Software for Real-time & Embedded Systems (B-KUL-H04L2A)

Professor(s)
Marc Lobelle

Authors
Dieter Castel
Jonas Devlieghere

Contributors
Wout Scheepers

Contents

Preface

The course **Software for Real-time and Embedded Systems** is part of the *Master of Engineering: Computer Science* at the KU Leuven. It is given by Professor Marc Lobelle.

This document contains a transcription of the course material found on the *Foditic* website.

The source and latest version of this document is located at `https://github.com/JDevlieghere/SORTES-Course`. We encourage future readers to contribute to the development of this document.

Part I

Real-Time Systems

Chapter 1

Introduction

1.1 Classification

The table below shows how computing systems can be classified. Please note that a system can refer to a simple program or a whole computer system. A system can be classified by its response time requirements or by the need of external interaction.

- **Non Real-Time Data Processing** There is no external stimulation of the system, it justs processes data. There are no time constraints.
- **Real-Time Data Processing** The system is paced by their external environment. The response time constraints are twofold: The **accept data** rate is dictated by the producer, the **processing rate** is dependent on the external environment.
- **Non Real-Time Reactive (Interactive)** Systems stimulated by the human user. The response time is expected to be as short as possible. The pace is set by the slowest entity: the user.
- **Real-Time Reactive** The external environment controls the system which processes data at peak rate.

Real-Time systems have a response time constraint. Usually this constraint is not really a timing constraint but rather an expression relative to a physical measurement unit.

Example 1.1.1. Take for example a carriage that must be stopped before it hits a bumper. The physical measurement unit here is **distance**. This constraint can easily be converted to a timing constraint given it's initial speed, braking characteristics, etc.

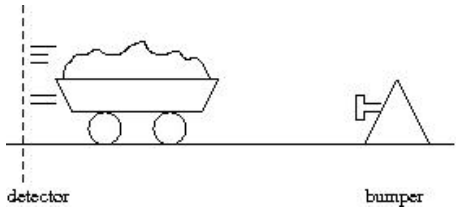

Figure 1.1: Example: Carriage stopping before bumper.

However, since a computer has no notion of distance, specifications in terms of physical units have to be converted to a timing constraint. When it comes to specifications on the other hand, it's better to include the original constraint than the computed timing constraint. This being said, the real-time programmer usually is only in charge of the computing system and cannot modify the original parameters.
Another way to classify real-time systems is by dividing them in **stand-alone systems** and systems that are **part of a hierarchy**. This course focusses on **embedded systems**: they are a component built-in to other equipment. They can either work alone or be part of a distributed system.

Example 1.1.2. An example of an **embedded system that works alone** is the electric ignition of a car, a modern washing machine or an electronic scale.

Example 1.1.3. An example of a **embedded system that is part of a distributed system** is a network programmable controller, a programmable machine tool or more generally controllers dedicated to a component of a larger system.

1.2 Real-Time Architecture

The classical architecture of a distributed real-time system is a two layer architecture:

1. Dedicated System
2. Coordinating systems

1.2.1 Dedicated Controllers

The dedicated controller usually handles tasks with critical real-time constraints. Such systems are called **hard real-time systems**. Decisions taken by the system are done so locally based on the current available information. The software's structure is simple so it offers deterministic and predictable behavior.

1.2.2 Central System

The central system often deals with less critical external constraints. However, the farther remote from the controlled system, the more *unpredictable events* increase the difficulty to satisfy the constraints.

The system centralizes information in order to provide a global view on the whole system. The operator interface is managed by this system and offers an overview of the situation, consisting of a global view and selected useful details. In a more advanced application the system might suggest actions to be undertaken by the user.

1.2.3 Connections

The connection type between the dedicated controller and the central system depends on the environment. In an industrial environment optical fibers might be preferred to copper wires because they are not sensitive to voltage differences at the ends of the link. Ethernet can be used on condition its behavior is deterministic. Specialized networks exist for real-time systems (**field busses**). Although they are not really fast they are cheap and deterministic. Either way regular traffic should be separated from the real-time traffic of the company.
In some cases the internal system bus of the controlling system is used instead of a network. Supplemental processors are then inserted in this bus, each with its own OS and its own memory. These processors then communicate through shared memory areas.

1.2.4 System Types

- **Coördinating System** (must be compatible with the external constraints)
  - PC or equivalent
  - Full fledged operating system (real-time variant of UNIX or Linux)
- **Dedicated Controllers** (must be compatible with the system dedicated to)
  - Programmable logic controllers
  - Small computers with specialized operating systems
  - Embedded Linux devices

PLC: Programmable Logic Controller

PLCs are small computers, generally micro-controllers, executing a programmable logic controller language interpreter. They do not offer interrupts and programs are structured as a single infinite loop. At the beginning of the loop data is read, it is then processed and finally a control signal is outputted.

The content of the loop might exist of an interpreter simulating a logic circuit. The language interpreted is then called a **ladder diagram**. Another possibility is the interpreter simulating a simple state machine. The programming language is then called **GRAFCET**. At each loop iteration a state transition is triggered based on the output of the previous state.

Ladder Diagram The ladder diagram is a language understandable by electrical technicians. Because its expressive power is so limited manufactures have enhanced it. The PLC interprets all the lines of the diagram, which can easily span 20 or more pages, each loop. Note that if variables are used, the order of the lines become important.

GRAFCET This is a graphical language loosely inspired by PETRI nets. Rather than using logical circuits, a sequential state machine is used as a model. Although this approach is suitable for simple problems, as the problem gets more advanced, the diagram becomes equally complex.

Microcomputer with Dedicated OS

The operating system is limited but offers simple but efficient functionality:

- Multitasking
- Fast task switching
- Interrupt support
- Inter-process communication and synchronization

Although this is more versatile than a PLC, a real programmer is needed to design the software. The more complex the software gets, the harder it becomes to guarantee response times and correctness.

1.3 Time in a Real-Time System

As mentioned in the first section, timing constraints are different for different types of systems:

- **Batch processing** requires the amount of throughput, in terms of processing, to be maximized withing the available time.
- **Interactive systems** need short response times but only as short as the user can detect.
- **Real-Time systems** require the action to occur before a specific event or deadline.

Unlike the other cases, in a real-time system, the value of work does not decrease with the time needed to do the work.

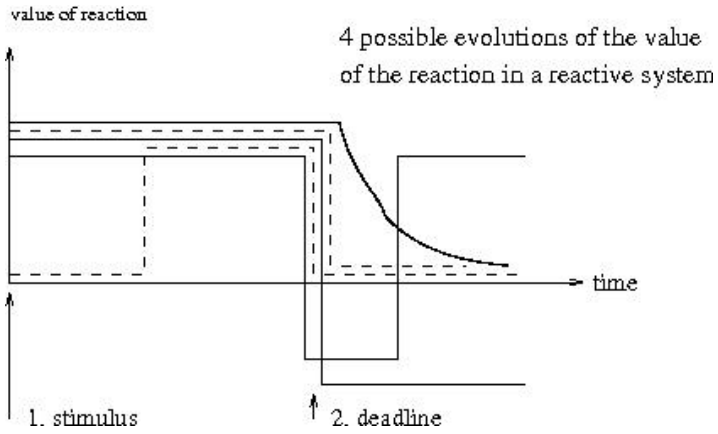

Figure 1.2: Example: Time in a real-time system.

The reaction to the stimulus (the event that triggers the response) must be performed within a certain time interval following it. (this time interval may start with the stimulus or later) During this time interval, the value of the response is constant. When the time interval is elapsed, the value of the response can:

- decrease *(like when cooking a soft boiled egg)*
- fall immediately to zero *(the carriage hits the obstacle)*
- become negative *(a hole is drilled in the printed circuit board, but at the wrong place)*
- become temporarily negative and then become zero *(the radar flashes the slow car driving in the right lane instead of the fast car speeding in the left lane)*

1.4 Design Strategy

When designing a real-time system the design must be as general as possible. We have to consider the hard- and software. We have to take into account the system being controlled, which is often a mechanical device and the interface between the device and the controlling system.

Example 1.4.1. By adding a simple hardware security to the controlled device, the diagram representing the value of the response in function of time can be made non-negative.

Example 1.4.2. By selecting the proper type and position of captors, the controlling system can be warned earlier and expand the acceptable reaction time span.

Example 1.4.3. Adding a DMA(Direct Memory Access) controller to the computing system makes higher rates of data acquisition possible and inversely, not adding one when none is needed makes the system cheaper.

Chapter 2

Design strategy for embedded real-time systems

2.1 Introduction

2.1.1 Time-related constraints

The following constraints should be carefully examined before designing the system:

- Constraints related to the device under control of the system.
- Constraints related to the computer(s) running the software.

  If you can't choose this : The first thing to check is if the computer can meet the needed constraints of the device it will control.

  If you can chose this First design the software assuming you have unlimited computing resources. Then select a suitable computing infrastructure.

  The goal of this strategy is to avoid that the infrastructure you selected is not able to meet the timing constraints. On top of that it helps to focus on the most important constraints of the device that will be controlled while designing the software.

2.1.2 Software architecture

Real-time software has a two-layer architecture:

1. real-time application layer
2. real-time operating system layer

Both layers are equally important in designing a real-time system and should be carefully designed or chosen. The OS can be designed alongside the application or existing solutions can be used. Mostly using a specifically designed OS is generally faster since versatility costs time.

2.2 Design Strategy

The creation of a real-time system is done using 6 steps. These steps are listed below and each step is discussed in further detail in this section.

1. Specify the problem. (see 2.2.1)
2. Design the software architecture. (see 2.2.2)
3. Identify the services the system layer must provide. (see 2.2.3)
4. Validation or design of the system layer. (see 2.2.4)
5. Specify the data processing activities. (see 2.2.5)
6. Write the program(s). (see 2.2.6)

2.2.1 Specify the problem

The first and most important step in designing is specifying the problem. This is "classical" analysis work. Errors made in this step of the process are most costly so caution is advised. Try to do this without preselecting a solution. The following models can help you model the problem:

Structural Model Model of all the components, subsystems that can work in parallel.

Behavioral Model Model of the desired behavior e.g. state machines of components.

Analysis

The analysis can cover an existing physical system (control systems are more frequently replaced than factories!) or specifications for a future physical system. In the latter case the following should be considered.

1. Are the specifications complete?
2. Is the future system fully understood?
3. Do the specifications cover exactly what the author meant?

Because of these questions the created model should be shown to the author of the specifications: He *must* **understand** and **agree** with the model.

2.2.2 Design the software architecture

Next the software architecture can be designed by refining the behavioral model. Two types of systems can be distinguished:

Reactive system The desired behavior of the controlled device depends on the commands from the system we are designing. In this case the behavior of the software must be *structurally similar* to that of the **controlled system**. The software sends commands to force the expected behavior of the controlled device.

Data processing system The behavior of the software must be *structurally similar* to that of the **monitored system** and is driven by events occurring in that system.

In both cases the following process should be followed:

- Model the structure of the physical system.
- Make a behavioral model of the physical system.
- Make a behavioral model of the software system.

The last step should give you a rough skeleton of the application layer.
The tasks of the software should be identified. A task can be seen as a sequence of states that may evolve in parallel. Almost always there are multiple possible sequences of states that represent the same global behavior. It is always possible to represent a behavior with entirely sequential (mutually exclusive) set of states. Generally this is not advised because the amount of states (and transitions) tends to get very large quickly. That's why it's advised to try and *maximize parallelism* in the behavioral model. This gives us a group of simpler sequential state machines evolving in parallel and interacting only when needed. The following task interactions are possible:

- Information Transfer
- Synchronizations
- Exclusions
- Filiations (relationships)

2.2.3 Identify the services the system layer must provide

See what kind of (common) services should be required in the system layer. This is "classical" analysis work of the behavioral model used to deduce the system services need.

