by the operating system. It is upto the application to specify which action should be taken in case of an error.

Monitoring Interrupts

An erroneous external interrupt has the potential to disrupt the temporal control structure of the real-time software within the node. At design time, the minimum inter-arrival periods of interrupts must be known to be able to estimate the peak load that must be handled by the software system. At run time, this minimum inter-arrival period must be enforced by the operating system by disabling the interrupt line to reduce the probability of erroneous sporadic interrupts.

Double Execution of Tasks

Fault-injection experiments have shown that the double execution of tasks and the subsequent comparison of the results is a very effective method for the detection of transient hardware faults that cause undetected errors in the value domain. The operating system can provide the execution environment for the double execution of application tasks without demanding any changes to the application task perse. It is thus possible to decide at the time of system configuration which tasks should be executed twice and for which tasks it is sufficient to rely on a single execution.

Watchdogs

A fail-silent node will produce correct results or no results at all. The failure of a fail silent node can only be detected in the temporal domain. A standard technique is the provision of a watchdog signal (heart-beat) that must be periodically produced by the operating system of the node. If the node has access to the global time, the watchdog signal should be produced periodically at known absolute points in time. An outside observer can detect the failure of the node as soon as the watchdog signal disappears. A more sophisticated error detection mechanism that also covers part of the value domain is the periodic execution of a challenge-response protocol by a node. An outside error detector provides an input pattern to the node and expects a defined response pattern within a specified time interval. The calculation of this response pattern should involve as many functional units of the node as possible. If the calculated response pattern deviates from the a priori known correct result, an error of the node is detected.

UNIT-III

SYSTEM DESIGN

The Scheduling Problem

A hard real-time system must execute a set of concurrent real-time tasks in such a way that all time-critical tasks meet their specified deadlines. Every task needs computational, data, and other resources (e.g. input/output devices) to proceed. The scheduling problem is concerned with the allocation of these resources to satisfy all timing requirements.

Classification of Scheduling Algorithms

The following diagram presents a taxonomy of real-time scheduling algorithms.

Static Versus Dynamic Scheduling

A scheduler is called static (or pre-run-time) if it makes its scheduling decisions at compile time. It generates a dispatching table for the run-time dispatcher off-line. For this purpose it needs complete prior knowledge about the task-set characteristics, e.g., maximum execution times, precedence constraints, mutual exclusion constraints, and deadlines. The dispatching table contains all information the dispatcher needs at run time to decide at every point of the sparse time-base which task is to be scheduled next. The run-time overhead of the dispatcher is small. The system behavior is deterministic. A scheduler is called dynamic (or on-line) if it makes its scheduling decisions at run time, selecting one out of the current set of ready tasks. Dynamic schedulers are flexible and adapt to an evolving task scenario. They consider only the current task requests. The run-time effort involved in finding a schedule can be substantial. In general, the system behavior is non-deterministic.

Non-preemptive and Preemptive Scheduling In non-preemptive scheduling, the currently executing task will not be interrupted until it decides on its own to release the allocated resources. Non-preemptive scheduling is reasonable in a task scenario where many short tasks (compared to the time it takes for a context switch) must be executed. In preemptive scheduling, the currently executing taskmay be preempted, i. e., interrupted, if a more urgent task requests service.

Centralized Versus Distributed Scheduling In a dynamic distributed real-time system, it is possible to make all scheduling decisions at one central site or to problem.

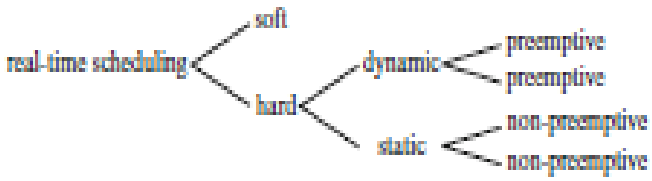


Fig. Taxonomy of real-time scheduling algorithms

The central scheduler in a distributed system is a critical point of failure. Because it requires up-to-date information on the load situations of all nodes, it can also contribute to a communication bottleneck.

Schedulability Test

A test that determines whether a set of ready tasks can be scheduled such that each task meets its deadline is called a schedulability test.We distinguish between exact, necessary, and sufficient schedulability tests. A scheduler is called optimal if it will always find a feasible schedule whenever it exists. Alternatively, a scheduler is called optimal, if it can find a schedule whenever a clairvoyant scheduler, i.e., a scheduler with complete knowledge of the future request times, can find a schedule. Garey and Johnson have shown that in nearly all cases of task dependency, even if there is only one common resource, the complexity of an exact schedulability test algorithm belongs to the class of NP-complete problems and is thus computationally intractable.

Sufficient schedulability test algorithms can be simpler at the expense of giving a negative result for some task sets that are, in fact, schedulable. A task set is definitely not schedulable if a necessary schedulability test gives a negative result. If a necessary schedulability test gives a positive result, there is still a probability that the task set may not be schedulable. The task request time is the instant when a request for a task execution is made. Based on the request times, it is useful to distinguish between two different task types: periodic and sporadic tasks. This distinction is important from the point of view of schedulability. If we start with an initial request, all future request times of a periodic task are known a priori by adding multiples of the known period to the initial request time. Let us assume that there is a task set {Ti} of periodic tasks with periods pi, deadline interval di, and execution time ci. The deadline interval is the duration between the deadline of a task and the task request instant, i.e., the instant when a task becomes ready for execution. We call the difference di -ci , the laxity li of a task. It is sufficient to examine

schedules of length of the least common multiples of the periods of these tasks, the schedule period, to determine schedulability.

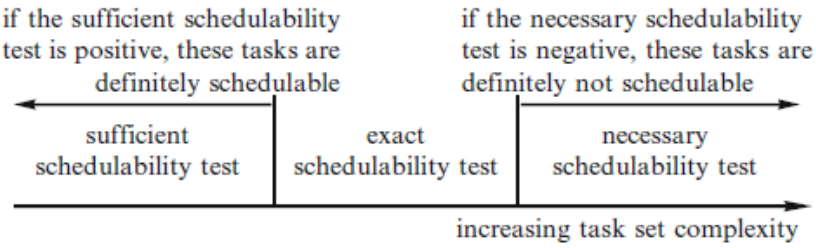


Fig. Necessary and sufficient schedulability test

A necessary schedulability test for a set of periodic tasks states that the sum of the utilization factors

$$\mu = \sum c_i/p_i \leqslant n,$$

must be less or equal to n, where n is the number of available processors. This is evident because the utilization factor of task Ti, mi, denotes the percentage of time task Ti requires service from a processor request times of sporadic tasks are not known a priori. To be schedulable, there must be a minimum interval between any two request times of sporadic tasks. Otherwise, the necessary schedulability test introduced above will fail. If there is no constraint on the request times of task activations, the task is called an a periodic task.

The Adversary Argument

Let us assume that a real-time computer system contains a dynamic scheduler with full knowledge of the past but without any knowledge about future request times of tasks. Schedulability of the current task set may depend on when a sporadic task will request service in the future. The adversary argument states that, in general, it is not possible to construct an optimal totally on-line dynamic scheduler if there are mutual exclusion constraints between a periodic and a sporadic task. The proof of the adversary argument is relatively simple. Consider two mutually exclusive tasks, task T1 is periodic and the other task T2 is sporadic, with the parameters given in Fig. The necessary schedulability test introduced above is satisfied, because

$$\mu = 2/4 + 1/4 = 3/4 \leqslant 1$$

Whenever the periodic task is executing, an adversary requests service for the sporadic task. Due to the mutual exclusion constraint, the sporadic task must wait until the periodic task is finished. Since the sporadic task has a laxity of 0, it will miss its deadline.

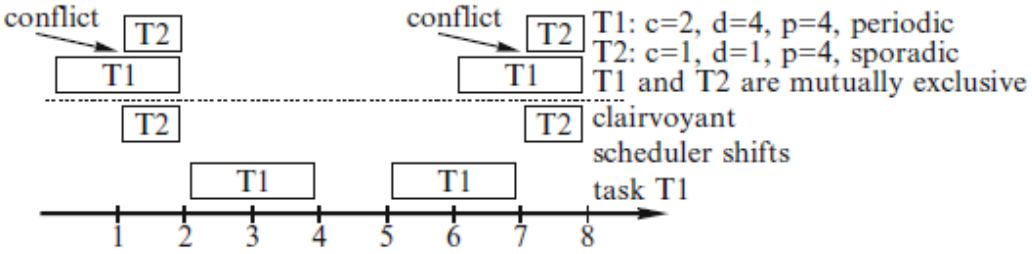


Fig. The adversary argument

The clairvoyant scheduler knows all the future request times of the sporadic tasks and at first schedules the sporadic task and thereafter the periodic task in the gap between two sporadic task activations (Fig. 10.3). The adversary argument demonstrates the importance of information about the future behavior of tasks for solving the scheduling problem. If the on-line scheduler does not have any further knowledge about the request times of the sporadic task, the dynamic scheduling problem is not solvable, although the processor capacity is more than sufficient for the given task scenario. The design of predictable hard realtime systems is simplified if regularity assumptions about the future scheduling requests can be made. This is the case in cyclic systems that restrain the points in time at which external requests are recognized by the computing system.

Worst-Case Execution Time

A deadline for completing an RT transaction can only be guaranteed if the worstcase execution times (WCET) of all application tasks and communication actions that are part of the RT-transaction are known a priori. The WCET of a task is a guaranteed upper bound for the time between task activation and task termination. It must be valid for all possible input data and execution scenarios of the task and should be a tight bound. In addition to the knowledge about the WCET of the application tasks, we must find an upper

bound for the delays caused by the administrative services of the operating system, the worst-case administrative overhead (WCAO). The WCAO includes all administrative delays that affect an application task but are not under the direct control of the application task (e.g., those caused by context switches, scheduling, cache reloading because of task preemption by interrupts or blocking, and direct memory access). This section starts with an analysis of the WCET of a non-preemptive simple task. We then proceed to investigate the WCET of a preemptive simple task before looking at the WCET of complex tasks and, finally, we discuss the state of the art regarding the timing analysis of real-time programs.

