9
Case Studies

9.1 Real-Time Acquisition and Analysis of Rolling Mill Signals

9.1.1 Aluminium rolling mill

Manufacturing process of an aluminium reel

The Péchiney Rhénalu plant processes aluminium intended for the packaging market. The manufacturing process of an aluminium reel is made up of five main stages:

1. The founding eliminates scraps and impurities through heat and chemical processes, and prepares aluminium beds of 4 m $\times$ 6 m $\times$ 0.6 m weighing 8–10 tons.
2. Hot rolling reduces the metal thickness by deformation and annealing and transforms a bed into a metal belt 2.5–8 mm thick and wound on a reel.
3. Cold rolling reduces the metal down to 250 micrometres (μm).
4. The thermal and mechanical completion process allows modification of the mechanical properties of the belt and cutting it to the customer's order requirements.
5. Varnishing consists of putting a coat of varnish on the belts sold for tins, food packaging or decoration.

The packaging market (tinned beverages and food) requires sheets with a strict thickness margin and demands flexibility from the manufacturing process. Each rolling mill therefore has a signal acquisition and analysis system that allows real-time supervision of the manufacturing process.

Cold rolling

Mill L12 is a cold rolling mill, single cage with four rollers, non-reversible, and kerosene lubricated. Its function is to reduce the thickness of the incoming belt, which may be between 0.7 and 8 mm, and to produce an output belt between 0.25 and 4.5 mm thick, and with a maximum width of 2100 mm. The minimum required thickness margins are 5 μm around the nominal output value. The scheme of the rolling mill is given in Figure 9.1.

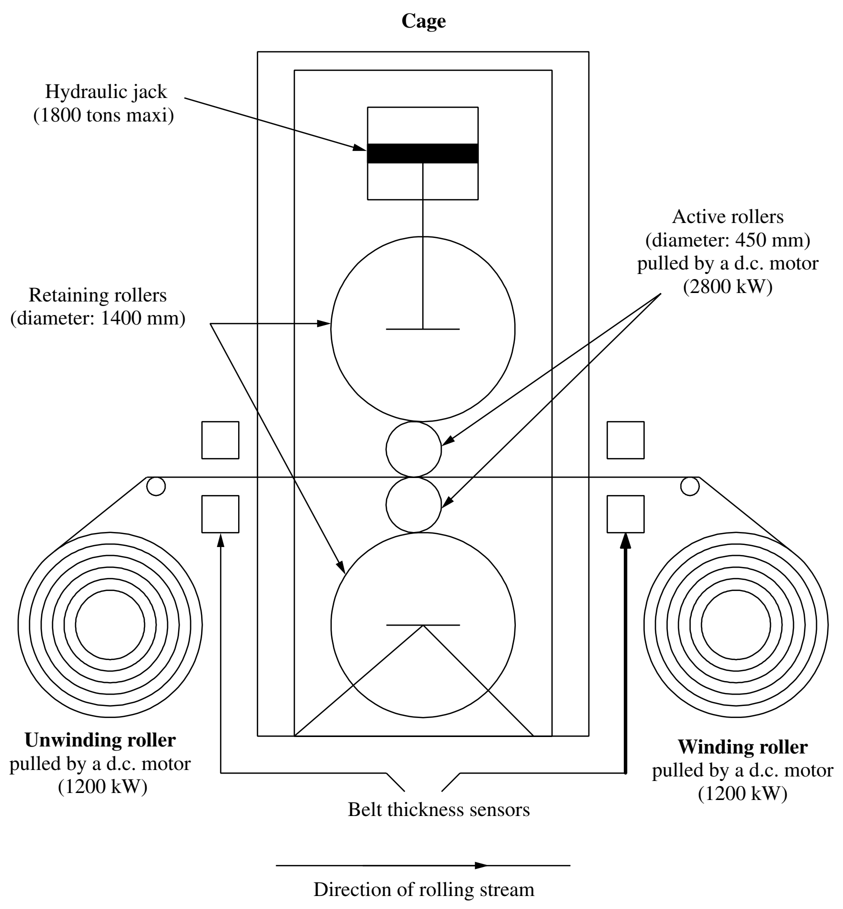
Cage
Hydraulic jack (1800 tons maxi)
Active rollers (diameter: 450 mm) pulled by a d.c. motor (2800 kW)
Retaining rollers (diameter: 1400 mm)
Unwinding roller pulled by a d.c. motor (1200 kW)
Winding roller pulled by a d.c. motor (1200 kW)
Belt thickness sensors
Direction of rolling stream

Figure 9.1 Scheme of the cold rolling mill

The thickness reduction is realized by the joint action of metal crushing between the rollers and belt traction. The belt output speed may reach 30 m/s (i.e. 108 km/h). The rolling mill is driven by several computer-control systems which control the tightening hydraulic jack and the motors driving the active rollers, the winding and unwinding rollers, the input thickness variation compensation, the output thickness control and the belt tension regulation. Three of the controlling computers share a common memory.

Other functions are also present:

- production management, which prepares the list of products and displays it to the operator;
- coordination of arriving products, initial setting of the rolling mill and preparation of a production report;
- rolling mill regulation, which includes the cage setting, the insertion of the input belt, the speed increase, and the automatic stopping of the rolling mill;

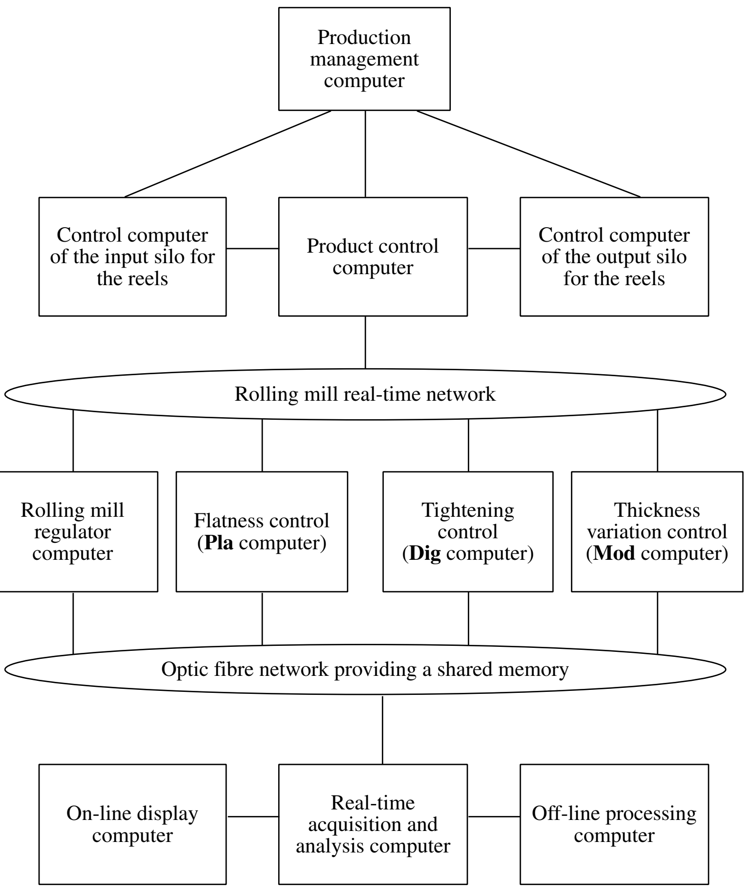
Production management computer
Control computer of the input silo for the reels
Product control computer
Control computer of the output silo for the reels
Rolling mill real-time network
Rolling mill regulator computer
Flatness control (**Pla** computer)
Tightening control (**Dig** computer)
Thickness variation control (**Mod** computer)
Optic fibre network providing a shared memory
On-line display computer
Real-time acquisition and analysis computer
Off-line processing computer

Figure 9.2 Physical architecture of the rolling mill environment

- management of two silos, automatic stores where the input reels and the output manufactured reels are stored.

Two human operators supervise the rolling mill input and output. The physical architecture of the whole application is given in Figure 9.2 where the production management computer, the control computers and their common memory, and the signal acquisition and analysis computer are displayed.

9.1.2 Real-time acquisition and analysis: user requirements

Objectives of the signal acquisition and analysis system

The objectives of the rolling mill signal acquisition and analysis are:

- to improve knowledge of the mill's behaviour and validate the proposed modifications;

- to help find fault sources rapidly;
- to provide operators with a manufacture product tracing system.

The signal source is the common memory of the three mill computers. The acquisition and analysis system realizes two operations:

- acquisition of signals which are generated by the rolling mill and their storage in a real-time database (RTDB);
- recording of some user configured signals on-demand.

Special constraints

The manufacturing process imposes availability and security constraints:

- Availability: the mill is operational day and night, with a solely preventive maintenance break of 8 or 16 hours once a week.
- Security: no perturbation should propagate up to the mill controlling systems since this may break the belt or cause fire in the mill (remember that the mill is lubricated with kerosene, which is highly flammable).