2.2.4 Validation or design of the system layer

In this step the system layer is evaluated or created depending on whether an OS is available. If an is OS available it should be evaluated as follows:

- Does it provide the required services and respect the real-time constraints?
- If not can it be adapted to satisfy these?
- If not can the proposed solution for the application layer be adapted to the possibilities of the available OS?

If no OS is available design the system layer so that it contains all the needed services of step 3. Firstly the services that should be provided should be specified. Two categories of services exist:

- Basic services: to be implemented

  - scheduler (can be very simple in some cases).
  - low level routines: To act as an interface to the hardware (e.g. interrupt routines) or for basic interactions between tasks (e.g. synchronization)

- Derived services: Built on top of others (e.g. file systems).

Another categorization is possible based on how services are activated:

- Activation by application layer.
  - With function calls: system layer must be linked directly to the application layer (single executable).
  - With system calls: Independent modules, context switch is possible and both layers can reside in independent memory.
- External Events (e.g. when a character is received, activation by interrupts).

2.2.5 Specify the data processing activities

Each of the behavioral states of the software should be complemented with a descriptions of the data processing activities activities at that state. How these activities are represented depending on the type of data to be processed. Sometimes these descriptions can be as simple as a few lines of C/C++ or even a ladder diagram.

2.2.6 Write the program(s)

The design was top down but the writing of the programs should be bottom-up. So first the system layer and next the application layer. Testing is vitally important so test as soon as as possible.

Implementation of the system layer

A minimal system layer implementation is described here.

Scheduling We simplify task scheduling by assuming there are only two kinds of tasks:

- Short tasks: Performed in response to an external event.
- Periodic tasks: Tasks re-executed with a fixed periodicity. Periodicity can be absolute (e.g. 2ms) or relative (e.g. X 3 times as often as Y).

Short tasks can be implemented as interrupt routines. Controlling these tasks can be done with interrupt masks and (static) priorities can be assigned according to the associated interrupt lines.
Periodic tasks should be implemented with a timer/clock (naive implementation is possible but dirty). To do so use clock interrupts and at each tick launch the needed tasks. It's important that the sum of the durations of the tasks launched at any given clock tick must be less than the clock period. An example can be seen in figure 2.1.

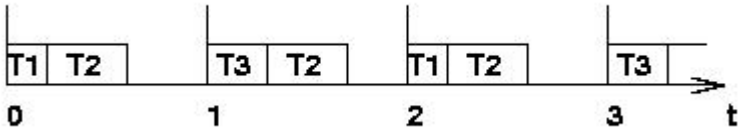

Figure 2.1: Example of periodic tasks: At even clock ticks T1 and T2 are launched at uneven clock ticks T3 and T2.

Using this kind of scheduling doesn't put a 100% load on the cpu. The lengths of the task can also vary a little if the sum of durations remains under the clock period. When calculating the clock period, the interrupt task's duration should be taken into account.

Task synchronization We'll explain a simple form of task synchronization next. The idea is that a tasks can suspend the execution of another until a given event. First the scheduler must be warned that the task has been suspended. Next some bookkeeping needs to be done in the table of tasks. Make the appropriate task un-runnable by adding a code representing an event that reactivates it in the "lock" field. To restart the task another task will have to detect the unlocking event and make the "lock" field null again. If more locked tasks have the same unlocking event a choice must be made to either unlock the first task (or with highest priority) or all the tasks waiting.
Other synchronizations can be implemented in a similar way.

Communications Two methods can be distinguished for task communication:

By Shared Variables It's simple and fast but shared variables must be written mutual exclusively. This makes the tasks strongly synchronized (long waits may occur).

By Messages The OS acts as a go-between. It can buffer the messages. Mutual exclusion is needed between tasks and OS. Waits can occur when the buffer is full. But messages can also be discarded if necessary an appropriate discarding scheme should then be chosen (often fifo). This buffering cause the tasks to be less synchronized.

Mutual Exclusion On a mono-processor mutual exclusion can sometimes be achieved with temporarily disabling interrupts. In any other case this can be implemented by using atomic READ/MODIFY/WRITE assembly language instructions. On a multiprocessor the memory access between the READ and WRITE will be blocked for all but one processor. Examples of READ/MODIFY/WRITE instructions are: TSET (test & set) and ASL (arithmetic shift left). For the latter we refer to image 2.2.

Programming

Which programming language should be chosen to implement the system layer? In general assembly language should be avoided, except when there are no other options (e.g. TSET instruction or tiny programs). A better choice would be an **extensible language** (e.g. C or FORTH). Interactions with the system in such languages goes through functions linked with the program that are not part of the language definition. A program can thus be interfaced

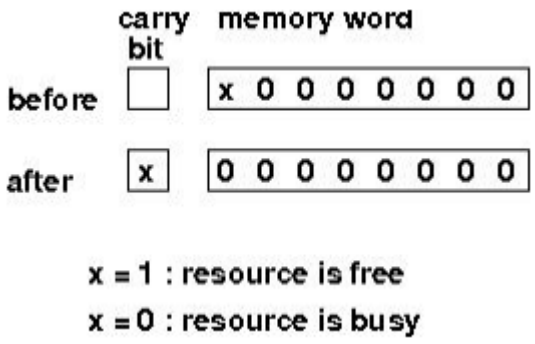

Figure 2.2: Figure showing how Arithmetic Left Shift works.

with any system layer by writing a new set of system interface functions (in the language or assembly). There are even some **dedicated languages** for embedded systems. These are usually imperative languages including following built-in functionalities for:

- Hardware Interaction (e.g. interrupt routines).
- Multi-programming management
- Inter-process communication
- Process synchronization

They also have some security improvements over extensible languages. Because they are usually modular and have those built-in functions, verifications of system interactions and syntax is possible by the compiler. Examples of dedicated languages are: ADA, MODULA 2 and PEARL

Deployment

After programming appropriate executable modules should be made. Most compilers are designed to make programs that run on top of an operating system. Executables for embedded systems have often specific **memory allocation constraints**:

- Interrupt Vectors
- Programs to be put in ROM
- Data
- Stack

The linker should be able to manage these facts and accept directives specifying the desired location and type of memory where the sections of the program should be placed in. The following are rules the linker/compiler should take into account:

- What parts of the program go in each section.
- Where is each section placed in memory.
- Is the section ROM or RAM.

Documentation

Three main documentation documents are required:

For the user Manual and description of behavior, functionalities of the program.

For the system engineer Guidelines to set up and deploy the program successfully.

For the designer This should be extensive documentation of all 6 steps for the designer himself or his possible successors. When documenting a posteriori there is a risk to forget important design decisions or strategic choices by focusing to much on the last implementation details.

Chapter 3

Identification of Suitable Hardware

To select suitable hardware for a real-time embedded system we should first define what suitable means in this case. Suitable hardware is hardware that can satisfy the given (time) constraints and is not too overdimensioned.

3.1 Select a maximal resource structure

- Each process executes on a different processor and has all the physical resource it needs.
- Processors are fast enough to satisfy the time constraints in this configuration. If the type of processor is predefined it should be verified that the processor can solve the given problems with a maximal resource structure. Else it means selecting one.
- Adding more resources of the same time is not useful

The following are properties of a maximal resource structure.

- A process is never "READY" (never waiting for a process to run on): It's either "RUNNING" or "WAITING" (for an event or resource).
- The analysis of the worst case response time is easier to compute because processes don't interfere because of resource access conflicts. The only interferences are those dictated by the specification.
- It satisfies response time constraints if all the different stimuli occur simultaneously at boot time. And if there is no accumulation of debts from the past (calculations of a process finished before the next stimulus arrives).

3.2 Reduce the structure by transformation

Don't try to find the minimal structure (this problem is NP-complete) but a realistic solution. To do so, follow these to rules to restrict choices:

1. No process migration (not a good idea in real-time)
2. A processor is never idle if it can run one of the processes assigned to it.

Example 3.2.1. *Given is that:* 2 processes, P1 and P2, receive each an external stimulus (resp. x et y). After some pre-processing, they send the result to a process P3 that, after some processingn warns process P4, that reacts on the external system.
Solution: If we neglect message transmission times, one can represent what the 4 processes would do, on a maximal resource structure, if all stimuli occurred at boot time. That gives us our (clearly correct) maximal resource structure as seen in 3.1. Not neglecting transmission times is an immediate adaptation of the representation. One can regroup P1 and P4 on a same processor, without lengthening the reaction times. It would also have been possible to regroup P2 and P4. Then the response time would have been increased, but would still satisfy the constraint. As, in normal working conditions (thus without taking initialization into account), P2 is working the most, it will be P2 that will determine the lowest possible repetition period of the stimuli (duration of pre-processing + post-processing of P2). However, at that rate, no grouping of processes is possible anymore.

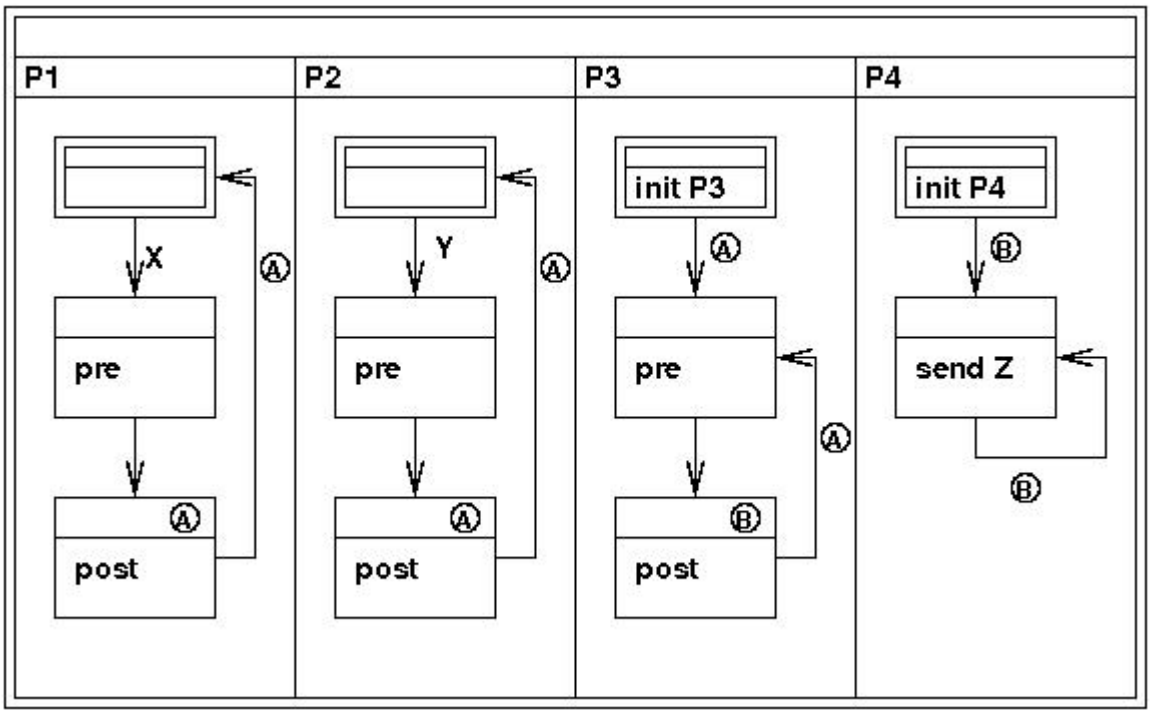

Figure 3.1: Figure representing the maximal resource structure of the given problem.