WCET of Simple Tasks

The simplest task we can envision is a single sequential S-task that runs on dedicated hardware without preemption and without requiring any operating system services. The WCET of such a task depends on:

1. The source code of the task
2. The properties of the object code generated by the compiler
3. The characteristics of the target hardware

In this section, we investigate the analytical construction of a tight worst-case execution time bound of such a simple task on hardware, where the execution time of an instruction is context independent.

Source Code Analysis. The first problem concerns the calculation of the WCET of a program written in a higher-level language, under the assumption that the maximum execution times of the basic language constructs are known and context independent. In general, the problem of determining the WCET of an arbitrary sequential program is unsolvable and is equivalent to the halting problem for Turing machines. Consider,

For example, the simple statement that controls the entry to a loop:

S: while(exp)

do loop;

It is not possible to determine a priori after how many iterations, if at all, the Boolean expression exp will evaluate to the value FALSE and when statement S will terminate. For the determination of the WCET to be a tractable problem there are a number of constraints that must be met by a the program:

1. Absence of unbounded control statements at the beginning of a loop
2. Absence of recursive function calls

3. Absence of dynamic data structures

The WCET analysis concerns only the temporal properties of a program. The temporal characteristics of a program can be abstracted into a WCET bound for every program statement using the known WCET bound of the basic language constructs.

For example, the WCET bound of a conditional statement
S: if(exp)
then S1
else S2;
can be abstracted as

$$T(S) = max\,[T(exp) + T(S_1),\ T(exp) + T(S_2)]$$

where T(S) is the maximum execution time of statement S, with T(exp), T(S1), and T(S2) being the WCET bounds of the respective constructs. Such a formula for reasoning about the timing behavior of a program is called a timing schema . The WCET analysis of a program which is written in a high-level language must determine which program path, i.e., which sequence of instructions, will be executed in the worst-case scenario. The longest program path is called the critical path. Because the number of programpaths normally grows exponentially with the program size, the search for the critical path can become intractable if the search is not properly guided and the search space is not reduced by excluding infeasible paths.

Compiler Analysis.

The next problem concerns the determination of the maximum execution time of the basic language constructs of the source language under the assumption that the maximum execution times of the machine language commands are known and context independent. For this purpose, the code generation strategy of the compiler must be analyzed, and the timing information that is available at the source code level must be mapped into the object code representation of the program such that an object-code timing analysis tool can make use of this information.

Execution Time Analysis.

The next problem concerns the determination of the worst-case execution time of the commands on the target hardware. If the processor of the target hardware has fixed instruction execution times, the duration of the hardware instructions can be found in the hardware documentation and can be retrieved by an elementary table look-up. Such a simple approach does not work if the target hardware is a modern RISC processor with pipelined execution units and instruction/data caches. While these architectural features result in significant performance improvements, they also introduce a high level of unpredictability. Dependencies among instructions can cause pipeline hazards, and cache misses will lead to a significant delay of the instruction execution. To make things worse, these two effects are not independent. A significant amount of research deals with the execution time analysis on machines with pipelines and caches. The excellent survey article by Wilhelm et al. presents the state of the art of WCET analysis in research and industry and describes many of the tools available for the support of WCET analysis.

Pre-emptive S-Tasks.

If a simple task (S task) is preempted by another independent task, e.g., a higher priority task that must service a pending interrupt, the execution time of the S-task under consideration is extended by three terms:

1. The WCET of the interrupting task
2. The WCET of the operating system required for context switching
3. The time required for reloading the instruction cache and the data cache of the processor whenever the context of the processor is switched

We call the sum of the worst-case delays caused by the context switch (2), and the cache reloading (3) the Worst-Case Administrative Overhead (WCAO) of a task preemption. The WCAO is an unproductive administrative operating system overhead that is avoided if task preemption is forbidden. The additional delay caused by the preemption of task A by task B is the WCET of the independent task B and the sum of the two WCAOs for the two context switches (shaded area in Fig. 10.4). The times spent in Microarchitecture-1 and Microarchitecture-2 are the delays caused by cache reloading. The Microarchitecture- 2 time of the first context switch is part of the WCET of task B, because task B

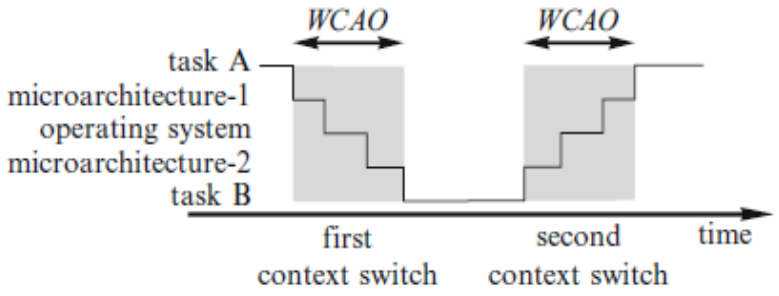


Fig. Worst-case administrative overhead (WCAO) of a task preemption

is assumed to start on an empty cache. The second context switch includes the cache reload time of task A, because in a non-preemptive system, this delay would not occur. In many applications with modern processors, the micro-architecture delays can be the significant terms that determine the cost of task preemption because the WCET of the interrupting task is normally quite short. The problem of WCAO analysis in operating systems is studied in [Lv09].

WCET of Complex Tasks

We now turn to the WCET analysis of a preemptive complex task (C-task) that accesses protected shared objects. The WCET of such a task depends not only on behavior of the task itself, but also on the behavior of other tasks and the operating system of the node. WCET analysis of a C-task is therefore not a local problem of a single task, but a global problem involving all the interacting tasks within a node. In addition to the delays caused by the task preemption (which was analyzed in the previous section), an additional delay that originates from the direct interactions caused by the intended task dependencies (mutual exclusion, precedence) must be considered. In the last few years, progress has been made in coping with the direct interactions caused by the intended task dependencies – e.g., access to protected shared objects controlled by the priority ceiling protocol [Sha94].

Anytime Algorithms

In practice, the time difference between the best-case execution time (BCET) and a guaranteed upper bound for the worst-case execution time (WCET) of a task can be substantial. Anytime algorithms are algorithms that use this time difference to improve the quality of the result as more execution time is provided Anytime algorithms consist of a root segment that calculates a first approximation of the result

of sufficient quality and a periodic segment that improves the quality of the previously calculated result. The periodic segment is executed repeatedly until the deadline, i.e. the guaranteed worst-case execution time of the root segment, is reached. Whenever the deadline occurs, the last version of the available result is delivered to the client. When scheduling an anytime algorithm, the completion of the root segment of the anytime algorithm must be guaranteed in order that a result of sufficient quality is available at this instant. The remaining time until the deadline is used to improve this result. The WCET problem of an anytime algorithm is thus reduced to finding a guaranteed upper bound for the WCET of the root segment. A loose upper bound of the WCET is of no serious concern, since the slack time between BCET and WCET is used to improve the result.

Most iterative algorithms are anytime algorithms. Anytime algorithms are used in pattern recognition, planning, and control.

An anytime algorithm should have the following properties:

1. **Measurable quality:** It must be possible to measure the quality of a result.
2. **Monotonic:** The quality of the result must be a non-decreasing function of time and should improve with every iteration.
3. **Diminishing returns:** The improvement of the result should get smaller as the number of iterations increases.
4. **Interruptability:** After the completion of the root segment, the algorithm can be interrupted at any time and deliver a reasonable result.

State of Practice

The previous discussion shows that the analytic calculation of a reasonable upper WCET bound of an S-task which does not make use of operating system services is possible under restricting assumptions. There are a number of tools that support such an analysis [Wil08]. It requires an annotated source program that contains programmer-supplied application-specific information to ensure that the program terminates and a detailed model of the behavior of the hardware to achieve a reasonable upper WCET bound.

Bounds for the WCET of all time-critical tasks are needed in almost all hard realtime applications. This important problem is solved in practice by combining a

number of diverse techniques:

1. Use of a restricted architecture that reduces the interactions among the tasks and facilitates the a priori analysis of the control structure. The number of explicit synchronization actions that require context switches and operating system services is minimized.
2. The design of WECT models and the analytic analysis of sub-problems (e.g., the maximum execution time analysis of the source program) such that an effective set of test cases biased towards the worst-case execution time can be generated automatically.
3. The controlled measurement of sub-systems (tasks, operating system service times) to gather experimental WCET data for the calibration of the WCET models.
4. The implementation of an anytime algorithm, where only a bound for WCET of the root segment must be provided.
5. The extensive testing of the complete implementation to validate the assumptions and to measure the safety margin between the assumed WCET and the actual measured execution times.

The state of current practice is not satisfactory, because in many cases the minimal and maximum execution times that are observed during testing are taken for the BCET and WCET. Such an observed upper bound cannot be considered a guaranteed upper bound. It is to be hoped that in the future the WCET problem will get easier, provided simple processors with private scratchpad memory will form the components of multi-processor systems-on-chips (MPSoCs).

Static Scheduling

In static or pre-runtime scheduling, a feasible schedule of a set of tasks is calculated off-line. The schedule must guarantee all deadlines, considering the resource, precedence, and synchronization requirements of all tasks. The construction of such a schedule can be considered as a constructive sufficient schedulability test. The precedence relations between the tasks executing in the different nodes can be depicted in the form of a precedence graph (Fig. 10.5).

Static Scheduling Viewed as a Search

Static scheduling is based on strong regularity assumptions about the points in time when future service requests will be honored. Although the occurrence of external events that demand service is not under the control of the computer system, the recurring points in time when these events will be serviced can be established a priori by selecting an appropriate sampling rate for each class of events. During system

design, it must be ascertained that the sum of the maximum delay times until a request is recognized by the system plus the maximum transaction response time
is smaller than the specified service deadline.