Signal acquisition frequency

The signal acquisition rate has to be equal to the signal production rate (which is itself fixed by the rolling evolution speed–the dynamics–and the Shannon theorem), and for the signal records to be usable, they have to hold all the successive acquired values during the requested recording period. The signals stored in the shared memory come from:

- the *Mod* computer, which writes 984 bytes every 4 ms (246 Kbytes/s) and additionally 160 bytes at a new product arrival (about once every 3 minutes);
- the *Dig* computer, which writes 544 bytes every 20 ms (27 Kbytes/s);
- the *Pla* computer, which writes 2052 bytes every 100 ms (20 Kbytes/s).

Rolling mill signal recording

It is required to record the real-time signal samples during a given period and after some conditioning. The recorded signals must then be stored in files for off-line processing. The operator defines the starting and finishing times of each record and the nature of the recorded samples. Records may be of three kinds:

- on operator request: for example when he wants to follow the manufacturing of a particular product;
- perpetual: to provide a continuous manufacturing trace;

- disrupt analysis: to retrieve the signal samples some period before and after a triggering condition. This condition may be belt tearing, fire or urgency stop.

The recording task has been configured to record 180 bytes every 4 ms over a 700 s period and thus it uses files of 32 Mbytes. These records are then processed off-line, without real-time constraints.

Immediate signal conditioning

The immediate signal conditioning includes raw signal analysis, real-time evolution display and dashboard presentation.

1. The raw signal analysis provides:
   - statistical information about a product and its quality trends;
   - computation of the belt length;
   - filtering treatment of the signal to delete noise and keep only the useful part of the signal, i.e. the thickness variations around zero.
2. Some values are displayed in real-time:
   - thickness variations of the input and output belt, with horizontal lines to point out the acceptable minimum and maximum;
   - flatness variations of the input and output belt. This flatness evolves during the production since heat dilates the rollers. Flatness is depicted on a coloured display called the flatness cartography. To get this cartography, the belt thickness is measured by 27 sensors spread across the belt width and is coded by a colour function of the measured value. The belt is plane when all the measures have the same colour. This allows easy visualization of the flatness variations as shown in Figure 9.3;
   - output belt speed. This allows estimation of the thickness variations caused by transient phases of the rolling mill;
   - planner of the regulations, in order to check them and to appraise their contribution to product quality;
   - belt periodic thickness perturbations which are mainly due to circumference defects of the rollers, caused by imperfect machining or by an anisotropic thermal dilatation. When the perturbations grow over the accepted margins, the faulty roller must be changed. These perturbations, at a 40 Hz frequency, are detected by frequency analysis using fast Fourier transform (FFT). Pulse generators located on the roller's axes pick up their rotation frequency. The first three harmonics are displayed. The FFT is computed with 1024 consecutive samples (the time window is thus $1024 \times 0.004 = 4\,\text{s}$).
3. The dashboard displays these evolutions, some numerical values, information and error messages, belt flatness instructions, and manufacturing characteristics (alloy, width, input and output nominal thickness, etc.). The screen resolution and its

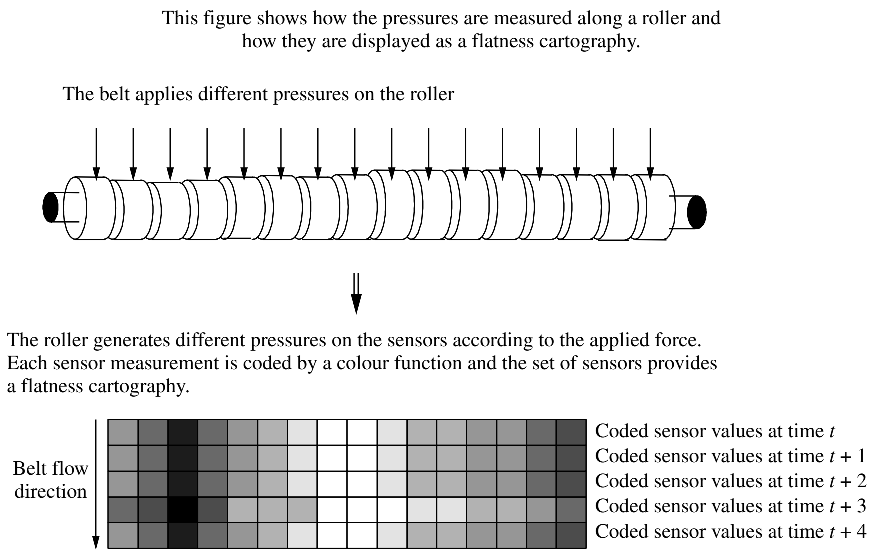
This figure shows how the pressures are measured along a roller and how they are displayed as a flatness cartography.
The belt applies different pressures on the roller
The roller generates different pressures on the sensors according to the applied force. Each sensor measurement is coded by a colour function and the set of sensors provides a flatness cartography.
Belt flow direction
Coded sensor values at time t
Coded sensor values at time t + 1
Coded sensor values at time t + 2
Coded sensor values at time t + 3
Coded sensor values at time t + 4

Figure 9.3 Roller geometry and flatness cartography

renewal rate (200 ms) are adapted to the resolution and dynamics of the displayed signals as well as to the eye's perception ability.

Automatic report generation

Every product passing in transit in the rolling mill automatically generates a report, which allows appraising of its manufacturing conditions and quality. The reported information is extracted from former computation and displays. The report is prepared in Postscript format and saved in a file. The last 100 reports are stored in a circular buffer before being printed. The reports are printed on-line, on operator request or automatically after a programmed condition occurrence. The requirement is to be able to print a report for every manufactured product whose manufacturing requires at least 5 minutes. The report printing queue is scanned every 2 seconds.

9.1.3 Assignment of operational functions to devices

Hardware architecture

The geographic distribution shows three sets:

- the control cabin for the operator, where the signal display and report printing facilities must be available;
- the power station, where all signals should be available and where the acquisition and analysing functions are implemented (computation, recording, report generation);

- the terminal room, where the environment is quiet enough for off-line processing of the stored records and for configuring the system.

Hardware and physical architecture choices

The Péchiney Rhénalu standards, the estimated numbers of interrupt levels and input–output cards, and the evaluation of the required processing power led to the following choices:

1. For the real-time acquisition and analysis computing system: real-time executive LynxOs version 3.0, VME bus, Motorola 2600 card with Power PC 200 MHz, 96 Mbytes RAM memory, 100 Mbits/s Ethernet port and a SCSI 2 interface, 4 SCSI 2 hard disks, each with a 1 Mbyte cache memory, and 8 ms access time. With this configuration, LynxOs reports the following performance:
   - context switch in 4 microseconds;
   - interrupt handling in less than 11 microseconds;
   - access time to a driver in 2 microseconds;
   - semaphore operation in 2 microseconds;
   - time provided by `getimeofday()` system call with an accuracy of 3 microseconds.
2. For off-line processing and on-line display: two Pentium PCs.
3. For connecting the real-time acquisition and analysis computer and the two other functionally dependent PC computers: a fast 100 Mbytes CSMA/CD Ethernet with TCP/IP protocol.
4. For acquiring the rolling mill data: the ultra fast optic fibre network Scramnet that is already used by the mill control computers. Scramnet uses a specific protocol simulating a shared memory and allowing processors to write directly and read at a given address in this simulated shared memory. Each write operation may raise an interrupt in the real-time acquisition and analysis computer and this interrupt can be used to synchronize it. The data are written by the emitting processor in its Scramnet card. The emission cost corresponds to writing at an address in the VME bus or in a Multibus, and the application can tolerate it. The writing and reading times have been instrumented and are presented Table 9.1.

Table 9.1 Scramnet access times

Action	**Number of useful bytes**	**Mean time (μs)**	**Useful throughput (Kb/s)**
Writing by *Mod*	984	689	1395
Writing by *Dig*	544	1744	305
Writing by *Pla*	2052	2579	777
Reading by LynxOs	984	444	2164

9.1.4 Logical architecture and real-time tasks

Real-time database

The application shares a common data table that is used as a blackboard by all programs, as shown in Figure 9.4. This table is resident in main memory and mapped into the shared virtual memory of the Posix tasks. Data are stored as arrays in the table.

To allow users to reference the signals by alphanumeric names, as well as allowing tasks to access them rapidly by addresses in main memory, dynamic binding is used and the binding values are initialized anew at each database restructuring. This use of precompiled alphanumeric requests causes this table to be called a real-time database (RTDB).

Real-time tasks

The set of periodic tasks and the recording of the rolling steps (rolling start, acceleration, rolling at constant speed, deceleration, rolling end) are synchronized by the emission of the *Mod* computer signals every 4 ms. This fastest sampling rate fixes the basic cycle. In the following we present the tasks, the precedence relations between some of them, the empirically chosen priorities, and the task synchronization implementation. The schemas of some tasks are given in Figures 9.5 and 9.8.