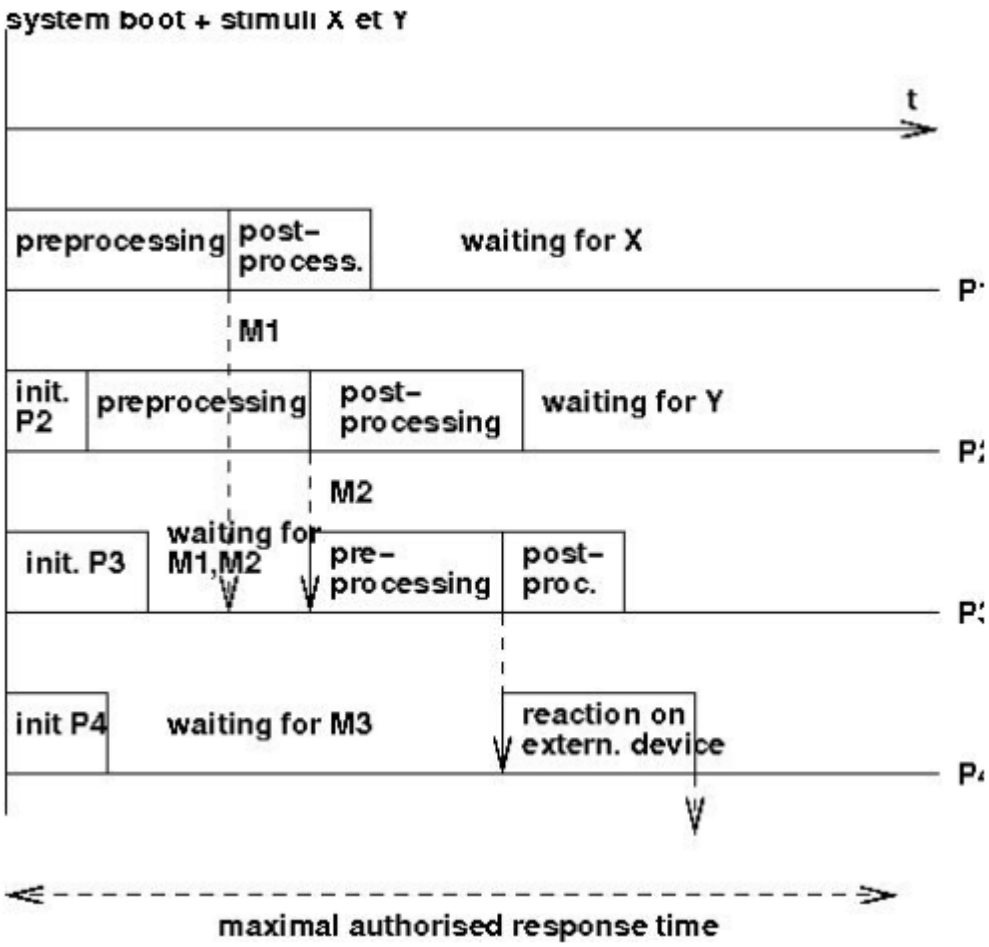

Figure 3.2: Figure representing the maximal resource structure of the given problem.

Chapter 4

Real-Time Operating Systems

4.1 Scheduling

The scheduling problem in a real-time operating system is finding a suitable order of task execution, more specifically one that satisfies the real-time constraints. When dealing with *hard real-time systems*, these constraints need to be satisfied, no matter what. The problem is twofold:

1. **Are there schedules that satisfy the constraints?**
   Can deadlines be maintained? What's the worst case processing time? To answer these questions processing time must be bound, resources must be available and interaction between processes possible.

2. **If such schedule(s) exist, find one!**

4.1.1 Classic Real-Time Scheduling

Static Scheduling

With static scheduling all decision are made **before the system is started**. Schedulability analysis is performed statically without the system running. Usually the program will consist of a loop that performs all tasks in sequence. Sometimes tasks are only executed a limited amount of times each iteration. The loop runs either continuously or is started every clock tick. The last approached is used by most programmable logic controllers.

Interrupts are often disabled, since they are not really necessary when we know on beforehand what's going to happen. If there are interrupt routines, they are taken into account as high priority periodical tasks which run at the highest possible repetition rate of the interrupt.

Static scheduling is only suitable for simple and immutable systems. However, system that do not change do not exist, so when anything changes the scheduling analysis needs to be repeated. This can be a problem, for example when a component needs urgent replacement by a non-identical one.

Dynamic Scheduling

Dynamic scheduling is used when static scheduling is not appropriate.

1. Perform a rough static scheduling analysis to evaluate the **worst-case processing time**. Also verify that, unless the scheduler makes a gross fault, the scheduler will be able to satisfy all timing constraints.

2. Let a dynamic scheduler choose the order of execution of tasks.

3. Unless it can be proven that the scheduler always makes the right choice, test should cover as many situations as possible in order to verify timing constraints are met.

Every mutli-tasking operating system (real-time or not) includes a scheduler. The classical qualities expected from the scheduler in an ordinary operating system consist of :

- **Fairness**: All tasks should receive a fair share of computing time.
- Optimization of the **mean response time**
- Optimization of **available processing time**

The real-time scheduler on the other hand has only one goal: guarantee that all real-time tasks will be finished before their deadline.

Independent of whether the scheduler is real-time, they can be divided in two classes:

- **Non preemptive**: when the a task is allowed to run, the scheduler waits for the task to finish before allowing another task to run.
- **Preemptive**: the scheduler may suspend the execution of a running task when it judges appropriate.

> At first sight, a preemptive scheduler who can suspend tasks in favor of more urgent tasks will obtain better results. This is not always the case because resource conflicts can arise. If the lower priority task has locked a resource which is needed by the more urgent task, this tasks may not be suspend before this resource is freed. If this is not the case, the higher priority task would simply block, waiting for the resource.

There are 3 techniques available for selecting the next task to run on a non-periodic real-time system.

- **Fixed Priorities**
  Each task is assigned a fixed priority which has to be chosen with care. The priority should incorporate the urgency as well as the criticality of the task. The scheduler always selects the task with all its resource available and the highest priority. This technique is easy to implement but if there's always a task with a high priority, the lower priority task might be overrun.

- **Closest Deadline First**
  The scheduler selects the task closest to its deadline. This implies the deadline of each task is known. This can only be known at run time.

- **Least Laxity First**
  This scheduling technique selects the tasks with the lowest latency. If the task is started after that time, it cannot respect its deadline. The deadline and maximum execution times of tasks must be known.

$$laxety = t_{deadline} - t_{now} - task\ execution\ time$$

Priority Inheritance

A scheduler always selects one of the tasks that are ready to run, which means all its resources are available. This can be a problem when several tasks need the same resource and one tasks needs the resource without it being released between subsequent runs.

Example 4.1.1. A task could have to send a message over the network to another machine. As long as the message is not entirely transmitted, no other task may send over the network. If not, what would travel on the network would be the beginning of the first message, followed by the second message, followed by the end of the first message. The resulting traffic would not make any sense.

To avoid the problem mentioned, a task can use a **mutex** on the resource he needs. If the mutex is free, the task can reserve it for itself, otherwise it waits for it to be released. The task is then suspended and only restarted when the resource is free.

Consider the following problem, called the **priority inversion** as illustrated below.

Example 4.1.2. There are 3 tasks T_i close to their deadlines with priority i. If there were no resource problems the scheduler would select them in the following order: T_1, T_2, T_3. Task T_1 and T_3 need the same resource. Assume that T_3 is running and using the resource and the scheduler learns that T_1 and T_2 must be run urgently. This is what would happen:

1. The scheduler suspends T_3 which is holding the resource.
2. Since T_1 cannot run, T_2 is chosen.
3. After T_2 only T_3 is available and the scheduler consequently activates it.
4. Finally T_1 is run as soon as T_3 finishes.

Because of the resource,T_2 has been executed before T1 instead of the opposite: This is called the priority inversion problem. What should have happened is that T_3 was allowed to finish so that the resource was released and T_1 could run before T_2.

To enforce correct behaviour, **priority inheritance** is used. The list of resources needed by each task is known by the scheduler. When a lower priority task is running and uses a

necessary resource, that task temporarily inherits the more urgent task's priority untill the resource is freed.

Notes:

- The higher the number of tasks with different priorities who need the same resource, the more often this problem arises.
- The longer a task uses a resource, the more this task will slow down a higher priority task even with our solution.
- If several tasks need a same set of resources at the same time, they must lock all resources atomically. If not, a deadlock might occur.

Effect of Interrupts

An interrupt management system behaves like a preemptive scheduler with fixed priorities. It shares the responsibility of allocating the computer's processing time with the scheduler but is hierarchically of greater importance.

The scheduler of the operating system can only allocate the processing time left by the interrupt routines. If we aim to have the system mainly managed by the scheduler, interrupt routines should be very short. The interrupt should inform the scheduler but the latter will have to decide for itself to activate it or not.

For timing calculations, one can try to estimate the maximum amount of time lost by interrupt routines and add this value to the task's own processing time.

Periodic Tasks

An interesting special case is that of systems that only have periodic tasks but with a different value of periodicity. If all tasks are independent and if it is possible to satisfy all timing constraints then the scheduler that always activates the task with the highest repetition rate will meet all the deadlines. This approach is called **rate monotonic scheduling**. Unfortunately, situations where all tasks are independent are rare.

4.1.2 Reservation Scheduling

Rather than executing the scheduling algorithm every time a new task must be selected, **Reservation Scheduling** will make this choice on beforehand, as soon as the necessary information is available.
If it is possible to schedule a task between the current time and the moment there will be no known tasks left to execute, a schedule is prepared. If the previous is not possible, the scheduler decides what tasks should be sacrificed, based on their importance (criticality) rather than their urgency. Sacrificed tasks may be replaced by shorter ones aimed at limiting the impact of sacrificing the task. Tasks that are urgent but not critical may simply be dropped.

Example 4.1.3. The considered system is an airplane. An example of a shorter task to replace the sacrificed task is the case where, if the plane cannot be saved, he is ejected. An example of a dropped task is fuel optimization when it must be leveled urgently.

This scheduling approach is implemented in two parts:

1. **Pre-Scheduler** prepares a scenario.

2. **Dispatcher** activates tasks according to the scenario. It resembles the static scheduler, it has nothing to decide.

Reservation Scheduling Types

There are two types of reservation schedulers for both the scheduler and the dispatcher. We're considering the scheduler first.

The first variant has the pre-scheduler simply check that the timing constraints can be satisficed and accepts or rejects tasks. The dispatcher works like a classical dynamic scheduler. This introduces the risk that the fact that there is one schedule that meets all timing requirements does not mean that all possible schedule do. The dispatcher could still make the wrong choice.

The second variant has the pre-scheduler build itself a schedule that it has proved to meet the requirements. The dispatcher then executes it without changing anything. This variant is considered to be the best.

The dispatcher can be preemptive or non preemptive. In the first case the pre-scheduler must decide when the dispatcher will suspend each running task. The problem here is that there must be a way to make sure a long task is not cut at the wrong moment. The solution is making use of **critical sections**. The scheduler then knows not to cut the task at that time.

When non preemptive scheduling is used no task can have a shorter guaranteed response time than the duration of the longest task.

