Chapter 9
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

H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Real-Time Systems Series, DOI 10.1007/978-1-4419-8237-7_9,

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.

9.1 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 introduced in Sect. 4.4. 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. The TII, the LIF and the TDI are discussed in this section, while the *Local Interface* is discussed in the section on process input/output.

9.1.1 Technology Independent Interface

In some sense, the technology independent interface (TII) is a meta-level interface that 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 (see Sect. 4.2.3) 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.

9.1.2 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. The LIF has been discussed extensively in Sect. 4.6.

9.1.3 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.

9.1.4 Generic Middleware

The software structure within a component is depicted in Fig. 9.1. 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 r*estart 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.

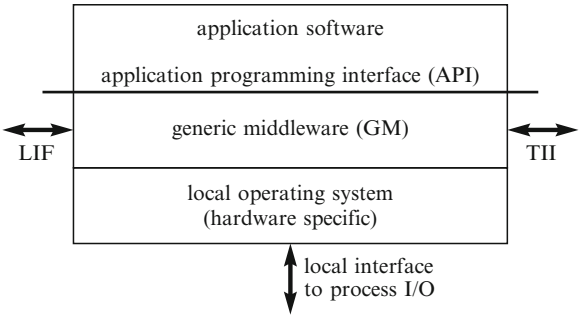

Fig. 9.1 Software structure within a component

9.2 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.

9.2.1 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 distinguish *time-triggered* (*TT*) *tasks* and (*ET*) *event-triggered tasks*. A *cycle* (see Sect. 3.3.4) 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 (Fig. 9.2). 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

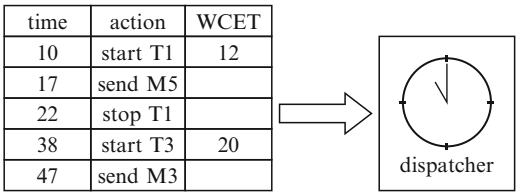

Fig. 9.2 Task descriptor list (TADL) in a TT operating system

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.

9.2.2 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 intolerable and the implementation of an interrupt is needed.

9.2.3 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*.

9.3 The Dual Role of Time

A real-time image must be *temporally accurate* at the *instant of use* (see Sect. 5.4). 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 is required as a consequence of the event occurrence. The occurrence of the event must initiate a control action without delay.

9.3.1 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 µs. 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 worst-case reaction time of 100 ms.

9.3.2 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.

9.4 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 9.3 depicts the problem. Two tasks, T1 and T2 access the same critical region of data. We call

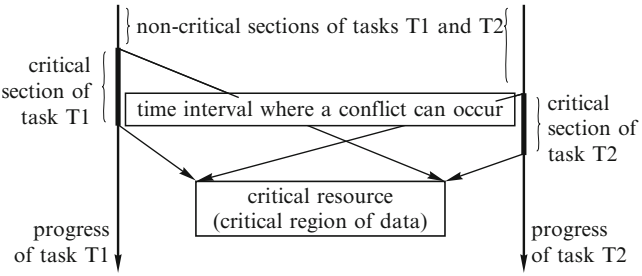

Fig. 9.3 Critical task sections and critical data regions

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

9.4.1 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.

9.4.2 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 [Kop93a] 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

```
initialization: CCF := 0;

writer:                              reader:
start: CCF_old := CCF;               start: CCF_begin := CCF;
       CCF := CCF_old + 1;                  if CCF_begin = odd
       <write to data structure>            then goto start;
       CCF := CCF_old + 2;                  <read data structure>
                                            CCF_end := CCF;
                                            if CCF_end ≠ CCF_begin
                                            then goto start;
```

Fig. 9.4 The non-blocking write (NBW) protocol

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 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 (see Fig. 9.4).

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 [Kop93a].

Non-locking synchronization has been implemented in other real-time systems, e.g., in a multimedia system [And95]. It has been shown that systems with non-locking synchronization achieve better performance than systems that lock the data.

9.4.3 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.

9.5 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*.

9.5.1 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

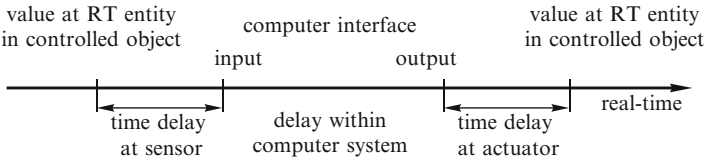

Fig. 9.5 Time delay of a complete I/O transaction

determined by the transfer function of the particular sensor. The step response of a sensor (see Fig. 1.4), 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 (Fig. 9.5). 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* (see Sect. 3.3.4).