The Role of Time. A static schedule is a periodic time-triggered schedule. The timeline is partitioned into a sequence of basic granules, the basic cycle time. There is only one interrupt in the system: the periodic clock interrupt denoting the start of a new basic granule. In a distributed system, this clock interrupt must be globally

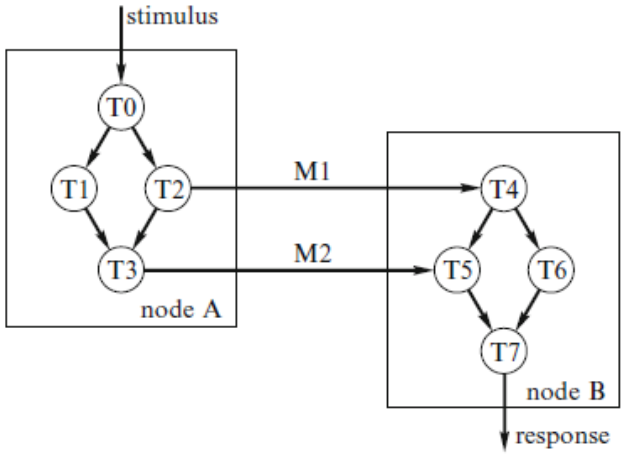


Fig. Example of a precedence graph of a distributed task set

synchronized to a precision that is much better than the duration of a basic granule. Every transaction is periodic, its period being a multiple of the basic granule. The least common multiple of all transaction periods is the schedule period. At compile time, the scheduling decision for every point of the schedule period must be determined and stored in a dispatcher table for the operating system for the full schedule period. At run time, the preplanned decision is executed by the dispatcher after every clock interrupt.

Example: If the periods of all tasks are harmonic, e.g., either a positive or negative power of two of the full second, the schedule period is equal to the period of the task with the longest period.
Static scheduling can be applied to a single processor, to a multiple-processor, or to a distributed system. In addition to preplanning the resource usage in all nodes, the access to the communication medium must also be preplanned in distributed systems. It is known that finding an optimal schedule in a distributed

system is in almost all realistic scenarios an NP-complete problem, i.e., computationally intractable. But even a non-optimal solution is sufficient if it meets all deadlines.

The Search Tree. The solution to the scheduling problem can be seen as finding a path, a feasible schedule, in a search tree by applying a search strategy. Every level of the search tree corresponds to one unit of time. The depth of the search tree corresponds to the period of the schedule. The search starts with an empty schedule at the root node of this tree. The outward edges of a node point to the possible alternatives that exist at this point of the search. A path from the root node to a particular node at level n records the sequence of scheduling decisions that have been made up to time-point n. Each path to a leaf node describes a complete schedule. It is the goal of the search to find a complete schedule that observes all precedence and mutual exclusion constraints, and which completes before the deadline. From Fig. 10.6, it can be seen that the right branch of the search tree will lead to a shorter overall execution time than the left branches.

A Heuristic Function Guiding the Search. To improve the efficiency of the search, it is necessary to guide the search by some heuristic function. Such a heuristic function can be composed of two terms, the actual cost of the path encountered until the present node in the search tree, i.e., the present point in the schedule, and the estimated cost until a goal node. Fohler [Foh94] proposes a heuristic function that estimates the time needed to complete the precedence graph, called TUR (time until

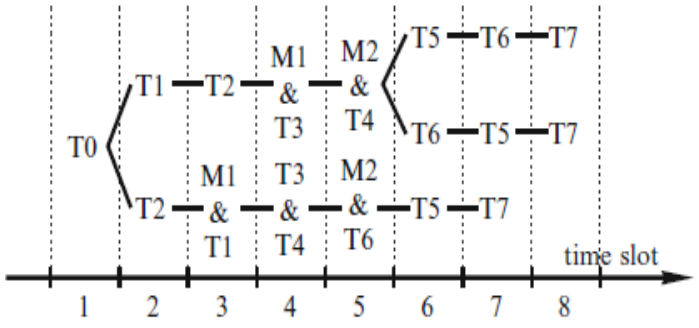


Fig. A search tree

response).

A lower bound of the TUR can be derived by summing up the maximum execution times of all tasks and message exchanges between the current task and the last task in the precedence graph, assuming true parallelism constrained by the competition for CPU resources of tasks that reside at the same node. If this

necessary TUR is not short enough to complete the precedence graph on time, all the branches from the current node can be pruned and the search must backtrack.

Increasing the Flexibility in Static Schedules

One of the weaknesses of static scheduling is the assumption of strictly periodic tasks. Although the majority of tasks in hard real-time applications is periodic, there are also sporadic service requests that have hard deadline requirements. An example of such a request is an emergency stop of a machine. Hopefully it will never be requested – the mean time between emergency stops can be very long.However, if an emergency stop is requested, it must be serviced within a small specified time interval.

The following three methods increase the flexibility of static scheduling:

1. The transformation of sporadic requests into periodic requests
2. The introduction of a sporadic server task
3. The execution of mode changes

Transformation of a Sporadic Request to a Periodic Request. While the future request times of a periodic task are known a priori, only the minimum inter-arrival time of a sporadic task is known in advance. The actual points in time when a sporadic task must be serviced are not known ahead of the request event. This limited information makes it difficult to schedule a sporadic request before run time. The most demanding sporadic requests are those that have a short response

time, i.e., the corresponding service task has a low latency. It is possible to find solutions to the scheduling problem if an independent sporadic task has a laxity l. One such solution, proposed by Mok [Mok93, p. 44], is the replacement of a sporadic task T by a pseudo-periodic task T0 as seen in Table.

This transformation guarantees that the sporadic task will always meet its deadline if the pseudo-periodic task can be scheduled. The pseudo-periodic task can be scheduled statically. A sporadic task with a short latency will continuously demand a substantial fraction of the processing resources to guarantee its deadline, although it might request service very infrequently.

Sporadic Server Task. To reduce the large resource requirements of a pseudo-periodic task with a long inter-arrival time (period) but a short latency,

Table Parameters of the pseudo-periodic task

Parameter	Sporadic task	Pseudo-periodic task
Computation time, c	c	$c' = c$
Deadline interval, d	d	$d' = c$
Period, p	p	$p' = \text{Min}(l - 1, p)$

Sprunt et al. have proposed the introduction of a periodic server task for the service of sporadic requests. Whenever a sporadic request arrives during the period of the server task, it will be serviced with the high priority of the server task. The service of a sporadic request exhausts the execution time of the server. The execution time will be replenished after the period of the server. Thus, the server task preserves its execution time until it is needed by a sporadic request. The sporadic server task is scheduled dynamically in response to the sporadic request event.

Mode Changes. During the operation of most real-time applications, a number of different operating modes can be distinguished. Consider the example of a flight control system in an airplane. When a plane is taxiing on the ground, a different set of services is required than when the plane is flying. Better resource utilization can be realized if only those tasks that are needed in a particular operating mode must be scheduled. If the system leaves one operating mode and enters another, a corresponding change of schedules must take place. During system design, one must identify all possible operating and emergency modes. For each mode, a static schedule that will meet all deadlines is calculated off-line. Mode changes are analyzed and the appropriate mode change schedules are developed. Whenever a mode change is requested at run time, the applicable mode change schedule will be activated immediately. We conclude this section with a comment by Xu and Parnas: For satisfying timing constraints in hard real-time systems, predictability of the systems behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system.

Dynamic Scheduling

After the occurrence of a significant event, a dynamic scheduling algorithm determines on-line which task out of the ready task set must be serviced next. The algorithms differ in the assumptions about the complexity of the task model and the future task behavior.

Scheduling Independent Tasks

The classic algorithm for scheduling a set of periodic independent hard real-time tasks in a system with a single CPU, the rate monotonic algorithm, was published in 1973 by Liu.

Rate Monotonic Algorithm. The rate monotonic algorithm is a dynamic preemptive algorithm based on static task priorities. It makes the following assumptions about the task set:

1. The requests for all tasks of the task set {Ti} for which hard deadlines must be satisfied, are periodic.
2. All tasks are independent of each other. There exist no precedence constraints or mutual exclusion constraints between any pair of tasks.
3. The deadline interval of every task Ti is equal to its period pi.
4. The required maximum computation time of each task ci is known a priori and is constant.
5. The time required for context switching can be ignored.
6. The sum of the utilization factors m of the n tasks with period p is given by

$$\mu = \sum c_i/p_i \leqslant n(2^{1/n} - 1)$$

can approach the theoretical maximum of unity in a single processor system. In recent years, the rate monotonic theory has been extended to handle a set of tasks where the deadline interval can be different from the period [But04].

Earliest-Deadline-First (EDF) Algorithm. This algorithm is an optimal dynamic preemptive algorithm in single processor systems which are based on dynamic priorities. The assumptions (1) to (5) of the rate monotonic algorithm must hold. The processor utilization m can go up to 1, even when the task periods are not multiples of the smallest period. After any significant event, the task with the earliest deadline is assigned the highest dynamic priority. The dispatcher operates in the same way as the dispatcher for the rate monotonic algorithm.

Least-Laxity (LL) Algorithm. In single processor systems, the least laxity algorithm is another optimal algorithm. It makes the same assumptions as the EDF algorithm. At any scheduling decision instant the

task with the shortest laxity l, i. e., the difference between the deadline interval d and the computation time c

d -c = l

is assigned the highest dynamic priority.

In multiprocessor systems, neither the earliest-deadline-first nor the least-laxity algorithm is optimal, although the least-laxity algorithm can handle task scenarios, which the earliest-deadline-first algorithm cannot handle and vice-versa.

System Design

The Design Process

Design is an inherently creative activity, where both the intuitive and the rational problem solving systems of the human mind are heavily involved. There is a common core to design activities in many diverse fields: building design, product design, and computer system design are all closely related. The designer must find a solution that accommodates a variety of seemingly conflicting goals to solve an often ill-specified design problem. In the end, what differentiates a good design from a bad design is often liable to subjective judgment.