In conclusion there are 3 acceptable solutions:

- **Non preemptive with short tasks**. It's up to the programmer to decide where to cut.

- **Preemptive with a scheduler that knows were not to cut**. The programmer must indicate these sections.

- **Preemptive with short critical sections**. Sections are so short that the chance of them being cut is negligible.

Assessment of Reservation Schedulers

Reservation schedulers take into account both urgency and criticality of a task. A classic dynamic scheduler will always select the most urgent task, but this is not always the best choice. Reservation schedulers make their decision earlier so they can balance their decision based on all known factors. They can decide to sacrifice a an urgent task for a more important one.

Why don't all real-time systems use reservation schedulers? This is because if the system has the necessary resources to never miss a deadline, a classic scheduler is able to do the job. However, since more and more real-time systems are computer controlled, overdimensioning the controlling computers is not always a good idea, especially not when they are battery operated. Reservation schedulers may in that case be the better choice.

4.2 Example: MicroC/OS-II

Information presented in this section is based on the book of Jean Labrosse, MicroC/OS-II, The Real-Time Kernel, Second Edition, ISBN-13:978-1-57820-103-7. The presentation is different.

MicroC/OS-II is a commercial product of Micrium. However, higher education institutions may use, without charge for teaching purposes, as well the sources of this software as the executable programs themselves.

In MicroC/OS-II, each task has a different priority (a value between 4 and 60, 4 being the highest and 60 the lowest) and the scheduler is a simple dispatcher that will always launch the highest priority task. This may look rather primitive, but it is actually rather versatile because the priority of tasks may be changed dynamically. MicroC/OS-II uses that feature internally to avoid priority inversion, but it could also be used to give a task the role of pre-scheduler. However, using the same value to identify a task and his priority can be a nuisance to handle if the priority changes relatively often.

The operating system is simply linked with the application: the main() function of the program is the OS; the code of application tasks is implemented as C functions called by this main().

```
1  /* declare global variables: */
2
3  /* stack areas of the 3 tasks */
4  OS_STK task1Stk[1024];    // OS_STK has usually the size of a word (e.g. 16 or 32 bits
5  OS_STK task2Stk[1024];
6  OS_STK task3Stk[1024];
7
8  /*private static variables of tasks that need it*/
9  structure t2data{          // variables used by several functions called in task2
10    char message[20];
11 NT16U t2counter;
12 }task2data;
13
14 /*events: mutexes, interrupts etc.*/
15 OS_EVENT mymutex;
```

```
16
17
18  void main (void){  // main is the "system layer" of the project
19   INT8U err;       // a variable declared as 8 bit unsigned integer
20   OSInit();           // first initialise uCOS/OS-2 general purpose data structures;
21           // also creates the idle task(lowest priority) and a task
22           // collecting statistical data on CPU usage
23
24   /*initialise uCOS/OS-2 specific data structures used by this project*/
25   mymutex = OSMutexCreate(9,&err);  // first arg is pip (priority inheritance priority
         :
26                   // must be higher than prio of all tasks using the mutex)
27
28   OSTaskCreate(task1,      // pointer to function including code of task
29        (void *)0,   // pointer to task private data area: none on this case
30      &task1Stk[1023], // Top of stack (stack assumed to grow downwards)
31      10);           // prio of task: is also task identifier for uCOS/OS-2
32
33   OSTaskCreate(task2,
34        &task2data,  // pointer to task private data area
35      &task2Stk[1023],
36      15);
37
38   OSTaskCreate(task3,
39        (void *)0,
40      &task3Stk[1023], // Top of stack (stack assumed to grow downwards)*/
41      20);
42   /*More tasks may be created within the tasks themselves, but at least one must have
         beeen created here*/
43
44   /*start scheduling the tasks*/
45   OSStart();
46  }
47
48
49  void task1 (void *pdata){
50   INT8U err;
51   while (1){
52      /*some code here*/
53      OSMutexPend(mymutex,    // sleep until I get the mutex
54              0,        // time-out in clock ticks 0(for ever or 1 to 65535)
55        &err);
56      /*access shared resource*/
57      OSMutexPost(mymutex);  // allow next task to use the resource
58      /*more code here */
59   }
60  }
61
62  /*similar code for tasks 2 and 3*/
```

code/microcosii.c

The private memory areas of each task must be declared as a global static variable, arrays are used to store the stack of tasks. Variables shared by several tasks (there is none in this example) must also be declared as global static variables. The EVENT structures associated with events that tasks can be waiting for must be declared as global static variables too:

- Interrupts
- Mutex and semaphores,
- Task synchronization

The `main()` function initializes the operating system and the data structures used and one or several tasks. Each task can launch other tasks. The last operation of `main()` is to start the scheduler. Initializing a task means associate with that task:

- The function can execute the tasks.
- A pointer to the data structure of the task (where static variables are stored)
- A pointer to the top of the task's stack
- The initial priority of the task

All this is performed by the function `OSTaskCreate()`. The rest of the program consists of the functions corresponding to the tasks and of other functions.

4.2.1 Tasks in MicroC/OS-II

The code of the functions inside the body of a task in MicroC/OS-II must respect two constraints:

1. **It may never reach the final bracket of the function nor call `return()`.** Indeed, each tasks has its own stack, and the function that is the body of a task is at the absolute top of this stack: there is nothing above, so, it is impossible to return to a calling program. A way to satisfy this requirement is to include in this function, possibly after an initialization phase an infinite loop. Another option is calling a self destruction function when the task has finished its job `OSTaskDel(OS_PRIO_SELF);`.

2. **It may not run continuously.** Since MicroC/OS-II is preemptive, if a task could run continuously, no task of lower priority would ever be allowed to run.

If you want a task to be allowed to run as often as possible you could give it the lowest priority. If the application must include several such tasks, a function should be written for each one and the lowest priority task might be an infinite loop calling all these functions. However, it should be noted that it is rarely useful to let a task run continuously. Running the function more frequently than needed therefore is wasted activity. It is often sufficient to wake the task up a certain amount of times per second and let it go back to sleep afterwards.

Task States in MicroC/OS-II

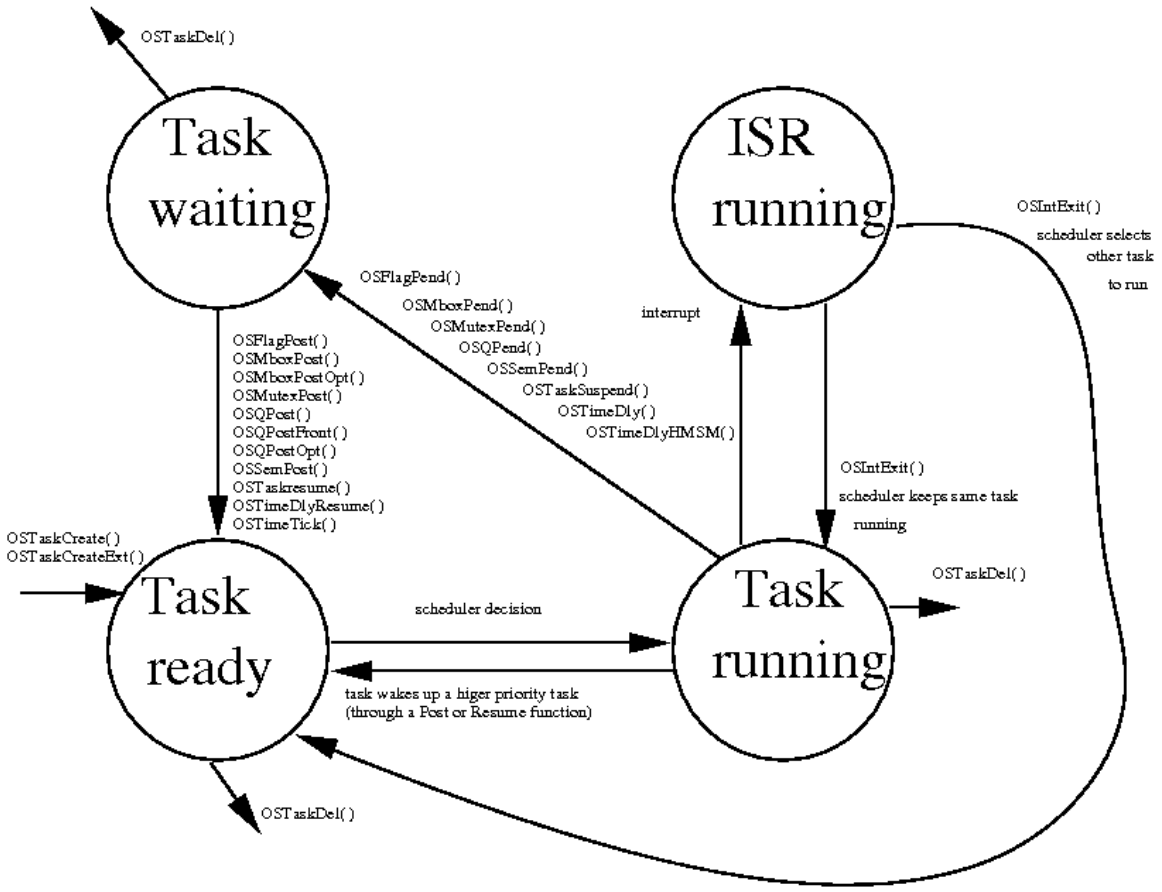

Figure 4.1: MicroC/OS-II States

A task is in either of the task states shown in figure 4.1.

- When the task is created, the task is **ready** to run.
- When the task is selected by the scheduler it's in the **running** state.
  - The task can be suspended by an interrupt.
  - The task can switch to the **waiting** state because it's waiting or suspended by another task.
- When the task is suspended due to an interrupt, the task switches to the **ISR** state. An *Interrupt Service Routine* is a sequence of assembly language instructions that saves the processors registers on the stack and calls a C function or runs assembly code that:
  - Warns the OS that an ISR starts by calling `OSIntEnter();`.
  - Acknowledges the interrupt request on the interface to the interrupting device in order to clear the request.

- May reset the interrupt mechanism, that was temporarily deactivated by the hardware at the start of the ISR.
- Performs the appropriate processing according to the event announced by the interrupt.
- Warns the OS when it has finished by calling `OSIntExit();`. This function verifies whether there was a higher priority task waiting. If this is the case the processor proceeds with this new task and puts the current one in the **ready** state.

The Main Task Management Functions

The tasks of a real-time application generally require to cooperate with each other. In order to do this in an organized manner, we need two techniques: **mutual exclusion** to gain exclusive access to a shared resource and a **rendez-vous** or **barrier** to meet each other after each task has completed.

Mutual Exclusion The most useful service offered by MicroC/OS-II is the mutual exclusion **mutex** which is associated with each shared resource. When a task wants to obtain a mutex that is not free, it waits. It's also possible to check the availability of the mutex without moving to the waiting state. Priority inheritance is supported.

Barrier Event Flags organize the *rendez-vous* between tasks. They are sets of bits (8/16/32) that may be set to 0 or 1 by tasks or ISRs. Tasks wait for a given pattern of bits to appear. Because this mechanism is so general it allows for complex conditions.

4.3 Peripherals and IO

System services will be as simple as possible. The more complex the system is, the harder it is to guarantee its constraints. This approach is called the **Reduced Instruction Set Computing** (RISC) Philosophy.