9.5.2 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 (Fig. 9.6), called the *contact bounce*, caused by the mechanical vibrations of the switch contacts. This contact bounce must be eliminated either by an analog low-pass 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. If the wheel travels past a defined *calibration point*, an additional digital input is signaled

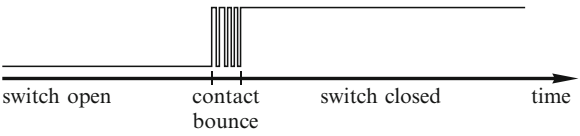

Fig. 9.6 Contact bounce of a mechanical switch

to the computer to set the pulse counter to a defined value. It is good practice to convert the relative event values to absolute state values as soon as possible.

Time Encoded Signals. Many output devices, e.g., power semiconductors such as IGBTs (insulated-gate-bipolar transistors), are controlled by pulse sequences of well-specified shape (pulse width modulation – PWM). A number of microcontrollers designed for I/O provide special hardware support for generating these digital pulse shapes.

9.5.3 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. [Pol95b] 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 [Eid06].

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.

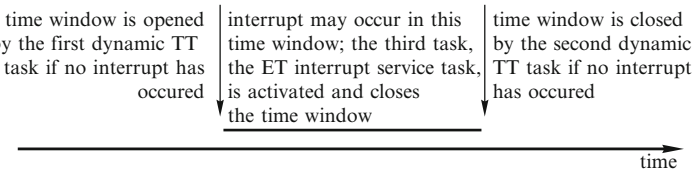

Fig. 9.7 Time window of an interrupt

Monitoring the occurrence of an interrupt. There are three tasks in the computer associated with every monitored interrupt [Pol95b] (Fig. 9.7). The first and second one are dynamically planned TT tasks that determine the interrupt window. The first one enables the interrupt line and thus opens the time window during which an interrupt is allowed to occur. The third task is the interrupt service task that is activated by the interrupt. Whenever the interrupt has occurred, the interrupt service task closes the time window by disabling the interrupt line. It then deactivates the scheduled future activation of the second task. In case the third task was not activated before the start of the second task, the second task, a dynamic TT task scheduled at the end of the time window, closes the time window by disabling the interrupt line. The second task then generates an error flag to inform the application of the missing interrupt.

The two time-triggered tasks are needed for error detection. The first task detects a sporadic interrupt that should not have occurred. The second task detects a missing interrupt that should have occurred. These different errors require different types of error handling. The more we know about the regularity of the controlled object, the smaller we can make the time window in which an interrupt may occur. This leads to better error-detection coverage.

Example: An engine controller of an automotive engine has such a stringent requirement regarding the point of fuel injection relative to the position of the piston in the cylinder that the implementation must use an interrupt for measuring the position [Pol95b]. The position of the piston and the rotational speed of the crankshaft are measured by a number of sensors that generate rising edges whenever a defined section of the crankshaft passes the position of the sensor. Since the speed and the maximum angular acceleration (or deceleration) of the engine is known, the next correct interrupt must arrive within a small dynamically defined time window from the previous interrupt. The interrupt logic is only enabled during this short window and disabled at all other times to reduce the impact of sporadic interrupts on the temporal control pattern within the host software. Such a sporadic interrupt, if not detected, may cause a mechanical damage to the engine.

9.5.4 Fault-Tolerant Actuators

An actuator must transform the signal generated at the output interface of the computer into some physical action in the controlled object (e.g., opening of a valve). The actuators form the last element in the chain between sensing the values of an RT-entity and realizing the intended effect in the environment. In a

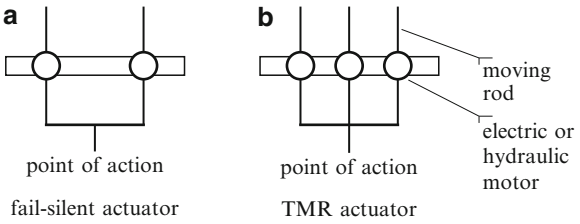

Fig. 9.8 Fault-tolerant actuators

fault-tolerant system, the actuators must perform the *final voting* on the output signals received on the replicated channels. Figure 9.8 shows an example where the intended action in the environment is the positioning of a mechanical lever. At the end of the lever there may be any mechanical device that acts on the controlled object, e.g., there may be a piston of a control valve mounted at the point of action.

In a replica-determinate architecture, the correct replicated channels produce identical results in the value and in the time domains. We differentiate between the cases where the architecture supports the fail-silent property (Fig. 9.8a), i.e., all failed channels are silent, and where the fail-silence property is not supported (Fig. 9.8b), i.e., a failed channel can show an arbitrary behavior in the value domain.