Example: Consider the design of an automobile. An automobile is a complex mass production product that is composed of a number of sophisticated subsystems (e.g., engine, transmission, chassis, etc.). Each of these subsystems itself contains hundreds of different components that must meet given constraints: functionality, efficiency, geometrical form, weight, dependability, and minimal cost. All these components must cooperate and interact smoothly, to provide the emergent transportation service and the look and feel that the customer expects from the system car. During the purpose analysis phase, the organizational goals and the economic and technical constraints of an envisioned computer solution are established. If the evaluation at the end of this phase results in a go ahead decision, a project team is formed to start the requirement analysis and the architecture design phase. There are two opposing empirical views how to proceed in these first life cycle phases when designing a large system:

1. A disciplined sequential approach, where every life-cycle phase is thoroughly completed and validated before the next one is started (grand design)
2. A rapid-prototyping approach, where the implementation of a key part of the solution is started before the requirements analysis has been completed (rapid

prototyping)

The rationale for the grand design is that a detailed and unbiased specification of the complete problem (the What?) must be available before a particular solution (the How?) is designed. The difficulty with grand design is that there are no clear stopping rules. The analysis and understanding of a large problem is never complete and there are always good arguments for asking more questions concerning the requirements before starting with the real design work. Furthermore, the world evolves while the analysis is done, changing the original scenario. The phrase paralysis by analysis has been coined to point to this danger. The rationale for the rapid prototyping approach assumes that, by investigating a particular solution at an early stage, a lot is learned about the problem space. The difficulties met during the search for a concrete solution guide the designer in asking the right questions about the requirements. The dilemma of rapid prototyping is that ad hoc implementations are developed with great expense. Since the first prototype does address limited aspects of the design problem only, it is often necessary to completely discard the first prototypes and to start all over again. Both sides have valid arguments that suggest the following compromise: In the architecture design phase, a key designer should try to get a good understanding of the architectural properties, leaving detailed issues that affect only the internals of a subsystem open. If it is not clear how to solve a particular problem, then a preliminary prototype of the most difficult part should be investigated with the
explicit intent of discarding the solution if the looked-for insight has been gained. In his recent book [Bro10], Fred Brook states that conceptual integrity of a design is the result of a single mind.
Some years ago, Peters [Pet79] argued in a paper about design that design belongs to the set of wicked problems. Wicked problems are described by the following characteristics:

1. A wicked problem cannot be stated in a definite way, abstracted from its environment. Whenever one tries to isolate a wicked problem from its surroundings, the problem loses its peculiarity. Every wicked problem is somehow unique and cannot be treated in the abstract.
2. A wicked problems cannot be specified without having a solution in mind. The distinction between specification (what?) and implementation (how?) is not as easy as is often proclaimed in academia.
3. Solutions to wicked problems have no stopping rule: for any given solution, there is always a better solution. There are always good arguments to learn more about the requirements to produce a better design.

4. Solutions to wicked problems cannot be right or wrong; they can only be better or worse.
5. There is no definite test for the solution to a wicked problem: whenever a test is successfully passed, it is still possible that the solution will fail in some other way.

The Role of Constraints

Every design is embedded in a design space that is bounded by a set of known and unknown constraints. In some sense, constraints are antonyms to requirements. It is good practice to start a design by capturing the constraints and classifying them into soft constraints, hard constraints, and limiting constraints. A soft constraint is a desired but not obligatory constraint. A hard constraint is a given mandatory constraint that must not be neglected. A limiting constraint is a constraint that limits the utility of a design.

Example: In building a house, the mandatory construction code of the area is a hard constraint, the orientation of the rooms and windows is a soft constraint, while the construction cost may be a limiting constraint. Constraints limit the design space and help the designer to avoid the exploration of design alternatives that are unrealistic in the given environment. Constraints are
thus our friends, not our adversaries. Special attention must be paid to the limiting constraints, since these constraints are instrumental for determining the value of a design for the client. It is good practice to precisely monitor the limiting constraints as a design proceeds.

Example: In the European research initiative ARTEMIS that intends to develop a cross domain architecture for embedded systems, the first step was the capture and documentation of the requirements and constraints that such an architecture must satisfy. These constraints are published in [Art06].

System Design Versus Software Design

In the early days of computer-application design, the focus of design was on the functional aspects of software, with little regard for the nonfunctional properties of the computations that are generated by the software, such as timing, energy efficiency, or fault tolerance. This focus has led to software design methods – still prevalent today – that concentrate on the data transformation aspects of a program with little regard for the temporal or energy dimension.

Example: A critical constraint in the design of a smart phone is the expected life of a battery load. This non-functional constraint is overlooked if the focus during the design is only on the functional properties of the design. Software per se is a plan describing the operations of a real or virtual machine. A plan by itself (without a machine) does not have any temporal dimension, cannot have state (which depends on a precise notion of real-time – see Sect. 4.2.1) and has
no behavior. Only the combination of software and the targeted machine, the platform, produces behavior. This is one of the reasons why we consider the component and not the job (see Sect. 4.2.2) as the primitive construct at the level of architecture design of an embedded system. The complete functional and temporal specification of the behavior of a job (i.e., the software for a machine) is much more complicated than the specification of the behavior of a component. In addition to the four message interfaces of a component described in Sect. 4.4, the complete specification of a job must include the functional and temporal properties of the API (application programming interface) of the job to the targeted virtual or real machine [Szy99]. If the underlying machine is virtual, e.g., an execution environment that is built bottom-up by some
other software, e.g., a hypervisor, the temporal properties of this virtual machine depend on the software design of the hypervisor and the hardware performance of the physical machine. But even without a hypervisor, the temporal hardware performance of many of today's sophisticated sequential processors with multiple levels of caching and speculative execution is difficult to specify.
Considering the implications of Pollack's rule (see Sect. 8.3.2), we conjecture that in the domain of embedded real-time systems predictable sequential processors combined with the appropriate system and application software will form the IP-cores, the components, of the envisioned multiprocessor systems-on-chips (MPSoC) of the embedded system of the future.

The intended behavior of a component can be realized by different implementation technologies:
1. By designing software for a programmable computer, resulting in a flexible component consisting of a local operating system with middleware and application software modules.
2. By developing software for a field-programmable gate array (FPGA) that implements the component's functionality by the proper interconnection of a set of highly concurrent logic elements.
3. By developing an application specific integrated circuit (ASIC) that implements the functionality of the component directly in hardware.

Viewed from the outside, the services of a component must be agnostic of the chosen implementation technology. Only then it is possible to change the implementation of a component without any effects at the system level. However, from the point of view of some of the non-functional component characteristics such as energy consumption, silicon real-estate requirements, flexibility to change, or non-recurring development costs, different component implementations have vastly different characteristics. In a number of applications it is desired to develop at first a hardware-agnostic model of the services of a component at the architecture level and to postpone the detailed decisions about the final implementation technology of the component to a later stage.

Example: In a product for mass-market consumer appliance, it makes sense to first develop a prototype of a component in software-on-a CPU and to decide later, after the market acceptance of the product has been established, to shift the implementation to an FPGA or ASIC.

Design Phases

Design is a creative holistic human activity that cannot be reduced to following a set of rules out of a design rule-book. Design is an art, supplemented by scientific principles. It is therefore in vain to try to establish a complete set of design rules and to develop a fully automated design environment. Design tools can assist a designer in handling and representing the design information and can help in the analysis of design problems. They can, however, never replace a creative designer. In theory, the design process should be structured into a set of distinct phases:

purpose analysis, requirements capture, architecture design, detailed component design and implementation, component validation, component integration, system validation, and finally system commissioning. In practice, such a strict sequential decomposition of the design process is hardly possible, since the full scope of a new design problem is not comprehended until the design process is well under its way, requiring frequent iterations among the design phases. The focus of this chapter is on the architecture design phases, while the validation phases are covered.

Purpose Analysis

Every rational design is driven by a given purpose. The purpose puts the design into the wider context of user expectations and economic justification and thus precedes the requirements. Purpose analysis, i.e., the analysis why a new system is needed and what is the ultimate goal of a design must precede the

requirements analysis, which already limits the scope of analysis and directs the design effort to a specific direction. Critical purpose analysis is needed in order to put the requirements into the proper perspective.

Example: The purpose of acquiring a car is to provide a transportation service. There are other means of transportation, e.g., public transport, which should be considered in the purpose analysis phase. In every project, there is an ongoing conflict between what is desired and what can be done within the given technical and economic constraints. A good understanding and documentation of these technical and economic constraints reduces the design space and helps to avoid exploring unrealistic design alternatives.

Requirements Capture

The focus of the requirements phase is to get a good understanding and a concise documentation of the requirements and constraints of the essential system functions that provide the economic justification of the project. There is always the temptation to get sidetracked by irrelevant details about representational issues that obscure the picture of the whole. Many people find it is easier to work on a well specified detailed side problem than to keep focus on the critical system issues. It requires an experienced designer to decide between a side problem and a critical system issue.

Every requirement must be accompanied by an acceptance criterion that allows to measure, at the end of the project, whether the requirement has been met. If it is not possible to define a distinct acceptance test for a requirement, then the requirement cannot be very important: it can never be decided whether the implementation is meeting this requirement or not. A critical designer will always be suspicious of postulated requirements that cannot be substantiated by a rational chain of arguments that, at the end, leads to a measurable contribution of the stated requirement to the purpose of the system.

In the domain of embedded systems, a number of representation standards and tools have been developed to support the system engineer. Standards for the uniform representation of requirements are of particular importance, since they simplify the communication among designers and users. The recent extension of the UML (Unified Modeling Language) with the MARTE (Modeling and Analysis of Real-Time Embedded Systems) profile provides a widely accepted standard for the representation of real-time requirements [OMG08].

Architecture Design

After the essential requirements have been captured and documented, the most crucial phase of the life cycle, the design of the system architecture, follows. Complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not [Sim81]. Stable intermediate forms are encapsulated by small and stable interfaces that restrict the interactions among the subsystems. In the context of distributed real-time systems, the architecture design establishes the decomposition of the overall systems into clusters and components, the linking interfaces of the components, and the message communication among the components.

In general, introducing structure restricts the design space and may have a negative impact on the performance of a system. The more rigid and stable the structure, the more notable the observed reduction in performance will be. The key issue is to find the most appropriate structure where the performance penalties are outweighed by the other desirable properties of the structure, such as composability, understandability, energy efficiency, maintainability, and the ease of implementing fault-tolerance or of adding new functionality in an evolving system and environment.