From a technical point of view interrupts routines are short to avoid interference with scheduling. Disk management, if any, tries to avoid disk access in actions that must be performed before a hard deadline. Contiguous files have sometimes been recommended for real-time systems. Their inconvenience however, exceed their advantages: each one must be created with the maximum size (which is a waste of space) and there is no advantage if blocks are read one by one.

4.3.1 I/O with MicroC/OS-II

MicroC/OS-II is a portable operating system. It consists of architecture independent and architecture dependent functions. Porting MicroC/OS-II means rewriting the latters; these are generally not visible by the application programs: they are underneath the architecture independent functions.

I/O functions are the exception: they are architecture dependent but visible by application programs. They are not part of the MicroC/OS-II distribution because they can differ as well by their interface as by their implementation in each application. They must thus be written by the application developper.

4.4 Communication

There are 3 ways to organize inter-process communications:

- **Shared Data**: requires a mutex to guarantee data integrity.
- **Messages**: require a lot of overhead.
- **Unstructured Flows**: requires correct interpretation.

The communication model should not be mistaken for its implementation. The message system can be implemented for example using shared memory.

The message model is easy to implement and has the advantage that it is asynchronous: the logistics are handled by the system, thus no need of mutex like for shared data.

4.4.1 Communication with MicroC/OS-II

MicroC/OS-II functions allow 3 types of task communications, all based on shared data structures but corresponding either to a shared data model or a message model.

Communication by Shared Data

Shared data structures are declared, for instance, as static global variables. To avoid interference between tasks, mutexes can be used to reserve exclusive access when needed.

CMailbox Communication

MicroC/OS-II provides a service called **mbox**, allowing to transfer a pointer to a data structure from one task to another. They allow transferring the permission to access the pointed data structure from one task to another.

Communication by messages queue

The mboxes allow synchronizing a task on arrival of data produced by another task. However, because only 1 message at a time can be in a mbox, the sender (producer) can run ahead of the receiver (consumer). This is a common problem in a data processing system. Message queues are a solution to this problem. Instead of a pointer to a message, they include a pointer to an array of pointers to messages. Like in any queue, the first message in the queue is the first message received.

4.5 Time Management and Synchronization

We expect a good clock to be **measurable**, **chronoscopic** (increasing monotonously and without discontinuities), to have a known precision and to be fault tolerant. In a centralised system, there is no real problem if the quartz is sufficiently precise and stable. A clock can be derived from this and all times can be measured with this clock.

4.5.1 Time Management with MicroC/OS-II

MicroC/OS-II includes a few time related functions based on regular interrupts of a timer. These interrupts are called **ticks**. At each tick, a 32 bit counter is incremented. If ticks happen every millisecond, there will be 85.410^6 a day. This counter will reset to 0 after a little less than 50 days.

4.5.2 Clocks in Distributed Systems

Computer clocks are based on quartz oscillators that provide a stable frequency. However, although stable, the frequencies of different quartz oscillators are never identical. In order to synchronize clocks, a mean time must be defined.

Clock adjustments must be progressive: increase the rate of slow clocks and decrease that of fast clock. Setting the time is no option since that would invalidate the chronoscopic property. Doing this might require a special electrical circuit.

Fault Tolerant Average

The **Fault Tolerant Average** (FTA) Algorithm is designed to synchronize clocks in a distributed system. There are two variants. The first ignores communication delays between systems, the second doesn't.

When communication delays are ignored, each system broadcasts a message announcing its local time. Each system deduces the time-lag between its own clock and the others. The k most different values from the mean are ignored. The correction to apply is the mean value of the time lags that have not been rejected.The correction is applied progressively, by slightly modifying the clock rate for a limited amount of time.

$$C_i = \frac{1}{N-k} \sum_{j=1}^{N-k} D_j i$$

Example 4.5.1. If $N = 2$ and $k = 0$ then

$$C_1 = \frac{D}{2} \quad C_2 = -\frac{D}{2}$$

The second variant does take communication delays into account. These delays trouble the correct measurement between the clocks: the time lag needs to be eliminated before calculating the mean time. We consider two cases: in the first one the delay is **symmetrical** and **predictable**. The second assumes the delay is predictable but **not symmetric**.

Symmetrical and Predictable Delay We consider two systems: A and B. System A sends a message to B including the time, measured in A, when the message was sent: $t_{a,0}$. Now B receives the message at time $t_{b,1}$. With $D_{b,a}$ the time lag and d_t the communication delay:

$$t_{b,1} - ta, 0 = D_{b,a} + d_t = \Delta_l$$

In the same way, B sends a message including its transmission time $T_{b,0}$.

$$t_{a,1'} - t_{b,1'} = D_{a,b} + d_t = \Delta_2$$

No A can compute

$$\Delta_l + \Delta_2 = (D_{b,a} + d_t) + (D_{a,b} + d_t)$$

Because $D_{b,a} = -D_{a,b}$ we know

$$d_1 = \frac{\Delta_l + \Delta_2}{2}$$

Ethernet is symmetrical but unpredictable because of collisions. So, the previous method could be applied if there were no collisions. Fortunately collision effects can easily be eliminated. The systems have to perform a few successive measurements of the time-lag instead of a single one. If the network is not overloaded, several results will be identical to the minimal value, which is the correct value.

Non Symmetrical Predictable Delay Now we have to find the relation between the two communication delays.

$$d_1(a \to b) = d_E + d_{t,ab} + d_R$$

Here is

- d_E the processing delay at the sender
- d_{rab} the network transmission delay
- d_R the processing delay at the receiver

Using a second measurement from A to A yields

$$d_p(a \to a) = d_E + d_{t,ab} + d_{t,ba} + d_R$$

$$\Delta_l + \Delta_2 = 2d_E + 2d_R + d_{t,ab} + d_{t,ba}$$

$$\Delta_{A \to A} = d_E + d_R + d_{t,ab} + d_{t,ba}$$

According to the relative positions of the stations, one knows the relation between $d_{r,ab}$ and $d_{r,ba}$.

Clock Resolution

The following symbols are used:

- Δ: accuracy of the synchronization
- g: clock resolution

It's necessary that $|\Delta| \leq g \leq |2\Delta|$:

- A higher resolution is not significant: it amounts only to adding insignificant decimals.
- A coarser resolution does not make fully use of the synchronization of the clocks.

In these conditions, one can identify the order of two events occurring on different sites, provided that they are at least two clock ticks apart.

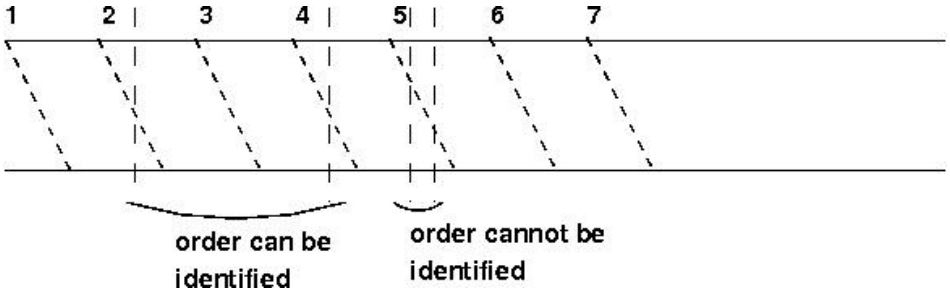

Figure 4.2: Clock Resolution

Chapter 5

Fault Tolerance

5.1 Fault Classification

Faults can either be due to material failure or human failure.

- **Material Faults**
  - Permament (broken optical fiber)
  - Transient (cosmic ray changes value bit in memory)
  - Internal Fault (overheating due to fan malfunction)
  - Environment Fault (power failure with no UPS (Uninterruptible Power Suply))
- **Human Faults**
  - Design Errors (analysis or programming error)
  - Interaction Error (wrong action by user or wrong data received from sensor)

Errors in data processing:

	Data NOK	Data OK
Data Rejected	n/a	Error 1
Faulty Result	Error 2	Error 3
System Crash	Error 4	Error 5

Errors 4 and 5 are **Fail Stop** errors. They're easy to detect because in case of a problem everything stops. Error 2 and 3 are more vicious, the system continues to work but not as intended.

5.2 Reliability Metrics

Reliability is measured by looking at what happens to a set of similar items. With N the amount of items:

- $S(t)$ is the number of surviving items after time t.
- $F(t)$ is the number of faulty items after time t.

Now we can define the **reliability** of this kind of items as

$$R(t) = \frac{S(T)}{N}$$

We define the **failure rate** as:

$$\text{Failure Rate} = \frac{1}{S(t)} \frac{dF(t)}{dt}$$

We define the **Mean Time Between Failures** (MTBF) as:

$$\text{MTBF} = \int_0^\infty R(T)dt$$

R is a dimensionless value representing time. The MTBF is the surface under the curve representing the evolution with respect to $R(t)$, which is the percentage of surviving items after t.

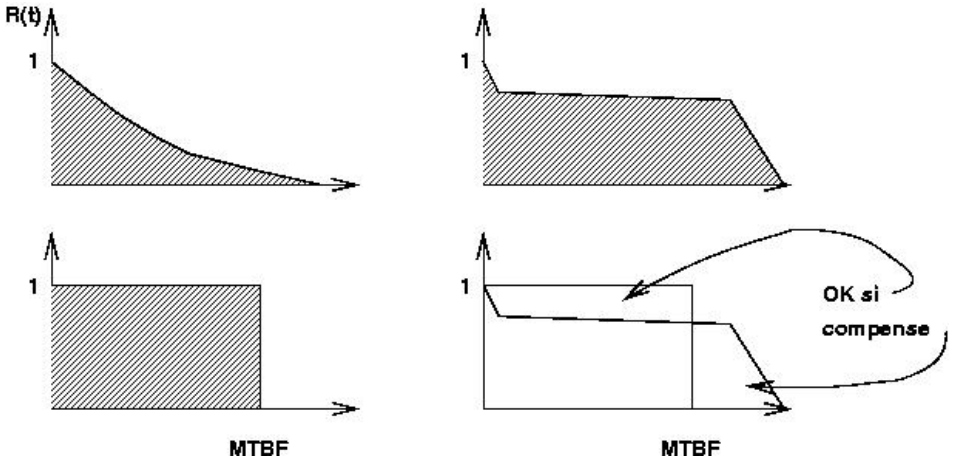

Figure 5.1: MTBF Illustration

Consider the two items on the right. After an initial drop corresponding to manufacturing defects R(t) stays nearly constant for a rather log time. This is the useful life time of the items. Extending the guarantee trough this time span is not expensive but has a psychological effect on clients. (this is why several manufacturers give long guarantees) Later on, wear begins to show and the number of surviving items drops. This curve shows that the MTBF will often be before the end of the plateau. It is therefore a good measurement of the time an item can be expected to work.

However, the MTBF cannot always be considered equal to the useful life of an item. For some devices, two values are published: the MTBF and the lifetime. In this case the computation of the MTBF is obtained by linear interpolation of R(t) at the beginning of the lifetime.

Example 5.2.1. An MTBF over 500.000 hours are given for hard disks along with a lifetime of 5 years. Knowing that a year includes 8760 hours, it means a MTBF of 57 years. In that case, the MTBF tells only that, after the time of early failures less that about 10% of the disk might have a problem during their useful life. Disks intended for servers often have a guarantee equal to their lifetime.

Other useful metrics are the **availability** $A(t)$ and the **Mean Time To Repair** (MTTR) :

$$A(t) = \frac{T_{\text{on}}}{T_{\text{on}} + T_{\text{off}}}$$

$$A(t) = \frac{T_{\text{on}}}{T_{\text{on}} + (N_{\text{faults}} \times \text{MTTR})}$$

$$N_{\text{faults}} = \frac{T_{textON}}{\text{MTBF}}$$

$$A(t) = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

5.3 Doubled Systems

The purpose of doubled systems is to **minimize the MTTR**. There are multiple ways to implement doubled systems.