Fail-Silent Actuator. In a fail-silent architecture, all subsystems must support the fail-silence property. A fail-silent actuator will either produce the intended (correct) output action or no result at all. In case a fail-silent actuator fails to produce an output action, it may not hinder the activity of the replicated fail-silent actuator. The fail-silent actuator of Fig. 9.8a consists of two motors where each one has enough power to move the point of action. Each motor is connected to one of the two replica-determinate output channels of the computer system. If one motor fails at any location, the other motor is still capable to move the point of action to the desired position.

Triple Modular Redundant Actuator. The triple modular redundant (TMR) actuator (Fig. 9.8b) consists of three motors, each one connected to one of the three replica-determinate output channels of the fault-tolerant computer. The force of any two motors must be strong enough to override the force of the third motor, however, any single motor may not be strong enough to override the other two. The TMR actuator can be viewed as a *mechanical* voter that will place the point of action into a position that is determined by the majority of the three channels, outvoting the disagreeing channel.

Actuator with a Dedicated Stateless Voter. In many applications where redundant actuators are already in place, a voting actuator can be constructed by combining the physical actuator with a small microcontroller that accepts the three input channels from the three lanes of a TMR system and votes on the messages received from the three lanes. This voter can be stateless, i.e., after every cycle the circuitry of the voter is reset in order to eliminate the accumulation of state errors caused by transient faults (Fig. 9.9).

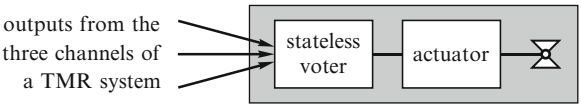

Fig. 9.9 Stateless voter associated with an actuator

Example: In a car, a stateless voter can be placed at the brake actuator at each one of the four wheels. The voter will mask the failure in any one of the TMR channels. A stateless voter is an example for an intelligent instrumentation.

9.5.5 Intelligent Instrumentation

There is an increasing tendency to encapsulate a sensor/actuator and the associated microcontroller into a single physical housing to provide a standard abstract message interface to the outside world that produces *measured values* at a field bus, e.g., a CAN bus (Fig. 9.10). Such a unit is called an *intelligent instrument*.

The intelligent instrument hides the concrete sensor interface. Its single chip microcontroller provides the required control signals to the sensor/actuator, performs signal conditioning, signal smoothing and local error detection, and presents/takes a meaningful RT image in standard measuring units to/from the field bus message interface. Intelligent instruments simplify the connection of the plant equipment to the computer.

Example: A MEMS acceleration sensor, micro machined into silicon, mounted with the appropriate microcontroller and network interface into a single package, forms an intelligent sensor.

To make the measured value fault-tolerant, a number of independent sensors can be packed into a single intelligent instrument. Inside the intelligent instrument, an agreement protocol is executed to arrive at an agreed sensor value, even if one of the sensors has failed. This approach assumes that independent measurements can be taken in close spatial vicinity.

The integration of a field bus node with an actuator produces an intelligent actuator device.

Example: An actuator of an airbag in an automobile must ignite an explosive charge to release the gas of a high-pressure container into the airbag at the appropriate moment. A small explosive charge, placed directly on the silicon of a microcontroller, can be ignited on-chip. The package is mounted at the proper mechanical position to open the critical valve. The microcontroller including the explosive charge forms an intelligent actuator.

Because many different field bus designs are available today, and no generally accepted industry wide field bus standard has emerged, the sensor manufacturer must cope with the dilemma to provide a different intelligent instrument network interface for different field buses.

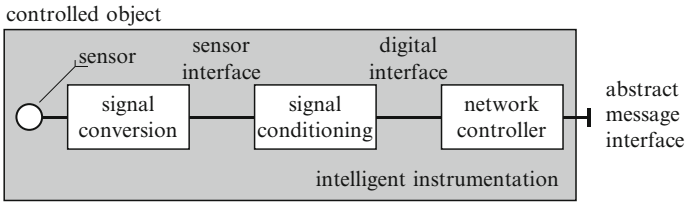

Fig. 9.10 Intelligent instrumentation

9.5.6 Physical Installation

It is beyond the scope of this book to cover all the issues that must be considered in the physical installation of a sensor-based real-time control system. These complex topics are covered in books on computer hardware installation. However, a few critical issues are highlighted.

Power Supply. Many computer failures are caused by power failures, i.e., long power outages, short power outages of less than a second also called sags, and power surges (short overvoltage). The provision of a reliable and clean power source is thus of crucial importance for the proper operation of any computer system.