The three acquisition tasks: modcomp, digigage and planicim The acquisition of rolling mill signals must be done at the rate of the emitting computer. This hard

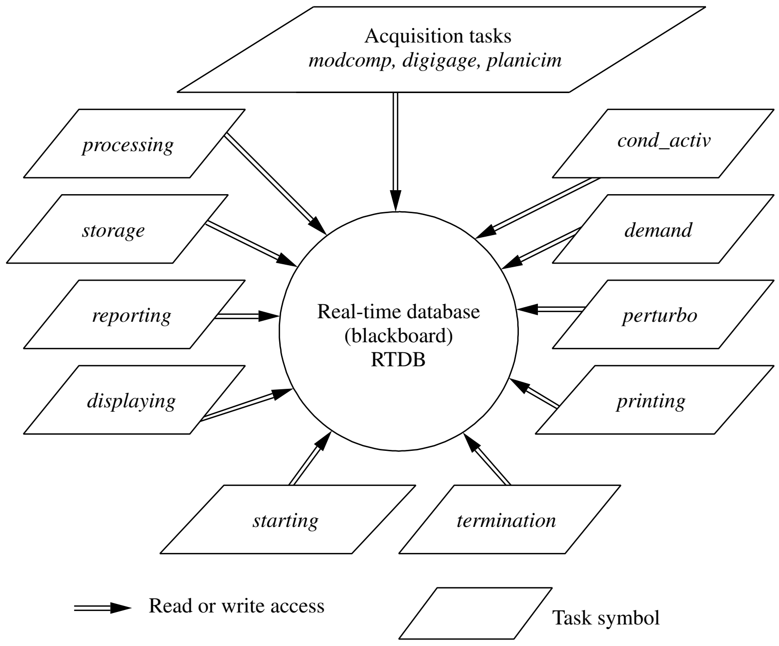

Figure 9.4 Real-time database utilization

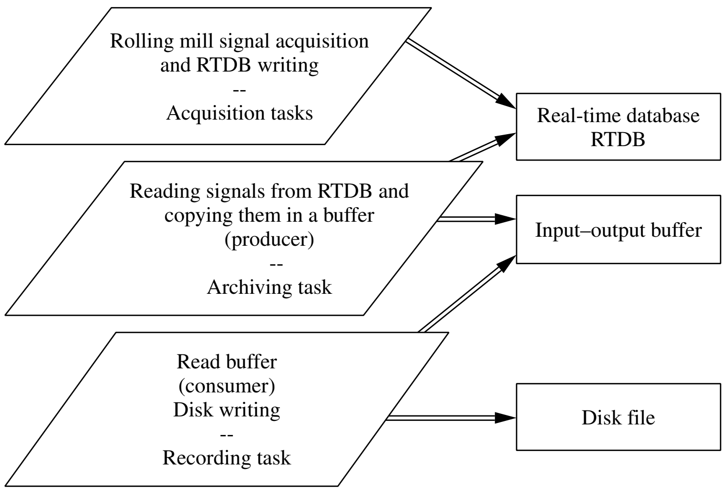

Figure 9.5 The recorded data flow

timing constraint (due to signal acquisition frequency) is necessary for recording the rolling mill dynamics correctly. Flatness regulation signals come from the *Pla* computer with a period of 100 ms. Thickness low regulation signals come from the *Dig* computer with a period of 20 ms. Thickness rapid regulation signals are issued from the *Mod* computer with a period of 4 ms. One acquisition task is devoted to each of these signal sources. An interrupt signalling the end of writes in Scramnet is set by the writer. We note the three acquisition tasks as *modcomp, digigage* and *planicim*. The acquisition task deposits the acquired signals in the RTDB memory-resident database. The interrupt signal allows checking whether the current computation time of a task remains lower than its period. A trespassing task, i.e. one causing a timing fault, is set faulty and stopped. This also causes the whole acquisition and analysis system to stop, without any perturbation of the rolling mill control or the product manufacturing.

Activation conditions task: cond_activ The activation condition task (called *cond_activ*) is the dynamic interpreter of the logic equations set specifying the list of samples to record or causing automatic recording to start when the signals detect that a product has gone out of tolerance. These logic equations are captured at system configuration, parsed and compiled into an evaluation binary tree. This task is triggered every 4 ms by the *modcomp* task with a relative deadline value equal to its period.

Immediate signal processing task: processing The signal processing task (called *processing*) reads the new signal samples in the database, computes the data to be displayed or stored and writes them in the database. It computes the statistical data, the FFT, the belt length, and the filtering of some signals. This processing must be done at the acquisition rate of the fastest signals to recording the rolling mill dynamics correctly. This task is triggered every 4 ms by the *modcomp* task with a relative deadline value equal to its period.

Record archiving tasks: storage, perturbo and demand The three record archiving tasks, called *storage, perturbo* and *demand*, must operate at the acquisition rate of the fastest signals. This means that some timing constraints have to be taken into account to record the rolling mill dynamics correctly. Thus the tasks are released every 4 ms by the *modcomp* task with a relative deadline value equal to its period. Each task reads the recorded signals in the database and transfers them to files on disks, using producer–consumer schemes with a two-slot buffer for each file. The archiving tasks (i.e. *storage, perturbo* and *demand* tasks) write to the buffers while additional tasks, called *recording* consume from the buffers the data to be transferred to disks. Those *recording* tasks, one per archiving task, consume very little processor time and this can be neglected. They have a priority lower than the least priority task of period 4 ms (their priority is set to 5 units below their corresponding archiving task).

Signal displaying task: displaying Signal displaying (task called *displaying*) requires a renewal rate of 200 ms. This is a deadline with a soft timing constraint, since any data which is not displayed at a given period may be stored and displayed at the next period. There is no information loss for the user, who is concerned with manufacturing a product according to fixed specifications. For this he or she needs to observe the minimum, maximum and mean values of the signal since the last screen refresh. The display programs use an X11 graphical library and the real-time task uses the PC as an X server.

Report generating task: reporting The reports must be produced (by the task called *reporting*) with a period of 200 ms. This task also has a soft deadline.

Report printing task: printing Report printing (the task is named *printing*) is required either automatically or by the operator. The task is triggered periodically every two seconds and it checks the Postscript circular buffer for new reports to print.

Initializing task: starting The application initialization is an aperiodic task (called *starting*) which prepares all the resources required by the other tasks. It is the first to run and executes alone before it releases the other tasks. A configuration file specifies the number, type and size of files to create. There may be up to 525 files, totalling 2.5 Gbytes. All files are created in advance, and are allocated to tasks on demand. At the first system installation, this file creation may take up to one hour.

Closing task: termination The application closure is performed by an aperiodic task (called *termination*) which releases all used resources. It is triggered at the application end.

Precedence relationships

The successive signal conditionings involve precedence relationships between the tasks: acquisition must be done before signal processing and the evaluation of activation conditions. These tasks must in turn precede record archiving, display and report generation. *Starting* precedes every task and *termination* stops them all before releasing their resources. Figure 9.6 shows the precedence graph.

When the task *modcomp* has set the signal samples in the database, it activates the other periodic tasks which use these samples; *digigage* and *planicim,* which have larger periods, also deposit some samples. The 4 ms period tasks check a version number to know when the larger period samples have been refreshed.

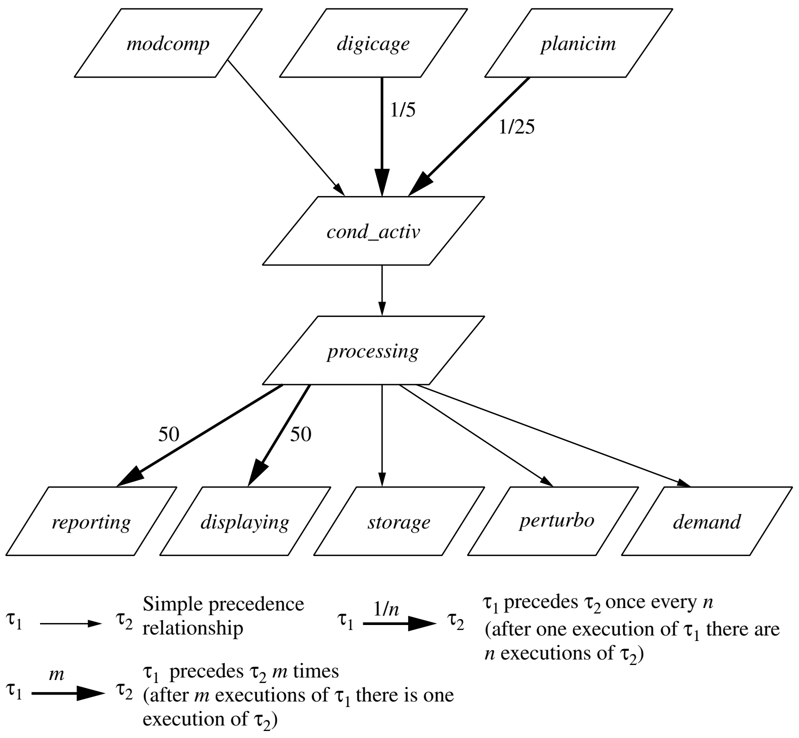
modcomp
digicage
planicim
1/5
1/25
cond_activ
processing
50
50
reporting
displaying
storage
perturbo
demand
$\tau_1 \to \tau_2$ Simple precedence relationship
$\tau_1 \xrightarrow{1/n} \tau_2$ τ_1 precedes τ_2 once every n (after one execution of τ_1 there are n executions of τ_2)
$\tau_1 \xrightarrow{m} \tau_2$ τ_1 precedes τ_2 m times (after m executions of τ_1 there is one execution of τ_2)

Figure 9.6 Tasks precedence graph

Empirical priorities of tasks

The LynxOs system has a fixed priority scheduler, with 255 priority levels, the higher level being 255. The priorities have been chosen on a supposed urgency basis and the higher priorities have been given to the tasks with the harder timing constraints. It has been checked that the result was a feasible schedule.