5.3.1 Standby Redundancy

With standby redundancy a spare system must be started. This might take up to 15 or 30 minutes before becoming fully operational. Improvements are possible to limit reconfiguration time. (= disconnecting equipment from one system and connecting it to another) Examples are shared peripherals between the main and spare system or the use of dual port peripherals.

5.3.2 Active Redundancy

The problem with active redundancy is threefold:

- Decision to switch between main and spare
- Synchronization between main processor and spare processor
- Restart the processor that halted

Basic Solution

A health report consists or a regular message indicating that the system is still alive, assuming the system is a **fail/stop system**. An alternative might be a signature based on significant components of the system. This signature may be created at regular intervals or at each significant state change. Synchronization is handled by the events occurring at the controller system. A watchdog decides to switch to the spare processor.

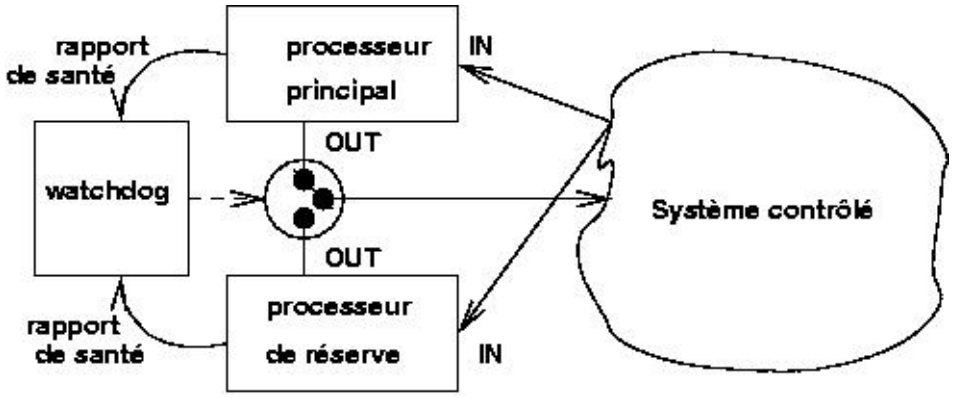

Figure 5.2: Active Redundancy

A restart is handled by copying the state of the current main processor to the spare one. During this, the main processor might be stopped. If not, delicate techniques are used to synchronize distributed systems.

It's important that the watchdog doesn't fail! Only if it fails without disrupting the system's mission it is simple and robust.

TANDEM Solution

This implementation does not feature a watchdog. It manages 2 to 16 autonomous processors with each

- Their own power supply.
- Their own copy of the OS.
- Well informed of the state of each other.
- Processors interconnected by two busses.
- Their own I/O channel.
- Peripherals controller connected to two channels.
- Monitored by a spy circuit that informs a control processor and the operator of a detected problem.

Synchronization is managed on software level using **checkpointing**. All software has been designed to run in the TANDEM fault tolerant environment. For each running process P_A there is a shadow process P_B on another processor. On a regular basis P_A sends its state to P_B. This is done using a helper toolbox but introduces 15% to 30% overhead.

The decision to switch is also made at software level. The operating system of each processor is aware of the state of all others. If the operating system of a processor X discovers that the operating system of another Y is dead and if the P_A of a P_B process running on X was running on Y then X activates the P_B that replaces P_A.

Pair & Spare

Each processing module is thus duplicated by a *spare*. Main and spare processor are synchronized by running the same program with the same data at the same time. Each module consists of two processing units and the results are permanently compared. This is method is called **bus snooping** and the concept is called a **self checking pair**. Time is synchronized by a central clock. The decision to switch is made on hardware level when the results are not identical. If it's the main pair, the system switches to the pair. It's the system itself that warns the maintenance service.

The module that has been repaired or replaced is synchronized without stopping the active module (copy of the memory contents, etc.) under control of the OS. There can be up to 16 pairs.

5.4 Multiple Systems

The principle of doubled systems is to transfer the activity as quickly as possible from a failing main system to a working spare. The idea of multiple systems is to have enough running resources to let them have a **vote**. This mean there must be at least 3 modules, and the system should have a **fail/stop** behavior.

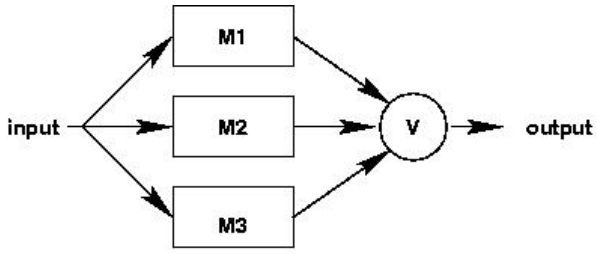

Figure 5.3: 3 Modules

Triple Modular Redundancy or **TMR** is a particular case for M and N. We assume the the system consisting of N components survives if at least M components survive.

5.4.1 Survival Probability

In order to calculate the **probability of survival** we have to make some assumptions:

- Malfunctions of the different components are independent.
- R is the survival probability of a single module alone.
- R_n is the probability that a single module fails.

The probability that one of the modules fails is

$$R^{n-1} \cdot (1 - R) \cdot C_n^{n-1}$$

The number of possible configurations where exactly one module fails is the number of possible combination of taking $(n-1)$ items out of N items.

$$C_n^{n-1} = \frac{n!}{1!(n-1)!} = n$$

The probability that exactly $0 \leq k \leq n-m$ modules fail simultaneously

$$R^{n-k} \cdot (1-R)^k \cdot C_n^{n-k}$$

The probability that at least m survive

$$\sum_{k=0}^{n-m} R^{n-k} \cdot (1-R)^k \cdot \frac{n!}{k!(n-k!)}$$

If the probability of survival of the voting device is R_v, the probability of survival of the system is

$$R_v \cdot \left(\sum_{k=0}^{n-m} R^{n-k} \cdot (1-R)^k \cdot \frac{n!}{k!(n-k!)} \right)$$

In case of TMR this becomes

$$\begin{aligned} R_{\text{TMR}} &= R_v \cdot \left(\sum_{k=0}^{1} R^{3-k} \cdot (1-R)^k \cdot \frac{3!}{k!(3-k!)} \right) \\ &= \left(R^3 + R^2 \cdot (1-R) \cdot 3 \right) \\ &= \left(3R^2 - 2R^3 \right) \end{aligned}$$

We consider the case where $R_{\text{TMR}} > R$. Assuming that $R_v = 1$ and $0 < R < 1$. This means

$$3R^2 - 2R^3 > R \rightarrow 3R - 2R^2 > 1$$

This equation has two solutions: $R = 1$ and $R = \frac{1}{2}$. This means R_{TMR} will be preferable to R as long as $R > \frac{1}{2}$.

Example 5.4.1. If R decreases exponentially in time as shown on figure 5.4 below, TMR improves the reliability of the system a the start but does not extend its useful life.

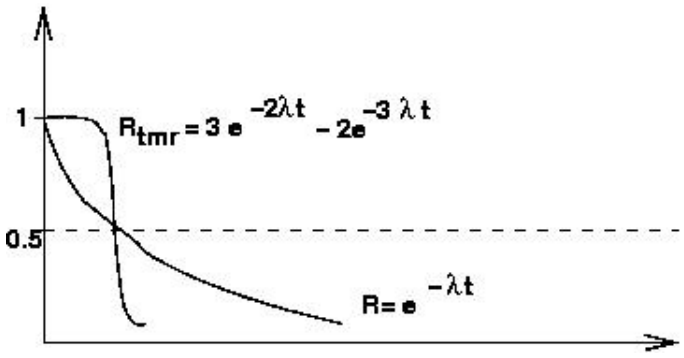

Figure 5.4: Lifetime with TMR

Part II

Asynchronous State Graphs

Chapter 6

ASG Language

ASG is a graphical specification language for distributed real-time systems and belongs to the family of hierarchical state languages such as **statecharts**, **Timed Transition Models** (TTM) and **modecharts**. All these languages have been introduced progressively from 1987. They allow to describe the behavior of systems and have enhanced the expressiveness of state machines by introduction of time, hierarchies and parallelism. This avoids the explosion of the number of states and allows reasoning at different abstraction levels. It was intended to be readable by non computer-scientists, in order to be able to verify that what is specified is actually what the client wanted.

ASG, which was developed by UCLouvain, has some specific characteristics.

- It takes into account the impossibility to detect instantaneously a state change in a distributed system if it happens remotely (transmission delay).

- It offers more restrictive rules on the destination state of a transition, e.g. it is not allowed to go to any other state that the initial sub-state of a state refined in sub-states if not coming from another sub-state of the same refinement. This allows to reason at a given hierarchical level without considering what happens at lower levels. Consequently this promotes a top-down design, by refining states into sub-states when appropriate.

6.1 Language Elements

An ASG diagram consists of **states** and **transitions**. Crossing a transition is instantaneous: the system is thus always in a state.

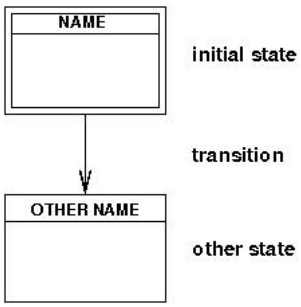

Figure 6.1: ASG States and Transitions

6.1.1 States

Each state has a name and some **time-related state properties:**

- **Minimum Visit Time** (MINVT) : Is the minimum time the system remains in the state.
- **Time-Out**: A time-out counter is started when the state is entered. When this time-out duration has elapsed, a boolean variable `state_name.timout` becomes true if the system is still in that state.

A state has an **action** which is executed when the state is entered. When the action is finished, the system might choose to wait in the current state. Executing the action takes some time and can be seen as happening in parallel with the system staying in the state. An action consists of 3 steps:

1. Read the necessary data
2. Process the data
3. Write the results atomically

An action has one property: its **deadline**, the maximum amount of time the action is allowed to take. The deadline is a **declarative temporal specification**: the system should be fast enough to respect that constraint. However, since the system assumes that deadlines are always respected, it's ignored by the system and a constraint for the implementer. On the other hand, **MINVT** and **time-outs** are elements that the system's behavior has to take into account.

Further properties of a state include a **rendez-vous label**. This is used to synchronize parallel behavior. Another property is a list of **resources** that prevents parallel behavior to reach incompatible states.

6.1.2 Transitions

Definition 6.1.1. An **event is detected** when the system learns the following in the origin state:

The system is in the origin state of the transition **and** the condition associated with the transition is true.

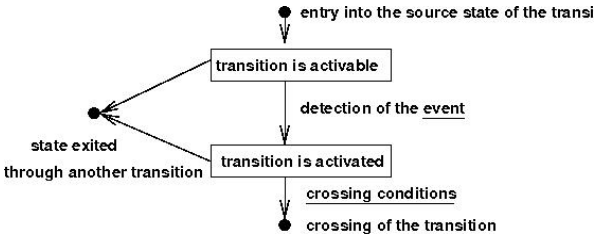

Figure 6.2: ASG Transitions

The condition is any boolean expression, based on environment variables or booleans indicating a *rendez-vous* or *time-out*. In a distributed system some time is needed in order to detect a state change of a remote variable. To avoid **hidden delays** the system works with local state changes and the local perception of the remote state.