Design of Components

At the end of the architectural design phase, the requirements have been allocated to components, and the linking interfaces (LIFs) of the components are precisely specified in the value domain and in the temporal domain. The design effort can now be broken down into a set of concurrent design activities, each one focusing on the design, implementation, and testing of a single component. The detailed design of the local interface (see Sect. 4.4.5) between a component and its local environment (e.g., the controlled object or other clusters) is not covered in the architectural design phase, since only the semantics, but not the syntax of these local interfaces are needed for the cluster-LIF specification of the components. It is up to the detailed design of a component to specify and implement these local interfaces. In some cases, such as the design of the concrete man-machine interface for the operator, this can be a major activity.

The detailed steps that have to be taken in order to implement the design of a component depend on the chosen implementation technology. If the services of a component are implemented by a software-on-a-CPU design, then the necessary design steps will differ radically from a design that targets an ASIC as its final outcome. Since the focus of this book is on the topic of architecture design of embedded systems, we do not cover the detailed component implementation techniques for the different implementation technologies at any length.

UNIT-IV

CAN

INTRODUCTION TO CAN

The Controller Area Network, commonly known as CAN,

A controller area network (CAN) is a serial bus network of microcontrollers that connects devices, sensors and actuators in a system or sub-system for real-time control applications. There is no addressing scheme used in controller area networks, as in the sense of conventional addressing in networks (such as Ethernet). Rather, messages are broadcast to all the nodes in the network using an identifier unique to the network.

It was originally designed for use in automobiles. By virtue of its massive adoption by automakers worldwide, low-cost microcontrollers with CAN controller interfaces are available from over twenty manufacturers, making CAN a mainstream network technology. Moreover, CAN has migrated into many non-automobile applications over the last ten years creating a requirement for an open, standardized higher-layer protocol that provides a reliable message exchange system along with a means to detect, configure and operate nodes. Several higher-layer CAN protocols emerged such as SAE J1939, Device Net and CAN-open. While each protocol has its own special purpose, CAN open is the most popular higher-layer protocol for embedded networking applications – those networks that are completely hidden within a machine or cell – and is found in over twenty vertical markets such as transportation, medical, industrial machinery, building automation and military.

HISTORY OF CAN:

In February of 1986, Robert Bosch introduced the CAN (Controller Area Network) serial bus system at the SAE congress in Detroit. In mid-1987, Intel delivered the first stand-alone CAN controller chip, the 82526. Shortly thereafter, Philips Semiconductors introduced the 82C200. Today, almost every new passenger car manufactured in Europe is equipped with at least one CAN network. Also used in other types of vehicles, from trains to ships, as well as in industrial controls, CAN is one of the most dominating bus protocols. To date, chip manufacturers have produced and sold more than 500 million CAN devices in total.

- **Pre CAN:** Car ECUs relied on complex point-to-point wiring
- **1986:** Bosch developed the CAN protocol as a solution
- **1991:** Bosch published CAN 2.0 (CAN 2.0A: 11 bit, 2.0B: 29 bit)
- **1993:** CAN is adopted as international standard (ISO 11898)

- **2003:** ISO 11898 becomes a standard series (11898-1, 11898-2, ...)
- **2012:** Bosch released the CAN FD 1.0 (flexible data rate)
- **2015:** The CAN FD protocol is standardized (ISO 11898-1)
- **2016:** The physical CAN layer for data-rates up to 5 M bit/s standardized in ISO 11898-2
- Today, the CAN protocol is standard in practically all vehicles (cars, trucks, buses, tractors, ...) - as well as ships, planes, EV batteries, industrial machinery and more.

  Further, more exotic cases include drones, radar systems, submarines or even prosthetic limbs.

APPLICATIONS:

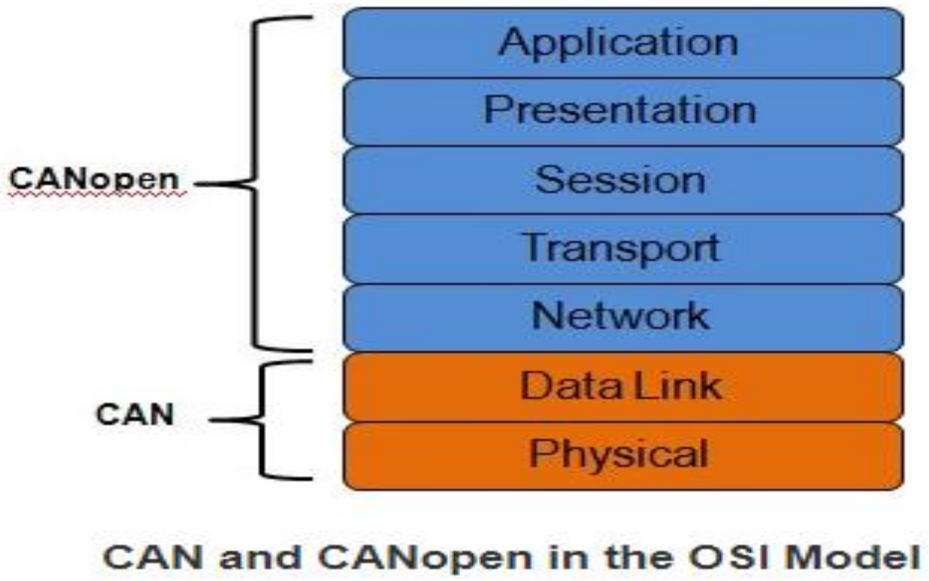
- Passenger vehicles, trucks, buses (gasoline vehicles and electric vehicles)
- Electronic equipment for aviation and navigation
- Industrial automation and mechanical control
- Elevators, escalators
- Building automation
- Medical instruments and equipment

CAN open Introduction:

CAN open is a high-level communication protocol and device profile specification that is based on the CAN (Controller Area Network)protocol. The protocol was developed for embedded networking applications, such as in-vehicle networks. The CAN open umbrella covers a network programming framework, device descriptions, interface definitions and application profiles. CAN open provides a protocol which standardizes communication between devices and applications from different manufacturers. It has been used in a wide range of industries, with highlights in automation and motion applications.

In terms of the OSI communication systems model, CAN covers the first two levels: the physical layer and the data link layer. The physical layer defines the lines used, voltages, high-speed nature, etc. The data link layer includes the fact that CAN is a frame-based (messages) protocol. CAN open covers the top five layers: network (addressing, routing), transport (end-to-end reliability), session (synchronization), presentation (data encoded in standard way, data representation) and application. The application layer describes how to configure, transfer and synchronize CAN open devices. The concepts of the application

layer, covered in specification CiA DS 301, are covered in this document. The intention is to give users a brief overview of the concepts of CAN open.



CAN and CANopen in the OSI Model

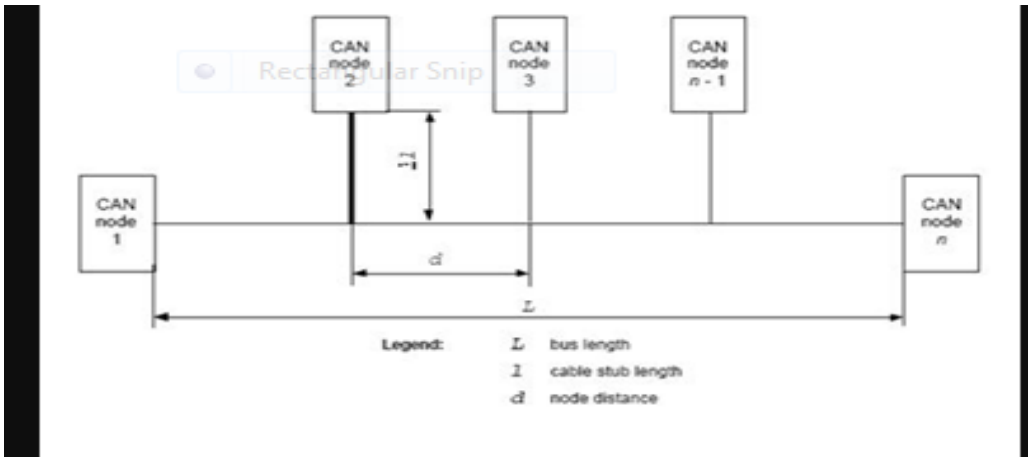
The Basics of the CAN open Application Layer

The following section explains the basic concepts related to the CAN open protocol application layer. This document is intended as a basic overview only, and users are encouraged to review the CiA DS 301 specification for more information.

CAN Physical Layer:

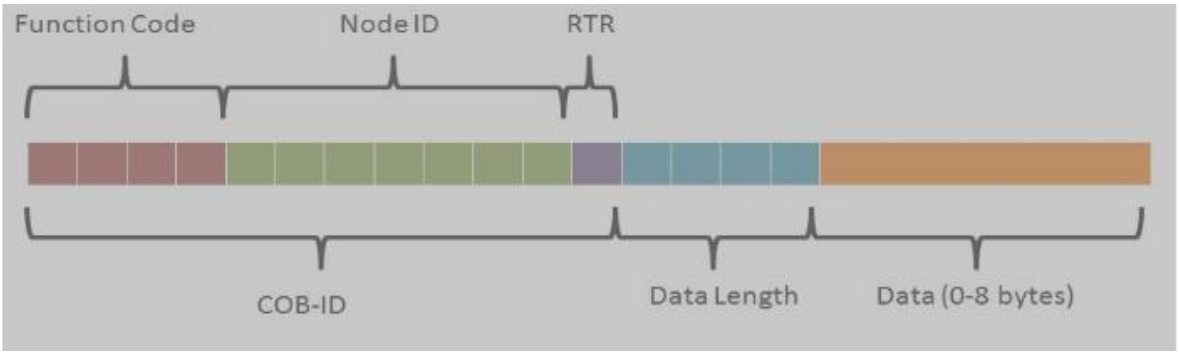
CAN Bus

It consists of two wire serial link- CAN_H and CAN_L and their voltage levels relative to each other determine whether a 1 or 0 is transmitted. This is differential signalling. The current flowing in each signal line is equal but opposite in direction, resulting in a field-cancelling effect that is a key to low noise emissions. This ensures a balanced differential signalling which reduces noise coupling and allows high rate of transmission over the wires. Usually the wires are twisted pair cables with bus length of 40 m and maximum of 30 nodes. It is a shielded or unshielded cable with a characteristic impedance of 120 Ohms.