Table 9.2 presents the empirical constant priorities given to each task, the period T, the measured computation time C (the minimum, maximum and mean values have been recorded by measuring the start and finish time of the requests with the `getimeofday()` system call), the relative deadline D and the reaction category in case of timing fault.

Synchronization by semaphores

In the studied system, the periodic tasks are not released by a real-time scheduler using the system clock. The basic rate is given directly by the rolling mill and by the end-of-write interrupt which is generated every 4 ms by the *Mod* computer.

The task requests triggering and the task precedence relationships are programmed with semaphores which are used as synchronization events. Recall that a semaphore S is used by means of two primitives, $P(S)$ and $V(S)$ (Silberschatz and Galvin, 1998; Tanenbaum and Woodhull 1997).

Table 9.2 The tasks of the acquisition and analysis system

Task	**Priority**	***T*** **ms**	***Cmin*** **(μs)**	***Cmax*** **(μs)**	***Cmean*** **(μs)**	***D*** **(ms)**	**Reaction to faults**
starting	50					30 000	5
modcomp	50	4	600	992	613	1	1
cond_activ	38	4	136	221	141	4	2
processing	36	4	92	496	106	4	2
storage	34	4	128	249	136	4	3
perturbo	33	4	112	218	120	4	3
demand	32	4	155	348	167	4	3
digigage	30	20	860	1430	1130	10	1
planicim	29	100	1800	2220	1920	50	1
displaying	27	200	512	1950	1510	200	4
reporting	26	200	475	2060	1620	200	4
printing	18	2000				300 000	4
termination	50						5

The periodic tasks are programmed as cyclic tasks which block themselves on their private semaphore (a semaphore initialized with state 0) at the end of each cycle. An activation cycle corresponds to a request execution. Thus *modcomp* blocks itself when executing *P(S_modcomp), cond_activ* when executing *P(S_cond_activ), processing* when executing *P(S_processing), demand* when executing *P(S_demand)*, and so on. At each 4 ms period end, all the tasks are blocked when there is no timing fault.

The *Mod* computer end-of-write interrupt causes the execution of a *V(S_modcomp)* operation, which awakes the *modcomp* task. When this task finishes and just before blocking again by executing *P(S_modcomp),* it wakes up all the other periodic tasks by executing *V(S_cond_activ), V(S_processing), . . ., V(S_demand).* Every 5 cycles it wakes task *digigage*; every 25 cycles it wakes task *planicim*; . . .; every 500 cycles it wakes task *printing*. The execution order is fixed by the task priority (there is only one processor and the cyclic tasks are not preempted for file output since the *recording* tasks have lower priorities). This implements the task precedence relationships. The synchronization of the 11 cyclic tasks is depicted in Figure 9.7.

Task *modcomp* also monitors each task τ_x it awakes. In nominal behaviour, τ_x is blocked at the time of its release. This is checked by *modcomp* reading S_τ_x's state (S_τ_x is the private semaphore of τ_x). S_τ_x's state represents the history of operations on S_τ_x and it memorizes therefore whether, before being preempted by *modcomp*, the cyclic task τ_x was able or not to execute the $P(S\_\tau_x)$ operation which ends the cycle, blocking τ_x anew. This solution is correct only in a uniprocessor computer and if *modcomp* is the highest priority task and able to preempt the other tasks.

To sum up, the task *modcomp* starts each 4 ms cycle when receiving the Scramnet interrupt mapped to a *V* semaphore operation. It executes its cyclic program, checks the time limit of the tasks and then awakes all the tasks concerned with the current cycle. Figure 9.8 presents the task schema of *modcomp* and of the archiving tasks.

Finally, when a task needs signals acquired by a task other than *modcomp*, it reads the database and checks for them. Each of the data structures resulting from acquisition or processing is given a version number that is incremented at each update. The client programs have their own counter and compare its value to the current version number to check for a new value. The version numbers are monotonously increasing. If their

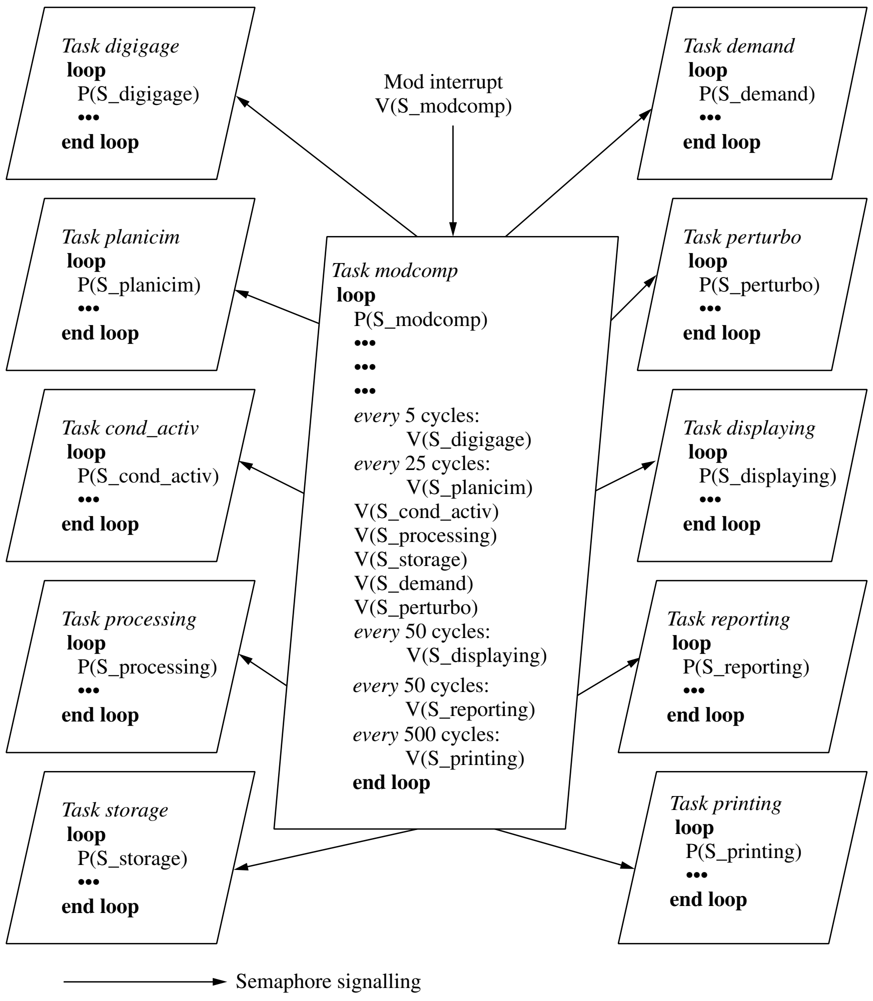

Figure 9.7 Synchronization by semaphores

incrementation can be made by an atomic operation (between tasks), there is no need to use a mutual exclusion semaphore.

Reactions to timing faults

Timing faults are detected by task *modcomp* as explained above. The reaction depends on the criticality of the faulty task (Table 9.2) and is related to one of the following categories:

- Category 1: the computing system is stopped since the sampled signals do not represent the rolling mill dynamics. The values have not been read at the same sampling instant (this category concerns the three acquisition tasks, *modcomp*, *digigage* and *planicim*).

```
Archiving task/** tasks storage, perturbo and demand
 begin
   open database
   open synchronization table
   open allocation table
   start the buffer consumer task
   while (no required stop) loop
     wait for a required archive
     read configuration and open archiving file
     create the two slots buffer for the recorded signals
       /** each buffer size is set to the recorded signal size and rate
     wait for the start recording authorization /** blocked by P(S_producer)
     while (not(end recording condition) or not(max recording time))loop
       write each signal in its current buffer
       if the current buffer is full then
         activate the consumer recording task   /** with V(S_consumer)
         point to the other buffer         /** with P(S_producer)
       end if
     end loop
     wait until the last buffer is saved
     close the archiving file
   end loop     /** loop controlled by (no required stop)
   close database
   close synchronization table
   close allocation table
 end/** archiving task

Recording task
 begin
   while (no required stop) loop
     wait until a buffer is ready /** with P(S_Consumer)
     transfer the buffer to the file, indicate free buffer /** with V(S_producer)
   end loop /** loop controlled by (no required stop)
 end/** Recording task

Acquisition task          /** task modcomp
 begin
   Scramnet initialization
   open database
   open synchronization table
   while(no required stop)loop
     wait the Scramnet interrupt /** with P(S_modcomp)
     read Scramnet and write the samples in the database
     monitor other tasks
     awake the other tasks, τx   /** with V(S_τx)
   end loop                      /** loop controlled by (no required stop)
   close database
   close synchronization table
 end/** acquisition task
```

Figure 9.8 *Modcomp* and archiving task schemes

- Category 2: the computing system is stopped since the computed values are incorrect and useless (this category concerns the conditions elaboration task, *cond_activ*, and the signal processing task: *processing*).
- Category 3: the function currently performed by the task is stopped since its results are not usable (this category concerns the three record archiving tasks, *storage, perturbo* and *demand*).
- Category 4: the current function is not stopped but the fault is recorded in the logbook (journal). The recurrent appearance of this fault may motivate the operator to alleviate the processor load by augmenting the task period or reducing the number of required computations (this category concerns the signal displaying task, *displaying*, the report generating task, *reporting*, and the report printing task, *printing*).
- Category 5: nothing is done since the fault consequences are directly noticed by the operator (this category concerns the initializing and the closing task).