Figure 6.3: ASG Regular and Abort Transition

When a transition event is detected, the action is not necessarily finished, the time spent in the state is not necessarily larger that MINVT and resources required to enter the destination state are not necessarily available. This gives cause to two families of transitions. The **crossing condition** depends on the transition family.

1. The crossing conditions of **regular transitions** (figure 6.3, left) are the following:
   - The action is finished.
   - The time spent in the state $/geq$ MINVT.
   - All resources are available.
   - No transition with a higher priority is active.
2. The crossing conditions of **abort transitions** (figure 6.3, right) is limited to the absence of a higher priority transition allowed to leave its state.

6.2 Diagram Structure

As with similar languages, there are two ways to structure the diagrams: **parallelism** and **hierarchy**.

6.2.1 Hierarchy

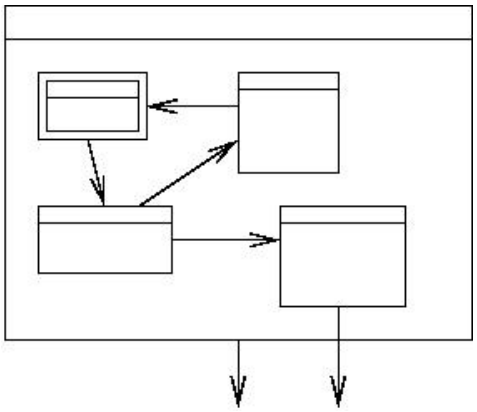

Figure 6.4: ASG Hierarchy

Any state may be refined in mutually exclusive sub-states. Any such sub-state is a state that may itself be refined. The system is both in the outer state as well as in the sub-state. Actions may be associated with the outer state and with each sub-state, at any level, including the initial sub-states. Actions attached to the outer state and to the initial sub-state start simultaneously and are performed in parallel.

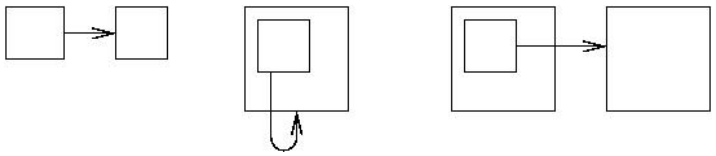

Figure 6.5: ASG Hierarchic State Leaving

A transition leaving a state may only end in a sibling state (a state belonging to the same refinement), an ancestor state (a state enclosing the current state) or the sibling of an ancestor. A state can be exited trough a transition leaving this state itself, leaving a state enclosing this state or leaving a state included in this state.

Example 6.2.1. State A could be exited trough transitions i, j and k.

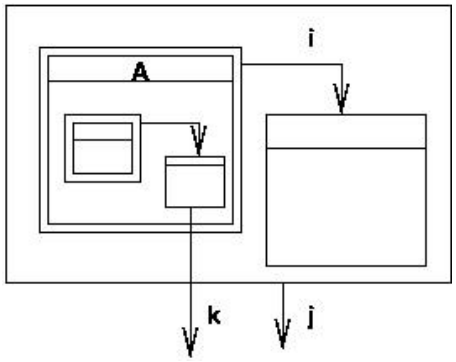

Figure 6.6: ASG Example Leaving State

Conditions of a transition must be satisfied for all the states that would be exited. (The system might have to wait for several actions to be completed and several MINVT to be elapsed)

6.2.2 Parallelism

In ASG, parallelism is always explicit by refining the state into several parallel components. Note that exiting a state through 2 transition at a time is not allowed in ASG.

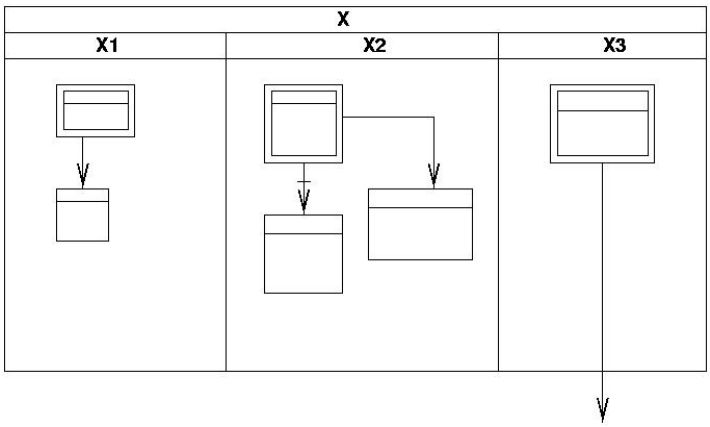

Figure 6.7: ASG Parallelism

When a state, refined in parallel components is entered, the initial sub-states of all the parallel components are entered at the same time. If the refined state is exited from one of the parallel components, all the parallel components are terminated. (Exiting the refined state may be through a regular or trough an abort transition)

6.3 ASG Components

6.3.1 Transition Priority Rules

One of the **crossability conditions** of a transition is to have the highest priority of all transitions conflicting with it. Transitions are called conflicting when they are activated and the

highest state (in hierarchy) exiting is the same, or an ancestor, of the other state.

Priority Rules:

1. An abort transition has a higher priority than a regular transition. If the state can be exited through several abort transition, one is selected according to the following rules.
2. If the outermost state exited by an abort transition encloses that of another abort transition, the former will have a higher priority.
3. Numbered transitions with a lower number have a higher priority. Unnumbered transitions have the lowest priority.
4. A transition activated before another has a higher priority than the latter.
5. If none of the above rules apply, a transition is selected at random.

6.3.2 The Rendez-Vous

Rendez-vous labels are associated with states. A **rendez-vous** happens when all the parallel components active and concerned by the same rendez-vous label are in a state labeled with that same rendez-vous. An active parallel component means that the system is in one of its states. A state is concerned with a rendez-vous label if at least one of its states is labeled with it.

A rendez-vous allows many types of synchronizations between parallel components. The simplest is to trigger a transition in a parallel component when a job is finished in another one. Generally a label will be assigned to the state following the one with the action. If necessary, a empty state (without an action) can be added for this purpose. Another use of the rendez vous is allowing a parallel decomposition to exit in a disciplined way.

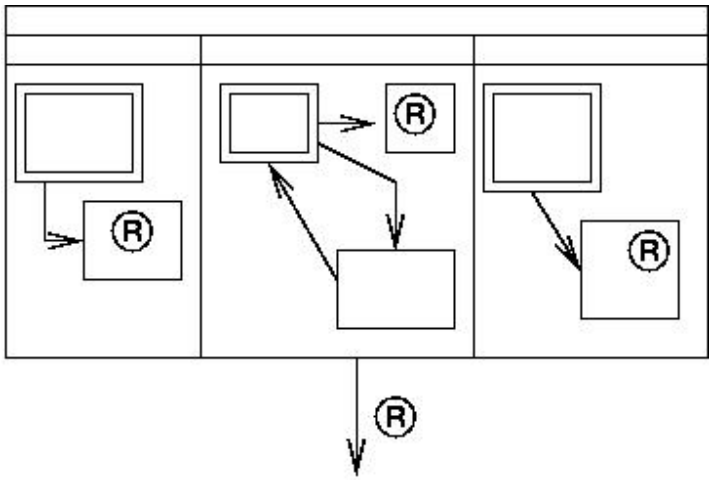

Figure 6.8: ASG Rendez-Vous

Nothing guarantees that two transitions (in different parallel components) will transition simultaneously, even if their event has the same rendez-vous point. This is because transition

conditions could be delayed and if the system is distributed, the event will not necessarily be perceived simultaneously everywhere.

In a distributed system, the detection of a rendez-vous is not necessarily immediate, but all rendez-vous will be detected, on condition that the time intervals in which the rendez-vous states are occupied overlap at least in part.

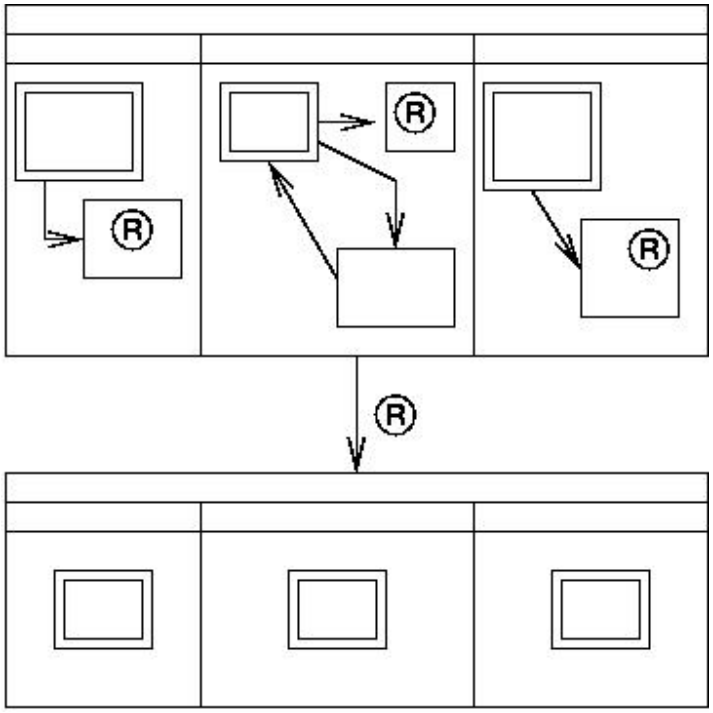

Figure 6.9: ASG Rendez-Vous

6.3.3 Resources

Resources allow the system to get temporarily exclusive access to a shared component. When resources are needed in a sub-state of some parallel component, the sub-states are labeled with their required resource.

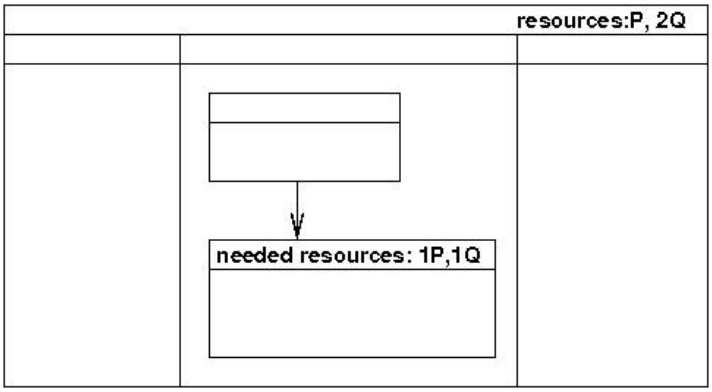

Figure 6.10: ASG Resources

A parallel component must have obtained the exclusive right to use of all the resources necessary in the next state before being allowed to transition to this state.
Resources are requested by a parallel component as soon as the transition to enter this state is activated. Resources are held until either a higher priority transition (through which the source state could exit) is activated or until the state that needed the resource is exited.
Obtaining requested resources may take some time. The system might have to wait until the resources are freed by the previous user, especially when several resources are needed to enter a state. In order to avoid deadlocks when gathering resources needed to cross a transition, a parallel component must always obtain resources in the reverse order of their declaration

Example 6.3.1. The left component will cause a deadlock if it does not respect the last rule.

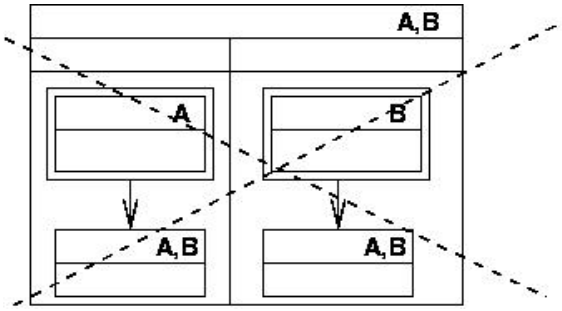

Figure 6.11: Deadlock

The aim of resources is to express mutual exclusion between states of different parallel components.