CAN open Message Format:

The message format for a CAN open frame is based on the CAN frame format. In the CAN protocol, the data is transferred in frames consisting of an 11-bit or 29-bit CAN-ID, control bits such as the remote transfer bit (RTR), start bit and 4-bit data length field, and 0 to 8 bytes of data. The COB-ID, commonly referred to in CAN open, consists of the CAN-ID and the control bits. In CAN open, the 11-bit CAN ID is split into two parts: a 4-bit function code and a 7-bit CAN open node ID. The 7-bit size limitation restricts the amount of devices on a CAN open network to 127 nodes.



CAN open Frame Format (bits shown except for data field)

All COB-ID's must be unique to prevent conflicts on the bus. In SDO communication, there should always be only one node that needs to access the individual object dictionary indices of the slave nodes.

Index	Data Type
1	BOOLEAN
2	INTEGER8
3	INTEGER16
4	INTEGER32
5	UNSIGNED8
6	UNSIGNED16
7	UNSIGNED32

The Object Dictionary Concept:

The core of any CAN open node is the Object Dictionary (OD), a lookup table with a16-bit Index and an 8-bit Sub index. This allows for up to 256 Subentries at each Index. Objective In the previous sections we have examined general requirements for embedded networks. In this section we would like to introduce CAN open and point out how many of the embedded networking requirements are met by CAN open. This section is an introduction to the primary functionality provided by CAN open and is intended to give students a quick start into the main ideas of CAN open. The chapters following this section will repeat some of the basic information presented here and add technical details that are missing in this first overview of CAN open functionality. Each entry can hold one variable of any type (including a complex structure) and length. In the following sections the terms Index, Sub index and Sub entry will be used when describing such Object Dictionary entries. All process and communication related information/data is stored as entries in predefined locations of the Object Dictionary. Unused entries do not need to be implemented.

The Object Dictionary not only provides a way to associate variables with an Index and Sub index value, it also specifies a data type definition table. The entries starting at Index 1 are exclusively used to specify data types. Table 1.1 shows the first seven entries in the Object Dictionary defining some commonly used data types. The complete listing of pre-defined data types is given in Chapter 2, Section 2.2.3. In addition, CAN open also supports application specific data types that can be added list of supported data types.

Object Dictionary Entries Starting at Index 1 Define Data Types

It should be noted that the entries mentioned above are only used to define data types, not to store any variables. The Object Dictionary entries beyond 1000h are used for variable storage; if an entry is specified to be of type “UNSIGNED16” then an alternate description of the data type (for example, used in electronically readable specifications) is used to indicate it is of data type 6.As specified, the Object Dictionary satisfies the basic networking requirement of being able to define data types and place variables into the network nodes. If a specification says that a node must have a variable called "X-Position" which is located at Index 2000h, Sub index 0 and its data type is 4, then according to the Object Dictionary the data type is INTEGER32, an integer value of 32 bits.

Index	SubIdx	Type	Description
1000h	0	UNSIGNED32	Device Type Information
1001h	0	UNSIGNED8	Error Register
1017h	0	UNSIGNED16	Heartbeat Time
1018h			Identity Object
	0	UNSIGNED8	= 4 (Number of sub-index entries)
	1	UNSIGNED32	Vendor ID
	2	UNSIGNED32	Product Code
	3	UNSIGNED32	Revision Number
	4	UNSIGNED32	Serial Number

Figure lists the mandatory Object Dictionary entries that every CAN open node must implement to be CAN open compliant. Primarily, these provide the device type information that gives an indication of which device profile a device belongs to (if any), an error register, and an identifier record. The heartbeat is a low-priority status message sent by a node on a periodic basis. The heartbeat time is listed here

because every node must support either the heartbeat or node guarding mechanism; today heartbeat is the recommended, preferred method.

Electronic Data Sheets:

Electronic Data Sheets (EDS) offer a standardized way of specifying supported Object Dictionary entries. Any manufacturer of a CAN open module delivers such a file with the module, which in layout is similar to the ".ini" files used with Microsoft Windows operating systems. (Note: a future standard for EDS files based on XML is currently in development.)

An example of an Object Dictionary entry in an EDS file is:

[1000]

Parameter Name=Device Type

Object Type=0x07

Data Type=0x0007

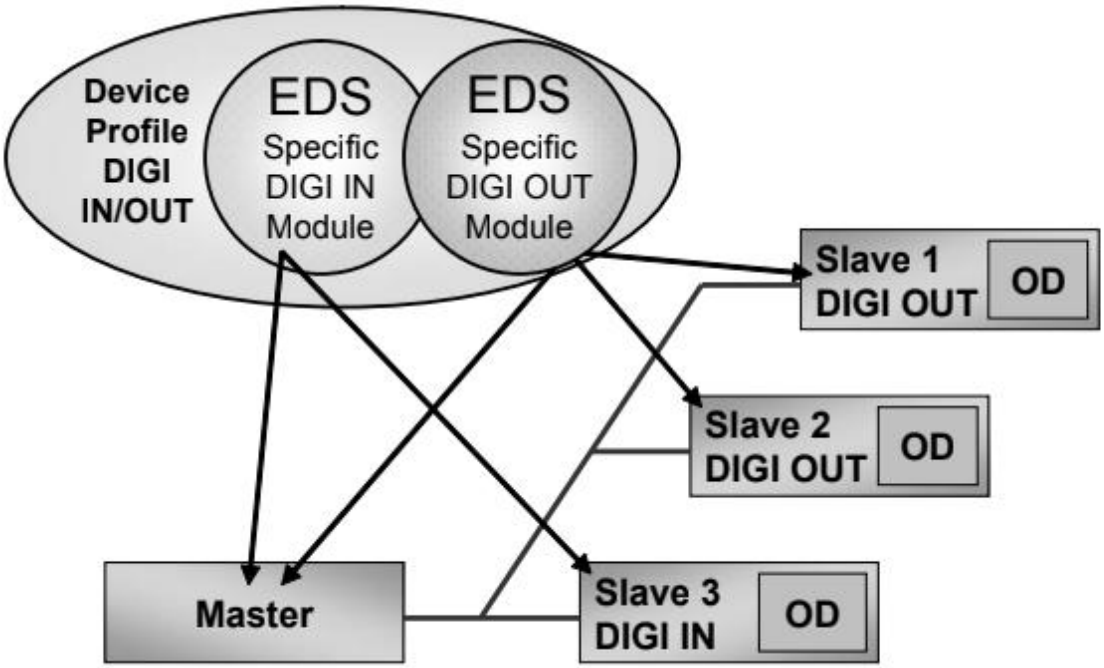
Access Type=r0

Default Value=0x00030191

PDO Mapping=0

The example above shows the EDS definition of the Object Dictionary entry

[1000h,00h]. The data type is 7 (UNSIGNED32, see Table 1.1).



Electronic Data Sheets (EDS) Specify the Contents of Object Dictionaries

A CAN open master or configuration tool running on a PC with a CAN card can directly load the EDS into its set of recognized devices. Once a device is found on the network, the master or configuration tool will try to find the matching EDS. Once found, all supported Object Dictionary entries are known by the master/configuration tool.

Figure shows the relationship between Device Profiles and Electronic Data Sheets .The Device Profile specifies the minimum entries that need to be supported by a device conforming to the profile. However, the EDS might only specify objects that are specific to a certain manufacturer or sub-type of module. Device Profiles and Electronic Data Sheets are the basic functionality needed to meet the requirement for "off-the-shelf" availability of network devices. From the communication point of view, any two nodes that conform to the same EDS are interchangeable their Object Dictionaries are identical and they have the same communication behaviour.

DEVICE PROFILES:

Although the Object Dictionary concept allows for structuring the data that needs to be communicated, there is still something missing: Which entry in the dictionary is used for what? The dictionary is far too

big to allow the master to take “wild guesses” and simply try to access certain areas of the dictionary to see if they are supported.

The solution is simple. First of all, there are a few mandatory entries that all CAN open nodes must support. These include the identity object with which a node can identify itself, and an error object to report a potential error state. In addition, optional entries are specified by the CAN open specification. The Device profiles are add-on specifications that describe all the communication parameters and Object Dictionary entries that are supported by a certain type of CAN open module. Such profiles are available for generic I/O modules, encoders and other devices. A master or configuration tool can read-access the identity object of any slave node using a Service Data Object or SDO (a messaging protocol – more about this shortly). As a reply, it receives an SDO with the information about which device profile a module conforms to. Assuming the master knows which object entries are defined for a particular device profile, it now knows which Object Dictionary entries are supported and can access them directly. There may be instances where an application requires the implementation of non standardized, manufacturer-specific Object Dictionary entries. This is not a problem, because CAN open is truly "open." Additional entries that disable or enable a certain functionality that is not covered by one of the existing device profiles can be implemented in any device, as long as they conform to the structural layout of the Object Dictionary.

UNIT- V

CAN STANDARDS

Object Dictionary

The Object Dictionary (called “OD” for short) is like a table that holds all network-accessible data, and each CANopen node must implement its own Object Dictionary.

The Object Dictionary contains a description of the CANopen configuration and functionality of the node it is stored in, and may be read and written to by other CANopen nodes. In addition, the Object Dictionary is used for storing application specific information that is used by the node in which it is stored. This information can also be used by other nodes on the network.

By writing data to the entries in the Object Dictionary of a node (and sometimes by reading from them), the node can be instructed to perform an operation of some kind, for example sampling current temperature or G-forces and making the sampled data present in the Object Dictionary to be read by others.

By reading the entries in the Object Dictionary of a node, other nodes may find out some information about what the node does and how it operates. Whether complete descriptive information or only minimal information about the node is present in the Object Dictionary can vary from application to application depending on the requirements of the network design. However, some information in the Object Dictionary is mandatory and must be present. Which information is mandatory often depends on which CANopen features are implemented by the node.

Object Dictionary Organization and Contents

The Object Dictionary is organized as a collection of entries, rather like a table. Each entry has a number called an Index, which is used to access the entry. The Index is 16 bits in size giving a maximum of 65,536 entries. Each entry in the Object Dictionary may have up to 256 Subentries, referenced using an 8-bit value called the Subindex. Each entry has at least one Subentry.