It should be noted that these reactions have some correlation with the task precedence relationships.

9.1.5 Complementary studies

Complementary studies of this rolling mill are suggested below.

Scheduling algorithms

Let us suppose that the task requests are released by a scheduler that uses the LynxOs real-time clock whose accuracy is 3 microseconds. The precedence relationships are no longer programmed but the scheduler takes care of them.

- Study the schedulability of the 11 periodic tasks with an on-line empirical fixed priority scheduler as in the case study.
- Study the schedulability of the 11 periodic tasks with the RM algorithm.
- Study the schedulability of the 11 periodic tasks with the EDF algorithm

Scheduling with shared exclusive resources

Let us suppose that the shared data in the database are protected by locks implemented with mutual exclusion semaphores (*P* or *V* operation time is equal to 2 microseconds). Analyse the influence of access conflicts, context switches (the thread context switch time is equal to 4 microseconds) and the additional delays caused by the database locking with different lock granularity.

Robustness of the application

Compute the laxity of each task and the system laxity for:

- evaluating the global robustness. For example, consider slowing down the processor speed as much as acceptable for the timing constraints.
- evaluating the margin for the task behaviour when its execution time increases.
- estimating the influence of random perturbations caused by shared resource locking.

To introduce some timing jitter, it is necessary to increase the processor utilization factor of some tasks. Reducing the period of some tasks can do this, for example. Then, once a jitter has appeared:

- introduce a start time jitter control for the signal displaying task,
- introduce a finish time jitter control for the *processing* and *reporting* tasks. This allows simulating a sampled data control loop monitoring the actuators.

Multiprocessor architecture

Let us suppose a multiprocessor is used to increase the computing power. Study the task scheduling with two implementation choices. In the first one, the basic rate is still given by the rolling mill, and cyclic task synchronization and wake up are done by program. In the second case, the LynxOs real-time clock (accuracy of 3 microseconds) and a real-time scheduler are used.

Task precedence must be respected and the mixing of priorities and event-like semaphores cannot be used, since the uniprocessor solution is no longer valid. The fault detection that the redundancy allowed is not valid either.

Network

The use of Scramnet is costly. Examine the possibilities and limits of other real-time networks and other real-time protocols. Consider several message communication schemes between the application tasks. Finally, as in the example presented in Section 6.4.3, consider message scheduling when the network used is CAN, FIP or a token bus.

9.2 Embedded Real-Time Application: Mars Pathfinder Mission

9.2.1 Mars Pathfinder mission

After the success of early Mars discovery missions (Viking in 1976), a long series of mission failures have limited Mars exploration. The Mars Pathfinder mission was an important step in NASA discovery missions. The spacecraft was designed, built and operated by the Jet Propulsion Laboratory (JPL) for NASA. Launched on 4 December 1996, Pathfinder reached Mars on 4 July 1997, directly entering the planet's atmosphere and bouncing on inflated airbags as a technology demonstration of a new way to deliver a lander of 264 kg on Mars. After a while, the Pathfinder stationary lander released a micro-rover, named Sojourner. The rover Sojourner, weighing 10.5 kg, is a six-wheeled vehicle controlled by an earth-based operator, who used images obtained by both the rover and lander systems. This control is possible thanks to two communication devices: one between the lander and Earth and the other between the lander and the rover, done by means of high frequency radio waves. The Mars Pathfinder's rover rolled onto the surface of Mars on 6 July at a maximum speed of 24 m/h. Sojourner's mobility provided the capability of discovering a landing area over hundreds of square metres on Mars.

The scientific objectives included long-range and close-up surface imaging, and, more generally, characterization of the Martian environment for further exploration. The Pathfinder mission investigated the surface of Mars with several instruments: cameras, spectrometers, atmospheric structure instruments and meteorology, known as ASI/MET, etc. These instruments allowed investigations of the geology and surface morphology at sub-metre to one hundred metres scale. During the total mission, the spacecraft relayed 2.3 gigabits of data to Earth. This huge volume of information included 16 500 images from the lander camera and 550 images from the rover camera, 16 chemical analyses and 8.5 million measurements of atmospheric conditions, temperature and wind.

After a few days, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total resets, each resulting in losses of data. By using an on-line debug, the software engineers were able to reproduce the failure, which turned out to be a case of priority inversion in a concurrent execution context. Once they had understood the problem and fixed it, the onboard software was modified and the mission resumed its activity with complete success. The lander and the rover operated longer than their design lifetimes. We now examine what really happened on Mars to the rover Sojourner.

9.2.2 Hardware architecture

The simplified view of the Mars Pathfinder hardware architecture looks like the one-processor architecture, based on the RS 6000 microprocessor, presented in Figure 9.9. The hardware on the rover includes an Intel 8085 microprocessor which is dedicated to particular automatic controls. But we do not take into account this processor because it has a separate activity that does not interfere with the general control of the spacecraft.

The main processor on the lander part is plugged on a VME bus which also contains interface cards for the radio to Earth, the lander camera and an interface to a specific 1553 bus. The 1553 bus connects the two parts of the spacecraft (stationary lander and rover) by means of a high frequency communication link. This communication link was inherited from the Cassini spacecraft. Through the 1553 bus, the hardware on the lander part provides an interface to accelerometers, a radar altimeter, and an instrument for meteorological measurements, called ASI/MET.

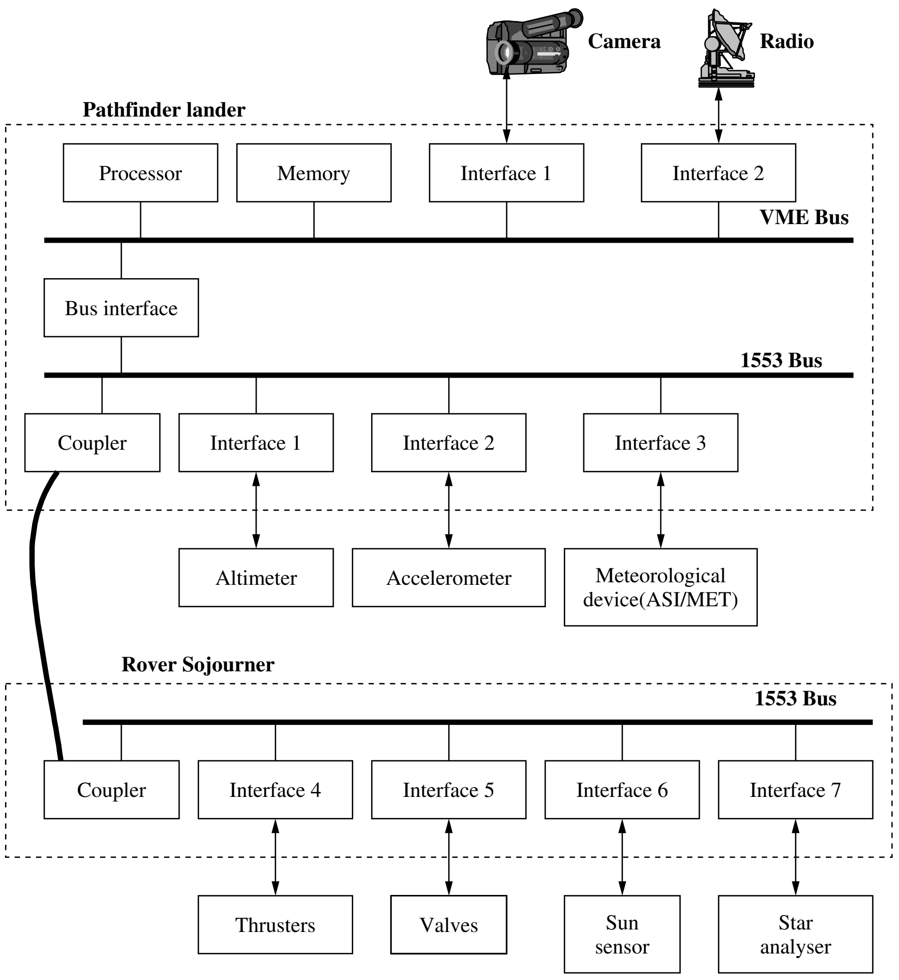

Figure 9.9 Hardware architecture of Pathfinder spacecraft

The hardware on the rover part includes two kinds of devices:

- Control devices: thrusters, valves, etc.
- Measurement devices: a camera, a sun sensor and a star scanner.

9.2.3 Functional specification

Given the hardware architecture presented above, the main processor of the Pathfinder spacecraft communicates with three interfaces only:

- radio card for communications between lander and Earth;
- lander camera;
- 1553 bus interface linked to control or measurement devices.