Example 6.3.2. Creating starvation situations with resources, making some states unaccessible is easy, as illustrated below.

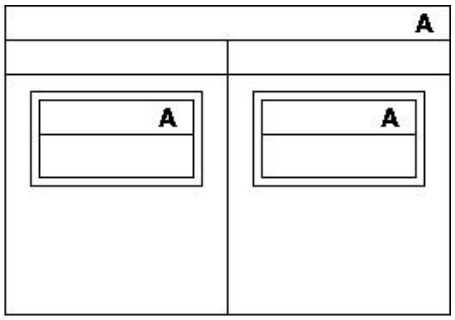

Figure 6.12: Deadlock

However, a static analysis of the diagram allows identifying the sets of mutually exclusive states and those that are not accessible.

6.4 Examples

Example 6.4.1. An elevator will only change direction if there are no more requests left in the current direction. The states *stopped*, *doors opening* and *doors opened* all have a **MINVT** to give the doors time to open and let people leave the elevator. The *open* state has a time-out after which the doors are closed. Consider the following activation conditions:

1. detection_door_completely_open
2. movement_requested or open.timeout or close_doors_button_pressed
3. presence_detected_in_door_opening
4. detection_door_fully_closed
5. always true
6. movement_request_in_current_direction
7. arrival_at_destination_floor or arrival_at_calling_floor_in_current_direction or (no_other_destination_ in_current_direction and arrival_at_calling_floor_in_other_direction)
8. movement requested_in_other_direction or called_from_other_direction
9. true

In order to be complete, one should add parallel components memorizing movement requests and a component to remove the request at each floor when the destination is reached.

Example 6.4.2. A sluice has two doors P_1 and P_2 and 5 sensors X, Y, Z, V, W to detect the presence of a ship. Sensors X, Y and Z are located in front of the doors and inside the sluice. Sensors V and W are located in such a way that they can detect the presence of the ship when passing the doors.

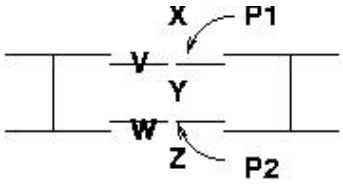

Figure 6.13: Sluice

Only one of the sluice's doors may be opened at a time. Outgoing traffic (from X to Z) always has priority.

Sequential Solution
The sequential solution simply expresses that there are two sequences that are allowed.

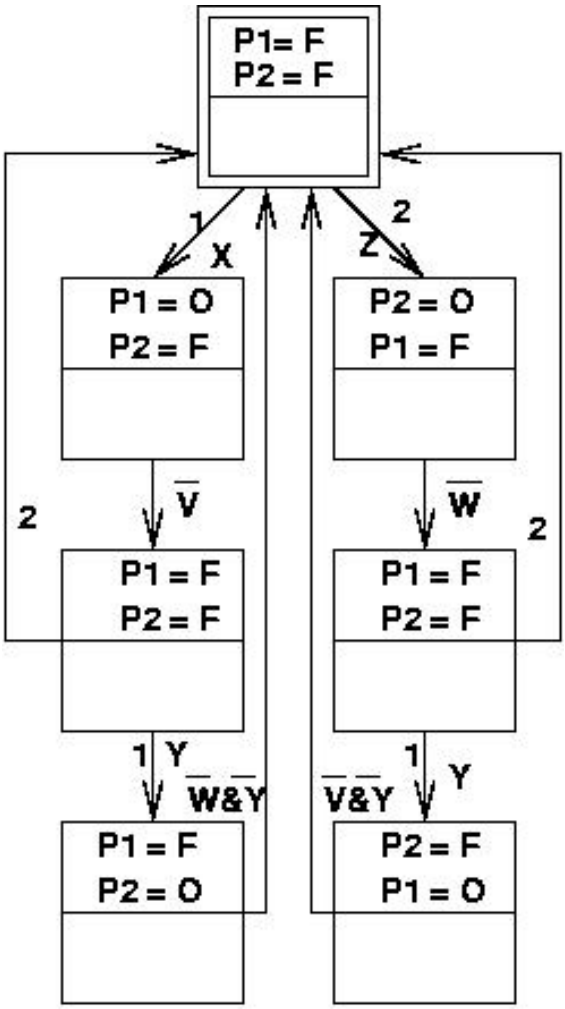

Figure 6.14: Sequential Solution

To be complete, each state should be refined in two sub-states: door sliding and door in position

Parallel Solution
A resource is used to prevent the two doors to open simultaneously.

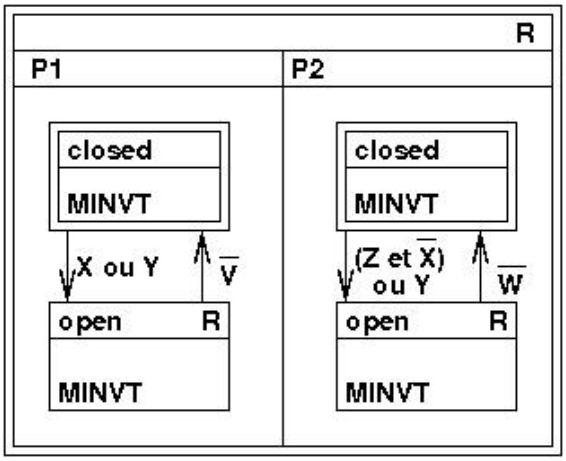

Figure 6.15: Parallel Solution

The purpose of the MINVT in the *open* states is to allow ships to enter or exit the sluice. Otherwise, the doors would just open and close without letting anybody in.

Chapter 7

Implementing ASG

In this chapter we have a look at how ASG can be implemented on different architectures.

7.1 Implementing ASG on a Naked Computer

The proposed implementation is intended for rather simple systems, a naked computer, running on a single processor. It can, of course be elaborated to suit more complex systems.

7.1.1 Scheduling Parallel Components

A naive way to schedule a set of tasks that may be appropriate in simple systems is the cooperative multitasking loop. This is simply having a main loop run with the different tasks in it. Depending on the relative frequency the task should run the task is called multiple times. This kind of scheduling is non preemptive meaning the OS will never interrupt a running task. Because of this each task should be as short as possible. The worse case response time will be the sum of the maximum execution times of the functions. A simple code example can be seen in 7.1 the comments there should explain what is being done.

```
1  enum states {
2    xxx,
3    xxx_W,
4    yyy
5  } state;
6  int stepAcounter;
7
8  void main(){
9    stepAcounter = 0;
10
11   // Outer state of the system is decomposed in parallel components
12   while(1){
13     taskB();
14     taskA();
15     taskB();
16   }
17 }
18
19 void taskA(){
20   //Do something depending on the state of A.
21   switch(state){
22     case xxx: //Action xxx should be performed.
23       //check abort of A.
```

```
24        if(!abortA){
25          switch(stepAcounter){
26            case 0:
27              subTaskA();
28              stepcounter++;
29              return;
30            case 1:
31              //...
32              stepcounter++;
33              return;
34            //..
35            case 3:
36                //...
37                state=xxx_W;
38                break;
39          }
40        } else {
41          return;
42        }
43  // NO break here since taskA goes directly to state xxx_W after finishing action xxx.
44      case yyy: //...
45
46      case xxx_W: //transition checking state.
47        //check abort of A.
48        if(!abortA){
49          if(checkTrasitionConditions()){
50            state=yyy;
51          }
52          // ...
53        } else {
54          return;
55        }
56        break;
57      default:
58        return;
59    }
60  }
61
62  void subTaskA(){
63    while(!endCondition){
64      taskC();
65      taskD();
66    }
67    //Leave parallel decomposition.
68    return;
69  }
```

Listing 7.1: Example of cooperative multitasking where task A is run twice as frequent as task B. Each task would represent an ASG (sub)state.

7.1.2 Implementing Rendez-vous

Implementing of the rendez-vous point is easy. The state enumeration variables should be external (outside a function). Assume that this is the condition to transition to state zzz in component t: The state of component u is aaa or bbb while the state of component v is qqq. Then the following can easily be checked in the function representing state t:

```
if (((state\_u==aaa)||(state\_u==bbb))&&(state\_v==qqq))state\_t=zzz;
```

7.1.3 Implementing Resource Management

Can easily be done with external boolean variables: TRUE means available, FALSE means BUSY. If there should only be transitioned to yyy if resource Y is available state=yyy (from 7.1) can be replaced with 7.2.

```
1    if(resource_R){
2      resource_R=FALSE;
3      state=yyy;
4    } // Don't forget to FREE the resource in state yyyy!!!!
5  }
```

Listing 7.2: Example of implementing resource management.

7.1.4 Handling MINVT & Timeouts

A software counter counting the amount of overflows in a hardware timer should be used. The hardware timer usually generates an interrupt when it overflows. The most significant bits reside in the software and the least significant in the hardware timer. Knowing the duration of a tick one can convert the duration of a minvt or timeout into ticks.
When entering a state with minvt or timeout the software time should be stored. Then the current time can be checked in the xxx_W state to see if the timeout is over or minvt has passed.

7.2 Implementating ASG in C on MicroC/OS-II

To implement an ASG diagram on top of MicroC/OS-II a couple of changes have to be made. We'll discuss them on the basis of the OS function calls. Instead of round robin scheduling, the OS uses **prioritized task scheduling**. Because the OS uses static priorities, only assign a higher priority to some task if what they do has a much shorter deadline than other tasks. Since the scheduling is preemptive be careful with shared resources since at any moment, any task can be suspended by a tasks with the higher priority.

- OSTaskCreate() All tasks should be created in the main with this function.
- OSTaskSuspend() Task can be temporairally suspended with this function. All tasks that do not need to run permanently should be put to sleep imediatly after being created.
- OSTaskResume() Used to resume a task that has been suspended.
- OSTaskFlagPost() Sets a certain flag to a certain value. Can be used to indicate that a resource is busy/free or a condition is satisfied.
- OSTaskFlagPend() Reads a certain flag and returns it value. This can be used to wait for a resource or to put the task to sleep till a transition condition is satisfied.
- OSTaskTimeDLY() Further execution of the task that calls this is temporairly suspended for the given time.
- OSTimeGet() Returns the current time.

7.2.1 Scheduling Parallel Components

With the functions above this is an easy task to accomplish.

7.2.2 Implementing Rendez-vous

With OSTaskFlagPost() conditions for the rendez-vous can be set and later read out with OSTaskFlagPend ().

7.2.3 Implementing Resource Management

This still has to be implemented with mutexes or semaphores.

7.2.4 Handling MINVT & Timeouts

If a state has a minvt, one must first get the value of the current time when entering the state (beginning of case xxx), using OSTimeGet(), then, when when the action is finished (beginning of case xxx_W one must again measure time with OSTimeGet() and if the minvt is not over, one must wait for the remaining time with OSTimeDLY().
Timeouts are not implemented by the OS. But it can easily be achieved in the application by creating a "timeout" task. By default this task is suspended if another tasks that needs a timeout, the temporized task, it starts the timeout task (with OSTaskResume()). The timeout task itself then creates a delay for the specified time. After that time it suspends itself and sets a flag to wake up the task it was waiting for. If the temporized task exists it's temporized state before the waiting is over it must notify the timeout task.