Not all entries in the Object Dictionary are implemented or used, creating gaps in the table. For example, the entries with Indexes 0000h - 09FFh are often not implemented, but the entry with Index 1000h is always implemented.

It is common practice to use hexadecimal (base 16) when referring to Object Dictionary Indexes and Subindexes. The CANopen specifications use hexadecimal notation for these values.For entries that store

only one value, there is only one Subentry at Subindex 00h. Entries that store more than one value must have a Subentry for each value, and store the number of the highest Subentry at Subindex 00h.

Data Types

The Object Dictionary may store both standard/pre-defined and manufacturer defined data types. In addition, the CANopen specification defines two basic classes of data types, Standard and Complex. To organize the range of Indexes used for defining the data types, the data types section of the Object Dictionary is further divided into the sections

Index Range	Description
0001h – 001Fh	Standard Data Types
0020h – 0023h	Pre-defined Complex Data Types
0024h – 003Fh	Reserved
0040h – 005Fh	Manufacturer Complex Data Types
0060h – 007Fh	Device Profile Standard Data Types
0080h – 009Fh	Device Profile Complex Data Types
00A0h – 025Fh	Multiple Device Modules Data Types
0260h – 0FFFh	Reserved

Table Data Type Storage in the Object Dictionary

Standard Data Types

Table lists the Standard Data Types, their descriptions and the Object Dictionary locations where they are defined,

Standard Data Type	Description	Stored in OD Index
BOOLEAN	Single bit value 0 or 1 indicating false or true	0001h
INTEGER8	8-bit signed integer	0002h
INTEGER16	16-bit signed integer	0003h
INTEGER24	24-bit signed integer	0010h
INTEGER32	32-bit signed integer	0004h
INTEGER40	40-bit signed integer	0012h
INTEGER48	48-bit signed integer	0013h
INTEGER56	56-bit signed integer	0014h
INTEGER64	64-bit signed integer	0015h
UNSIGNED8	8-bit unsigned integer	0005h
UNSIGNED16	16-bit unsigned integer	0006h
UNSIGNED24	24-bit unsigned integer	0016h
UNSIGNED32	32-bit unsigned integer	0007h
UNSIGNED40	40-bit unsigned integer	0018h
UNSIGNED48	48-bit unsigned integer	0019h
UNSIGNED56	56-bit unsigned integer	001Ah
UNSIGNED64	64-bit unsigned integer	001Bh
REAL32	32-bit single precision floating point number	0008h
REAL64	64-bit double precision floating point number	0011h

Table Standard Data Types

Complex Data Types

Complex Data Types are types that contain one or more of the standard data types grouped together, allowing sets of data to be constructed. This is analogous to structures in the C programming language.

Complex Data Types are really a shorthand or simplification for describing Object Dictionary entries that use different types for each of their Subentries, and are useful when a specific collection of data types are to be used frequently.

Mandatory Entries

Device Type (1000h)

The Device Type is a 32-bit value that describes in a limited way some of the capabilities of the node. For example, it can describe if the node is a digital input/output module, and if so, whether inputs and/or outputs are implemented.

Error Register (1001h)

The Error Register is an 8-bit value that can indicate if various generic errors have occurred in the node, for example, current error, temperature error, communication error, etc. The only bit that must be implemented is the generic error bit. There is a manufacturer specific bit available to indicate an application specific error. This byte is also transmitted in Emergency Objects.

Guard Time (100Ch)

Nodes must support either heartbeats or node guarding. Both mechanisms are discussed later in this chapter. To summarize, these mechanisms allow nodes to determine if a specific node is alive and well and able to communicate to the network, along with the node's current state. The Guard Time is a 16-bit value that specifies how frequently the node guarding request is transmitted by the master or must be received by the node. This entry must be implemented if heartbeats are not used.

Life Time Factor (100Dh)

The Life Time Factor is an 8-bit value that works with the Guard Time. It specifies how many multiples of the Guard Time must pass without transmission from the master or reception of a response from a slave before an error condition is generated. This entry must be implemented if heartbeats are not used.

Producer Heartbeat Time (1017h)

If the node is not using node guarding then it must implement heartbeats. This entry specifies how often the node should transmit heartbeat messages. It can be set to zero, however, to disable heartbeat transmission. This entry must be implemented if node guarding is not used.

Identity Object (1018h)

The Identity Object provides identifying information about the node. It must contain at a minimum the CAN In Automation assigned Vendor ID, which is unique to a particular vendor. It may also contain a product code to identify the product the node is in, a revision number and a serial number.

Device Profile Parameters

The CANopen specification [CiADS301] provides a variety of communication services. Once a specific node is implemented, the designer of the node (or the network where it will be used) has to specify which of these communication services are used and how. A Device Profile specifies the process data variables a node knows and the default configuration and communication settings. There are proprietary profiles, as well as CiA standardized Device Profiles and Application Profiles.

The Electronic Data Sheets (EDS) and Device Configuration Files (DCF)

In order to provide CANopen software tools such as monitors, analyzers and configuration tools with a way to recognize which Object Dictionary entries are available in CANopen nodes, an electronically readable file format is required.

EDS Format and Editing

The format of the EDS is specified in [CiADSP306]. It is similar to that of Microsoft Windows ".ini" files and a regular ASCII-editor could be used to read and/or modify it. However, in order to be compliant with the standard, entries must not only have the appropriate parameters but several entries must also be cross-referenced. Thus trying to edit and maintain an EDS with an ASCII-editor, although possible, is not really practical.

EDS Usage

There are several tools available that can work with EDS files. High-end CAN monitors and analyzers or CANopen configuration tools can extract symbolic information from these files and use them in their displays. A monitor or analyzer with this feature can listen to CANopen traffic on the network and associate the symbols of these files with the messages seen on the network. So if there is a process data variable defined in an EDS that is called "Boiler Temperature" and that value is transmitted over the CANopen network, these tools can directly make the symbolic link and display the text "Boiler Temperature" along with the current value transmitted.

Other tools that work with EDS files are high-end CANopen masters. Such a CANopen master is typically used in a system to receive all inputs, run some control algorithm and then transmit all outputs. A CANopen master that can read EDS files can use the symbolic names from the EDS file in the control algorithm. So in the case of the example above a variable called "Boiler Temperature" is available to be used by the control algorithm.

DCF Format and Usage

The format of the Device Configuration File is almost identical to the EDS. However, the usage is very different and justifies giving it a separate name and not just referring to it as some sort of "EDS variant."

The EDS defines the format of an Object Dictionary that may apply to multiple nodes. The idea of a DCF is to store the configuration parameters of a specific node.This can include minimum, maximum and

default values for each entry. The DCF stores a specific setting, the current value that an Object Dictionary entry has or should have. The idea is that a CANopen configuration tool or master can use the EDS to find out which entries are accessible in a node, and they can use a DCF to store (or retrieve) the values that a node has in these Object Dictionary entries. Thus it becomes possible to save and restore all settings of a node: to save current settings a tool/master would read all Object Dictionary entries of a node and store the values read in a DCF.

Accessing the CANopen Object Dictionary (OD) with Service Data Objects (SDO)

Client and Server

Each CANopen node not only implements its own Object Dictionary, it also implements a server that handles read and write requests to its Object Dictionary. So a master or configuration tool acts as a client to that server and can send read or write requests to it. As an example, a configuration tool could send a request: "Node Number 5, I need to know what you have at Index 1000h, Subindex 00h." Node 5 would recognize the request and reply with a response: "Whoever requested it, here is the data that I have in my Object Dictionary at Index 1000h, Subindex 00h."

Message Identifiers Used for SDOs

As discussed earlier, CANopen uses unique message identifiers – one message ID is only used for one purpose in an entire CANopen network (this is a requirement of the CAN arbitration feature that is explained in Section 5.2.8). There are some exceptions to this rule that are primarily used for specific configuration services during initialization, test or debugging.In order to implement a point-to-point communication channel two such message identifiers need to be reserved; one to send requests to a specific node and one for responses sent by that node. Figure 2.2 shows the message identifiers that are used by default. The message identifier that is used to send a request to a specific node is calculated by adding the Node ID of that node (1-127) to a base address of 600h. Thus addresses 601h to 67Fh are used to provide 127 channels from one client to up to 127 servers. The message identifier that is used to send a response from each node back to the client who sent the request is calculated by adding the Node ID of the node (1-127) to a base address of 580h. Thus addresses 581h to 5FFh are used to provide 127 channels from as many as 127 servers back to the client.

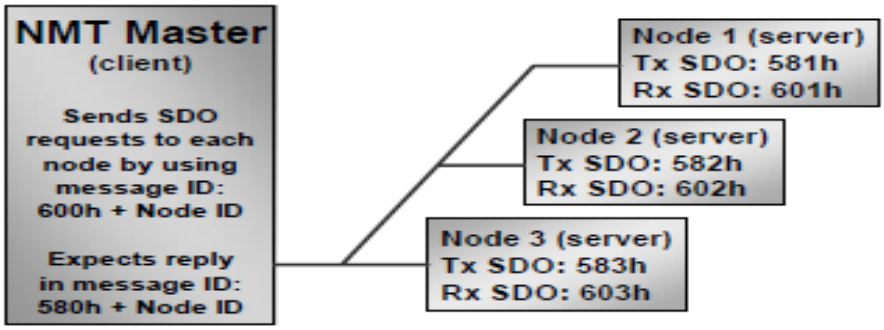


Figure Default Message IDs for SDO Communication

It should be noted that the default scheme used for assigning the message identifiers only allows one client to be on the network. Because message IDs must be unique, no two devices have the right to send SDO requests to the same node at the same time. The entire SDO communication was designed around the idea that only one node in the system needs the power to access each and every Object Dictionary entry in each and every node. This is either a configuration tool or some sort of master responsible for configuration.

SDO Message Contents

Every SDO request and response message contains 8 bytes of data of which the first byte is a so-called "specifier." The bits in it primarily specify whether this message contains a read, write or abort (error indication). Other bits are used to indicate if this is an "expedited transfer" where all data exchanged is part of this message, or a "segmented transfer" where the data does not fit into one message and multiple messages are used. The optional "block transfer" is optimized for the transfer of large data blocks and is described in Section 2.8.5.