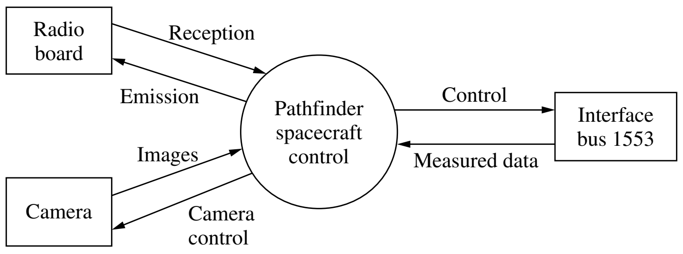

Figure 9.10 Context diagram of Pathfinder mission according to SA-RT method

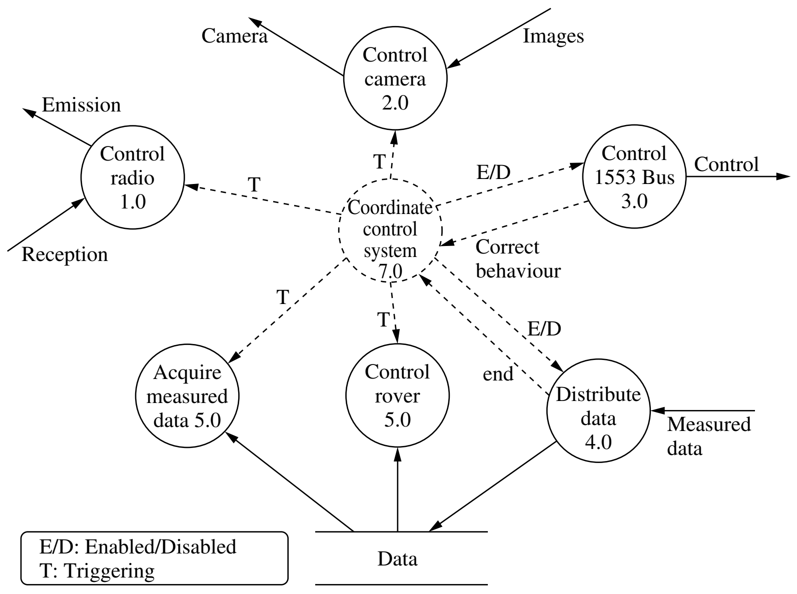

Figure 9.11 Preliminary data flow diagram of Pathfinder mission

The context diagram of this application is presented in Figure 9.10 according to the Structured Analysis for Real-Time systems (SA-RT) (Goldsmith, 1993; Hatley and Pirbhai, 1988). As explained above, there are only three terminators, external entities connected to the monitoring system. The first step of decomposition is shown as a preliminary data flow diagram in Figure 9.11. In order to simplify the analysis of this complex application, only the processes active during the Mars exploration phase have been represented. Other processes, active during the landing phase or the flight, have been omitted. The control process, numbered 7.0, corresponds to the scheduling of the other functional processes and could be specified by a state transition diagram.

9.2.4 Software architecture

The software architecture is a multitasking architecture, based on the real-time embedded system kernel VxWorks (Wind River Systems). The whole application includes over 25 tasks. These tasks are either periodic (bus management, etc.) or aperiodic (error analysis, etc.). The synchronization and communications are based on reader/writer paradigm or message queues. Some of these tasks are:

- mode control task (landing, exploration, flight, etc.);
- surface pointing control task (entering Mars's atmosphere);
- fault analysis task (centralized analysis of the error occurring in the tasks);
- meteorological data task (ASI/MET);
- data storage task (in EEPROM);
- 1553 bus control task (see further detailed explanations);
- star acquisition task;
- serial communication task;
- data compression task;
- entry/descent task.

It is important to outline that the mission had quite different modes (flight, landing, exploration), so a specific task is responsible for managing the tasks that have to be active in each mode. In this study we are only interested in the exploration mode. Moreover, in order to simplify the understanding of the problem, the application presented and analysed here is derived from the original real Pathfinder mission.

The simplified software task architecture is presented in Table 9.3 and in Figure 9.12 according to a diagram of the Design Approach for Real-Time Systems (DARTS) method (Gomaa, 1993). This task diagram consists of the different tasks of the application and their communications. All the tasks of the analysed application are considered to be periodic and activated by an internal real-time clock (RTC). It is important to notice that four tasks (*Data_Distribution*, *Control_Task*, *Measure_Task*, *Meteo_Task*) share a critical resource, called *Data*, that is used in mutual exclusion. Two operations are provided by the data abstraction module: read and write. The different tasks are

Table 9.3 Task set of Pathfinder application in the exploration mode

Priority	**Task**	**Comments**
The highest	*Bus_Scheduling*	1553 bus control task
↑	*Data_Distribution*	1553 bus data distribution task
↑	*Control_Task*	Rover control task
↑	*Radio_Task*	Radio communication management task
↑	*Camera_Task*	Camera control task
↑	*Measure_Task*	Measurement task
The lowest	*Meteo_Task*	Meteorological data task

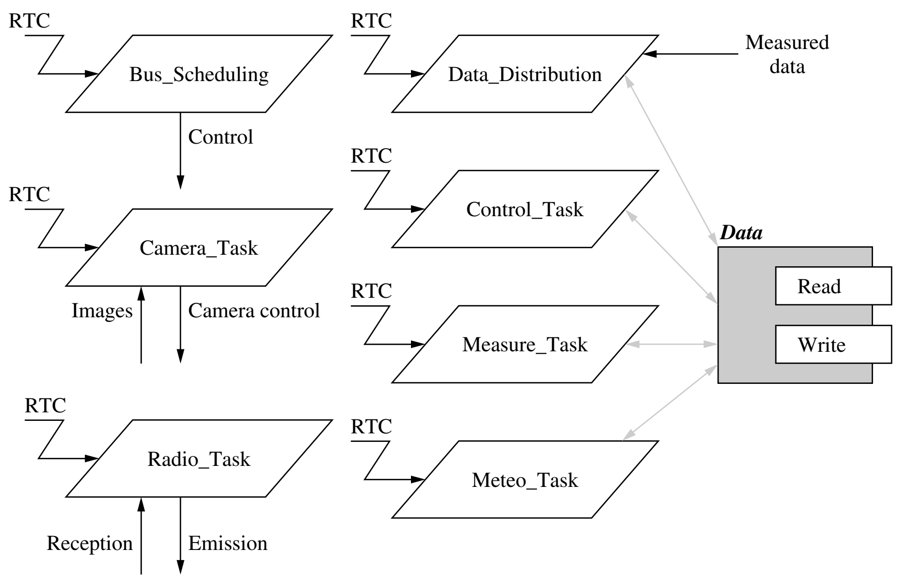

Figure 9.12 Task architecture of Pathfinder mission (RTC: real-time clock)

reader and writer tasks that can access these data in a critical section. The theory presented in Chapter 3 has been applied to this case study.

9.2.5 Detailed analysis

The key point of this application is the management of the 1553 bus that is the main communication medium between tasks. The software schedules this bus activity at a rate of 8 Hz (period of 125 ms). This feature dictates the architecture software which controls both the 1553 bus itself and the devices attached to it.

The software that controls the 1553 bus and the attached instruments is implemented as two tasks:

- The first task, called *Bus_Scheduling*, controls the setup of the transactions on the 1553. Each cycle, it verifies that the transaction has been correctly realized, particularly without exceeding the bus cycle. This task has the highest priority.
- The second task handles the collection of the transaction results, i.e. the data. The second task is referred to as *Data_Distribution*. This task has the third highest priority in the task set; the second priority is assigned to the entry and landing task, which has not been activated in the studied exploration mode. So the main objective of this task is to collect data from the different instruments and to put them in the shared data module *Data*.

A typical temporal diagram for the 1553 bus activity is shown in Figure 9.13. First the task *Data_Distribution* is awakened. This task is completed when all the data distributions are finished. After a while the task *Bus_Scheduling* is awakened to set

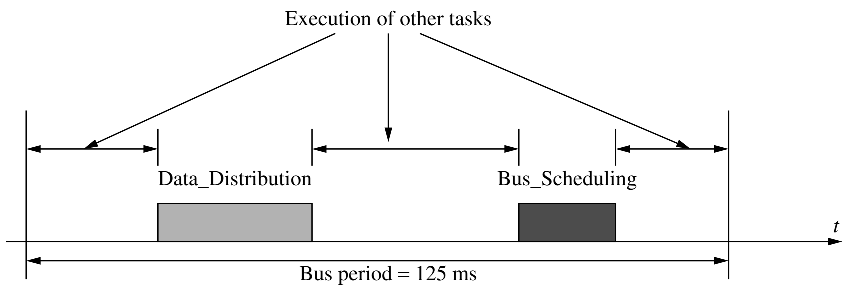

Figure 9.13 Typical temporal diagram for the 1553 bus activity

up transactions for the next cycle. The times between these executions are devoted to other tasks. This cycle is repeated indefinitely.