Grounding. The design of a proper grounding system in an industrial plant is a major task that requires considerable experience. Many transient computer hardware failures are caused by deficient grounding systems. It is important to connect all units in a tree-like manner to a high quality true ground point. Loops in the ground circuitry must be avoided because they pick up electromagnetic disturbances.

Electric Isolation. In many applications, complete electric isolation of the computer terminals from the signals in the plant is needed. Such isolation can be achieved by opto couplers for digital signals or signal transformers for analog signals.

9.6 Agreement Protocols

Sensors and actuators have failure rates that are considerably higher than those of single-chip microcomputers. No critical output action should rely on the input from a single sensor. It is necessary to observe the controlled object by a number of different sensors and to relate these observations to detect erroneous sensor values, to observe the effects of actuators, and to get an *agreed* image of the physical state of the controlled object. In a distributed system *agreement* (also called *consensus* in [Bar93]) always requires an information exchange among the agreeing partners. The number of rounds of such an information exchange needed depends on the type of agreement and the assumptions about the possible failure modes of the partners.

9.6.1 Raw Data, Measured Data, and Agreed Data

In Sect. 1.2.1, 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*.

9.6.2 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 (see Sect. 6.4.2).

9.6.3 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 (see Sect. 6.1.4) 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.

9.7 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.

9.7.1 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 (see Sect. 10.2). 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 up to the application to specify which action should be taken in case of an error.

9.7.2 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 (see Sect. 9.5.3).

9.7.3 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 [Arl03]. The operating system can provide the execution environment for the double execution of application tasks without demanding any changes to the application task per se. 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.

9.7.4 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.

Points to Remember

- We distinguish two levels of system administration in a component-based distributed real-time system: (1) the coordination of the message-based communication and resource allocation among the components, and (2) the establishment,

coordination of, and control of the concurrent tasks within each one of the components.

- Since 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*.
- In an entirely time-triggered system, the static temporal control structure of all tasks is established a priori by off-line scheduling tools. 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 is not necessary.
- In a 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. 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 timing analysis of a C-task is not a local issue of a single task anymore; it becomes a global system issue. In the general case it is impossible to give an upper bound for the WCET of a C-task.
- It is important to distinguish *time as data* and *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.
- Care must be taken that the integrity of data that is read or written concurrently by more than one task is maintained. In a time-triggered system, the task schedules can be constructed in such a way that critical sections of tasks do not overlap.
- 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* in a time-triggered system.
- 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.
- A voting actuator may be constructed by assigning a small microcontroller to the physical actuator that accepts the three input channels of the three lanes of a TMR system and votes on the messages received from the three lanes.
- 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.

Bibliographic Notes

Many of the standard textbooks on operating systems contain sections on real-time operating systems, e.g., the textbook by Stallings [Sta08]. The most recent

research contributions on real-time operating systems can be found in the annual Proceedings of *the IEEE Real-Time System Symposium* and the Journal *Real-Time Systems* from Springer Verlag.

Review Questions and Problems

9.1 Explain the difference between a standard operating system for a personal computer and an RT operating system within the node of a safety-critical real-time application!

9.2 What is meant by a *simple task*, a *trigger task*, and a *complex task*?

9.3 What is the difference between *time as data* and *time as control*?

9.4 Why is the classical mechanism of *semaphore operations* sub-optimal for the protection of critical data in a real-time OS? What alternatives are available?

9.5 How is *contact-bounce* eliminated?

9.6 When do we need interrupts? What is the effect of spurious interrupts? How can we protect the software from spurious interrupts?

9.7 A node of an alarm monitoring system must monitor 50 alarms. The alarms must be reported to the rest of the cluster within 10 ms by a 100 kbit/s CAN bus. Sketch an implementation that uses periodic CAN messages (time-triggered with a cycle of 10 ms) and an implementation that uses sporadic event-triggered messages, one for every occurring alarm. Compare the implementations from these points of view: generated load under the conditions of no alarm and all alarms occurring simultaneously, guaranteed response time, and detection of a crash failure of the alarm node.

9.8 Let us assume that an actuator has a failure rate of 10^6 FITs. If we construct a voting actuator by adding a microcontroller with a failure rate of 10^4 FITs to this actuator, what is the resultant failure rate of the voting actuator?

9.9 What is the difference between *raw data*, *measured data*, and *agreed data*?

9.10 What is the difference between *syntactic agreement* and *semantic agreement?* Which technique is more important in the design of real-time applications?

9.11 List some of the generic error-detection techniques that should be supported by a real-time OS!

9.12 Which types of failures can be detected by the double execution of tasks?