Typically bytes 2 to 4 contain the "multiplexor" – the combination of 16-bit Index and 8-bit Subindex identifying the Object Dictionary entry that is accessed with this SDO. The byte order for the multiplexor is as follows: low byte of 16-bit Index, high byte of 16-bit Index and 8-bit Subindex.

The remaining bytes (5 to 8) are used to transmit data where applicable. If the data transferred is 4 bytes or less it is typically part of the message (expedited), otherwise it follows in additional messages (segmented).

In the case of expedited transfers the number of bytes used for data is indicated by additional bits in the specifier.

In the case of a segmented transfer the first SDO request and SDO response do not contain data, but an indication of how many bytes will need to be transferred in total. Each segment transmitted after that also contains the specifier byte and up to 7 data bytes. The specifier contains bits that specify if this is the last segment and if it is, how many of bytes in the current message are data bytes that belong to the transfer.

At any time during a transfer, any of the two communication partners may abort the communication by sending an SDO Abort message.

SDO Download vs. Upload

Per [CiADS301] an "SDO Download" implements a write access to the Object Dictionary of a node and an "SDO Upload" implements a read access.

In the authors' experience these terms are easily confused, and they are only listed here for completeness. In the following we will use the terms "SDO Read Access" and "SDO Write Access." These are easily understandable, especially as they correspond to the access type field available for all Object Dictionary entries, where possible values include read-only, read-write and write-only.

SDO Usage Limitation

The SDO Read Access and SDO Write Access as explained in this section provide a mechanism for generic read and write access to the Object Dictionary of each node on the network. Because all configuration and process data of a node is part of its Object Dictionary, the SDO transfers can be used to access the process data and it would be possible to implement a communication system entirely based on SDO communication. However, that was never the intention of the SDO communication (recall that SD stands for "Service Data") and thus this communication mode is not very efficient.

For real-world implementations a leaner, more efficient communication method is desirable in order to minimize the communication overhead and to make best usage of the available bandwidth. In CANopen, this lean and efficient communication method is provided by the Process Data Objects (PDOs).

Network Management (NMT)

NMT Slave State Diagram

Every CANopen slave node must implement an NMT state machine that allows the slave to be in different operating states. The diagram in Figure 2.8 illustrates the major states a slave node can be in. It should be noted that some of these state transitions can be made automatically (by the slave themselves) where others can only be made upon receiving the corresponding NMT Master message. The NMT Master message can be directed at either individual nodes or at all nodes simultaneously. It contains the new state that the addressed node(s) should switch to.

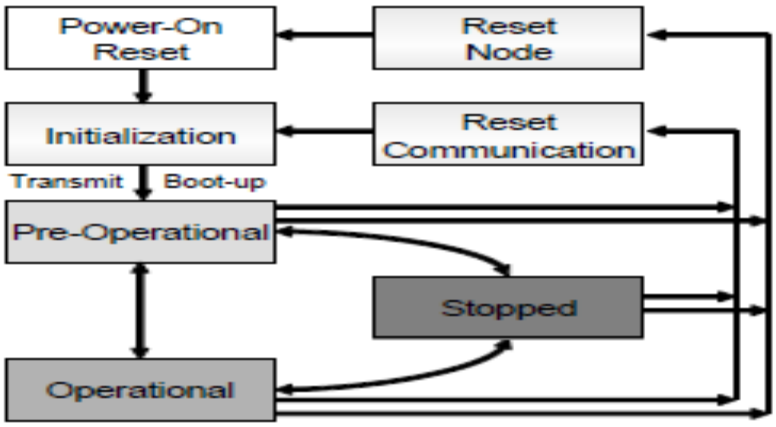


Figure The Network Management States of a Slave

Upon power-up a CANopen slave node comes out of "Power-On Reset" and goes into Initialization. It initializes the entire application and the CAN/CANopen interfaces and communication. At the end of the initialization the node tries to transmit its boot-up message. As soon as it is transmitted successfully, the node switches to the Pre-operational state.

Using the NMT Master message, an NMT Master can switch individual nodes or all nodes back and forth between the three major states: Pre-operational, Operational and Stopped. In addition, the NMT Master has the option to request two different reset actions. Upon receiving the "Reset Communication" command, a CANopen slave node will reset the CAN/CANopen communication interfaces. A "Reset Node" command, however, results in a reset of the entire node with all peripherals and all software. Both reset states result in a new boot-up message being transmitted by the node and the node reverting back to the Pre-operational state, where it will wait for further NMT message commands.

CANopen Messages Produced and Consumed

The main difference between the various NMT states is that not all types of CANopen communication are actively used in each state. Table 2.17 shows which communication a node may perform when it is in a particular NMT state.

In the Initializing state a node may only produce the boot-up message and it does not consume any messages.In the Pre-operational state a node actively participates in all communication related to SDOs, Emergencies (if used by the node), Timestamps (if used by the node) and

Heartbeat/Node Guarding.

There is only one difference between the Pre-operational state and the Operational state. The operational state adds PDO communication, allowing the node to exchange and work with process data. Only in Operational mode does a CANopen node truly run, meaning it executes all the input and output functions that it was designed to do. In the Stopped state a node literally stops all communication, except for the minimal NMT services.

Contents of CANopen Messages

Endianess

All numerical data types consisting of multiple bytes are transferred in CANopen (whether in SDO or PDO) in the “Little Endian” format. Bytes are ordered by significance and the lower significant bytes come first.

For example, a 2-byte word would be transmitted low-byte first, followed by the high-byte. A 4-byte word is transmitted with the least significant byte first, followed by the bytes of next higher significance and the most significant byte transmitted last.

SDO Communication

When using SDO communication, one needs to differentiate between two major communication modes, typically referred to as “expedited transfer” and “segmented transfer.” A third, optional mode is the "block transfer", an optimized method to transfer large data amounts. Section 2.8.5 explains the messages used for block transfer. An expedited transfer consists of one SDO request and one SDO response. A

segmented transfer consists of an SDO initiation request and response and then one pair of request and response for each 7-byte segment.

With expedited transfer up to four bytes of data can be directly embedded in an SDO request or response, suitable for accesses to Object Dictionary entries that are up to 4-bytes long. The segmented transfer allows for transmission of data bigger than 4-bytes and is required to access Object Dictionary entries that are longer than 4-bytes.

The Initiate SDO Download – Request

The client (typically the node trying to configure a CANopen slave) sends this request to a SDO server (implemented within a CANopen slave) by using the CAN identifier 600h plus the Node ID of the CANopen slave addressed. The download request is a request to write to a specific Object Dictionary entry.

The Download SDO Segment – Request

If in the initiation sequence a segmented transfer was negotiated, this message is used to transmit the next segment (of up to 7 bytes) from client to SDO server.

The Download SDO Segment – Response

This is the response sent back from the SDO server to the client indicating that the previously received download (write) segment request was processed successfully.

The Initiate SDO Upload – Request

The client (typically the node trying to configure a CANopen slave) sends this request to an SDO server (implemented within a CANopen slave) by using the CAN identifier 600h plus the Node ID of the CANopen slave addressed. The upload request is a request to read from a specific Object Dictionary entry.

The Initiate SDO Upload – Response

This is the response sent back from the SDO server to the client indicating that the previously received upload (read) request can be processed. If expedited transfer is used, the data read from the Object

Dictionary is part of the response, otherwise additional segmented transfers are used. The default CAN identifier used for this message is 580h plus the Node ID of the node implementing the SDO server.

The Upload SDO Segment – Request

If in the initiation sequence a segmented transfer was negotiated, this message is used to request that the next segment (of up to 7 bytes) be transmitted from SDO server to client.

The Upload SDO Segment – Response

This is the response sent back from the SDO server to the client indicating that the previously received upload (read) segment request was processed successfully. The data segment is part of this message.

The Abort SDO Transfer

At any time the client or server may abort an SDO transmission. The error code gives an indication as to why the transfer was aborted. Typical errors are that a desired Object Dictionary entry is not implemented by the SDO server or that the entry is of a different length (for example writing a 2-byte value to a 4-byte entry).

Network Management (NMT) Communication

The NMT Master Message

The NMT Master message has the CAN message identifier 0 (zero) and contains 2 bytes. All CANopen slave nodes must be able to receive this message and act upon its content.

The Heartbeat

The heartbeat message sent by an individual node has the CAN message identifier 700h plus the Node ID. It only contains one byte showing the current NMT state of that node.

Emergency Communication

The emergency message sent by an individual node has a CAN identifier of 80h plus the Node ID.

SDO Block Transfer The block transfer mode is an optimized transfer mode for Object Dictionary entries that contain large amounts of data. In this transfer mode, up to 889 bytes (segmented into 127

messages with each 7 bytes) are combined into one data block and are transmitted using back-to-back messages.

The block transfer mode is optional and can only be used if both client and server support this communication mode. If one of the nodes does not support block transfer, the segmented or expedited transfer has to be used.

A download is divided into the following communication stages:

- Initiate Block Download - Client requests from Server to use block transfer mode for a download.
- Download Blocks - Client sends data blocks to Server and expects one response per block. Each block contains up to 127 segments.
- End of Download Block - Client and Server confirm that the transmission is now complete.

At any stage, any of the two communication partners may abort the transfer by sending an abort SDO transfer message.

Initiate Block Download

To initiate a block download the client sends the request to the server to which the server sends a response.

Download Blocks

After successful initiation, the client starts transmitting the blocks. Each block consists of as many segments as specified by "blksize" during initiation. At the end of a block, the client expects the server to send a response.

End of Download Block

After the client transmitted all blocks and the server acknowledged all blocks, the client and server confirm to each other if the transmission was successful.

Initiate Block Upload

To initiate a block upload a total of three messages are exchanged.

Upload Blocks After successful initiation, the server starts transmitting the blocks. Each block consists of as many segments as specified by "blksize" during initiation. At the end of each block, the server expects the client to send a response.

End of Upload Block

After the server transmits all blocks and the client acknowledges all blocks, the client and server confirm to each other if the transmission was successful.

CRC Calculation

The Cyclic Redundancy Checksum used for the block transfer has 16 bits and is calculated over the entire data range of each block.