Except for the periods of the first two tasks *Bus_Scheduling* and *Data_Distribution*, which are specified with exact values corresponding to the real application, the timing characteristics of tasks (execution time and period) were estimated in order to get a better demonstrative example. These task parameters are presented in Table 9.4 in decreasing priority order. The timing parameters (C_i and T_i) have been reduced by assuming a processor time unit of 25 ms. In order to simplify the problem, we assume that the critical sections of all tasks using the shared critical resource have a duration equal to their execution times. Except for the task called *Meteo_Task*, the parameters are considered as fixed. The *Meteo_Task* has an execution time equal to either 2 or 3, corresponding to more or less important data communication size.

The processor utilization factor of this seven-task set is equal to 0.72 (respectively 0.725) for an execution time of *Meteo_Task* equal to 2 (respectively 3). We can note that both values are lower than the limit ($U \leq 0.729$) given by the sufficient condition for RM scheduling (see condition (2.12) in Chapter 2). So this application would be schedulable if the tasks were independent. But the relationships between tasks, due to the shared critical resource *Data*, lead to simulation of the execution of the task set with the two different values of the *Meteo_Task* execution time. This simulation has to be done over the LCM of the task periods, that is to say 5000 ms (or 200 in reduced time).

In Figure 9.14, the execution sequence of this task set for the *Meteo_Task* execution time equal to 2 is shown. As we can see, the analysis duration is limited to the reduced

Table 9.4 Pathfinder mission task set parameters

Task	Priority	Parameters (ms)		Reduced parameters		Critical section duration
		C_i	T_i	C_i	T_i	
Bus_Scheduling	7	25	125	1	5	—
Data_Distribution	6	25	125	1	5	1
Control_Task	5	25	250	1	10	1
Radio_Task	4	25	250	1	10	—
Camera_Task	3	25	250	1	10	—
Measure_Task	2	50	5000	2	200	2
Meteo_Task	1	{50, 75}	5000	{2, 3}	200	{2, 3}

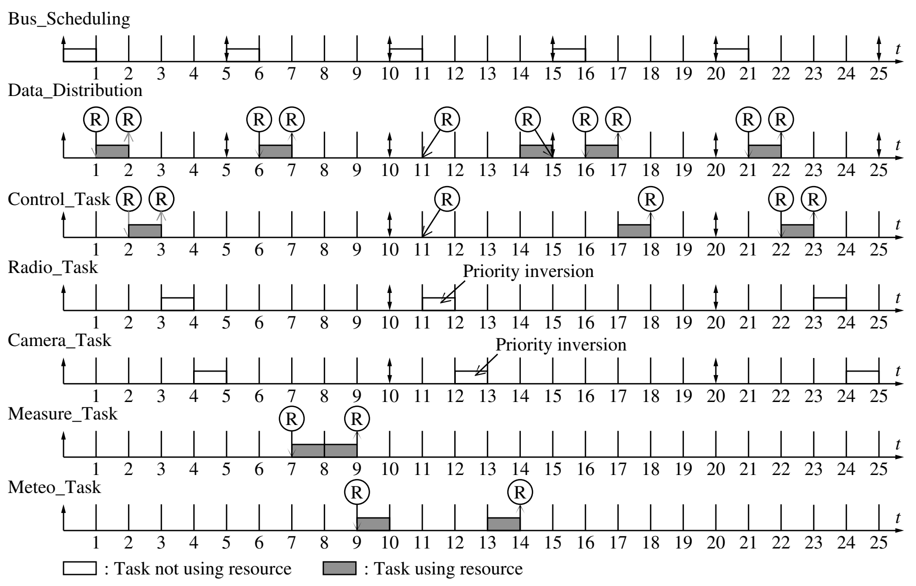

Figure 9.14 Valid execution sequence of Pathfinder mission for a *Meteo_Task* task with execution time equal to 2

time 25 because, when the execution of the *Measure_Task* and *Meteo_Task* tasks have completed, the study can be limited to the next period of the other tasks. The obtained execution sequence is valid in the sense that all tasks are within their deadlines. The *Measure_Task* and *Meteo_Task* tasks end their executions at time 14 and the others produce a valid execution sequence in the time range [20, 30] that is indefinitely repeated until the end of the major cycle, equal to 200.

It is worth noticing that all the waiting queues are managed according to the task priority. Moreover, the tasks which use the critical resource *Data* are assumed to acquire it at the beginning of their activation and to release it at the end of their execution. When this resource request cannot be satisfied because another task is using the critical resource, the kernel primitive implementing this request is supposed to have a null duration.

It is not difficult to see that a priority inversion phenomenon occurs in this execution sequence. At time 11, the *Data_Distribution* task which is awakened at time 10, should get the processor, but the *Meteo_Task* task, using the critical resource, blocks this higher priority task. The *Camera_Task* and *Radio_Task* tasks, which do not need the shared exclusive resource and are awakened at time 11, have a priority higher than task *Meteo_Task*, and as a consequence they get the processor one after the other at times 11 and 12. Then *Meteo_Task* task can resume its execution and release the critical resource at time 14. Finally the higher priority task, *Data_Distribution* task, resumes its execution and ends just in time before its deadline 15 (Figure 9.14).

The priority inversion phenomenon leads to an abnormal blocking time of a high priority task, here *Data_Distribution* task, because it uses a critical resource shared by a lower priority task, *Meteo_Task*, and two intermediate priority tasks, *Camera_Task* and *Radio_Task* tasks, can execute.

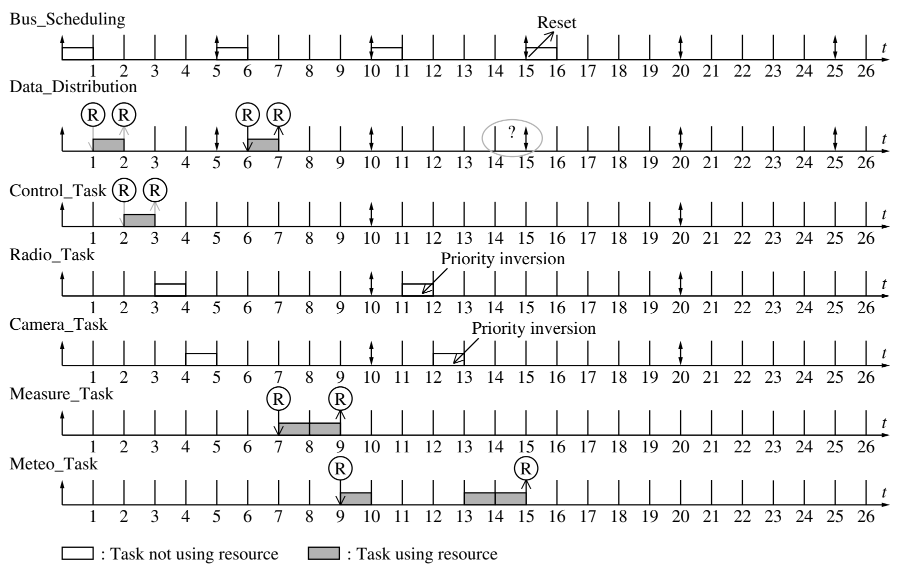

Figure 9.15 Non-valid execution sequence of Pathfinder mission for a *Meteo_Task* task with execution time equal to 3

Let us suppose now that *Meteo_Task* has an execution time equal to 3. The new execution temporal diagram, presented in Figure 9.15, shows that the *Data_Distribution* task does not respect its deadline. This temporal fault is immediately detected by the *Bus_Scheduling* task and leads to a general reset of the computer: this caused the failure of Pathfinder mission. This reset initialized all hardware and software. There is no loss of collected data. However, the remainder of the activities were postponed until the next day.

In order to prevent this priority inversion phenomenon, it is necessary to use one specific resource management protocol as seen in Chapter 3. Figure 9.16 illustrates the efficiency of the priority inheritance protocol. The execution sequence is now valid even though *Meteo_Task* task has an execution duration equal to 3. In fact the intermediate priority tasks, *Camera_Task* and *Radio_Task* tasks, are executed after *Meteo_Task* task because this task inherits the higher priority of *Data_Distribution* task. In this case, it is interesting to notice that the *Meteo_Task* task execution time can be as long as 3 units without jeopardizing the valid execution sequence.

In order to avoid the priority inversion phenomenon, one can also use another protocol based on the assignment of the highest priority to the task which is in a critical section. Actually, this resource management protocol leads to forbidding the execution of other tasks during critical sections (Figure 9.17). But a drawback of this protocol is that a lower priority task using a resource can block a very high priority task, such as the *Bus_Scheduling* task in the considered application.

9.2.6 Conclusion

Being focused on the entry and landing phases of the Pathfinder mission, engineers did not take enough care over testing the execution of the exploration mode. The actual

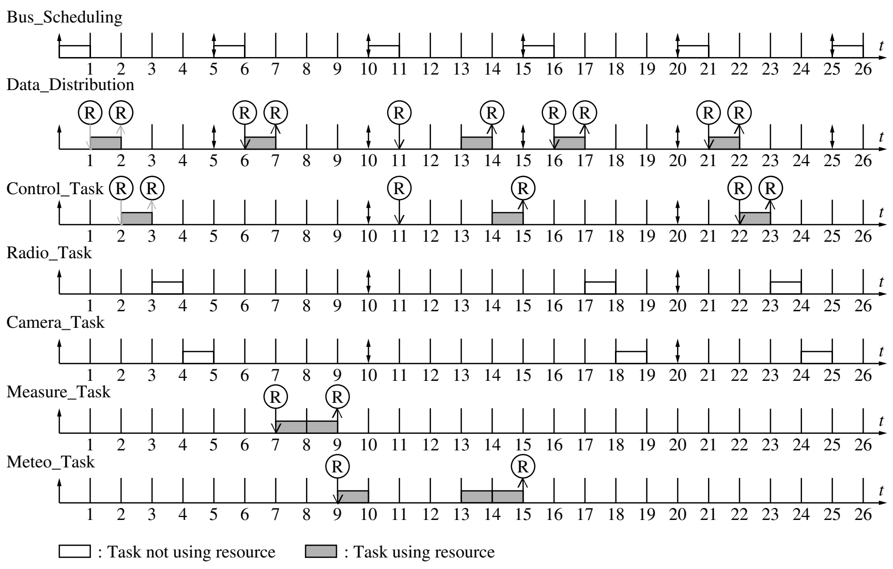

Figure 9.16 Valid execution sequence of Pathfinder mission by using a priority inheritance protocol and for a *Meteo_Task* task with execution time equal to 3

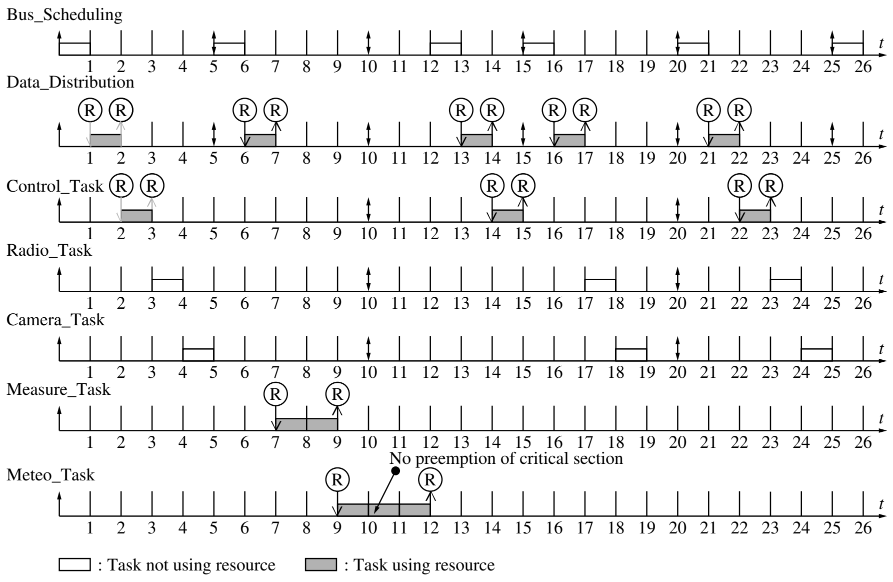

Figure 9.17 Valid execution sequence of Pathfinder mission by using a highest priority protocol and for a *Meteo_Task* with execution duration equal to 3

data rates were higher than estimated during the pre-flight testing and the amount of science activities, particularly meteorological instrumentation, proportionally greater. This higher load aggravated the problem of using the critical resource (communication on 1553 bus). It is important to outline that two system resets had occurred in the pre-flight testing. As they had never been reproducible, engineers decided that they were probably caused by a hardware glitch. As this part of the mission was less critical, the software was not protected against the priority inversion phenomenon by using a mutex semaphore implementing priority inheritance. A VxWorks mutex object includes a Boolean parameter that indicates whether priority inheritance should be performed by the semaphore management. In this case the mutex parameter was off. Once the problem was understood the modification appeared obvious: change the creation flags of the semaphore and enable the priority inheritance. The onboard software was modified accordingly on the spacecraft. This application, which we have simplified for a better understanding, has been studied by assuming a scheduling based on fixed priority (RM algorithm) and a priority inheritance protocol for managing the exclusive resource. This study can be prolonged by analysing the execution sequence produced by the following scheduling contexts:

- scheduling with variable priorities (for example, earliest deadline first);
- other resource management protocol (for example, priority ceiling protocol).

9.3 Distributed Automotive Application

9.3.1 Real-time systems and the automotive industry

Nowadays, car manufacturers integrate more and more microcontrollers that manage the brakes, the injection, the performance, and the passenger comfort (Cavalieri et al., 1996). For instance, the engine control system aims to manage the engine performance in terms of power, to reduce fuel consumption and to control the emission of exhaust fumes. This control is obtained by sending computed values to the actuators: electronic injectors, electromagnetic air valve for managing the idling state of the engine (i.e. the driver does not accelerate) and fuel pump. The ABS system prevents the wheels from locking when the driver brakes. The system must also take into account sudden variations in the road surface. This regulation is obtained by reading periodically the rotation sensors on each wheel. If a wheel is locked, then the ABS system acts directly on the brake pressure actuator. Complementary information on the process control functionalities can be found, for instance, in Cavalieri et al. (1996). The different processors, named ECUs (Electronic Component Units), are interconnected with different fieldbuses such as CAN (Control Area Network) and VAN (Vehicle Area Network) (ISO, 1994a,b,c).

One of the recent efforts from car manufacturers and ECU suppliers is the definition of a common operating system called OSEK/VDX (OSEK, 1997). The use of this operating system by all ECUs in the future will enhance the interoperability and the reusability of the application code. Such an approach drastically reduces the software development costs.

9.3.2 Hardware and software architecture

The specific application that we are going to study is a modified version derived from an actual one embedded in the cars of PSA (Peugeot-Citroën Automobile Corp.)

(Richard et al., 2001). The application is composed of different nodes interconnected by one CAN network and one VAN network. Prominent European fieldbus examples targeted for automotive applications are CAN and VAN. These fieldbuses have to strive to respect deterministic response times. Both correspond to the medium access control (MAC) protocol, based on the CSMA/CA (Carrier Sense Multiple Access / Collision Avoidance) protocol.

CAN is a well-known network; it was presented in Section 6.4.3. We just recall that the maximum message length calculation should include the worst-case bit stuffing number and the 3 bits of IFS (Inter Frame Space). For a message of n bytes, this length is given by $47 + 8n + \lfloor (34 + 8n)/4 \rfloor$, where $\lfloor x \rfloor$ $(x \geq 0)$ denotes the largest integer less than or equal to x.

Hardware architecture

The complete application is composed of nine ECUs (or nodes) interconnected by one CAN network and one VAN network as shown by Figure 9.18. They are: Engine controller, Automatic Gear Box, Anti-lock Brake System/Vehicle Dynamic Control, Suspension controller, Wheel Angle Sensor/Dynamic Headlamp Corrector, Bodywork, and three other specialized units dedicated to passenger comfort functions (Table 9.5). To make understanding of the rest of this chapter easier, Table 9.5 links a number to each main ECU of the application.

CAN is used for real-time control systems such as engine control and anti-lock brakes whereas VAN is used in bodywork for interconnecting ECUs without tight time-critical constraints. The bodywork computer (node 6) ensures the gateway function between CAN and VAN. The need for exchanges between these two networks is obvious. For example, for displaying the vehicle speed, a dashboard in the bodywork needs information from the ECU connected to CAN; when requiring more power, the engine controller can send a signal to the air condition controller to inhibit air conditioning. And this latter is also under real-time constraints.

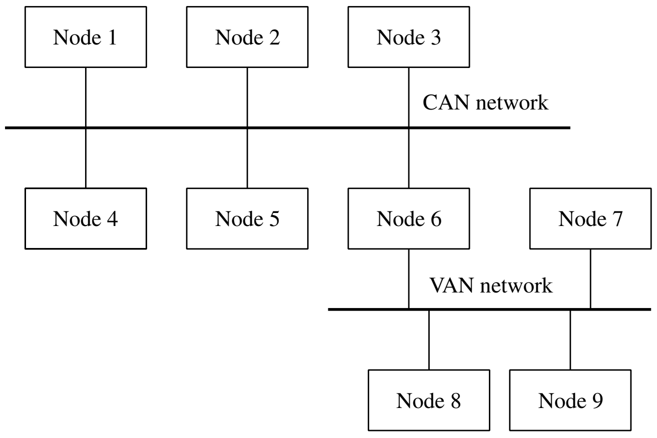

Figure 9.18 Hardware architecture of the automotive application

Table 9.5 Functions of the main nodes of the distributed automotive application

Node	Function
Node 1	Engine controller
Node 2	Automatic gear box
Node 3	Anti-locking brake system/Vehicle dynamic control
Node 4	Wheel angle sensor/Dynamic headlamp corrector
Node 5	Suspension controller
Node 6	Bodywork (between CAN and VAN networks)
Nodes 7, 8, 9	Passenger comfort functions

9.3.3 Software architecture

The entire application has 44 tasks distributed among the different processors and 19 messages conveyed by the two networks. More precisely, the critical part of the application uses the CAN network, and has 31 tasks and 12 messages, whereas the non-critical part uses the VAN network and has 13 tasks and 7 messages. In order to simplify the study of this complex example, we limit the temporal analysis to the nodes connected to the CAN network, that is to say to the critical real-time part of the application. So the corresponding software architecture of the automotive application is given in Figure 9.19.

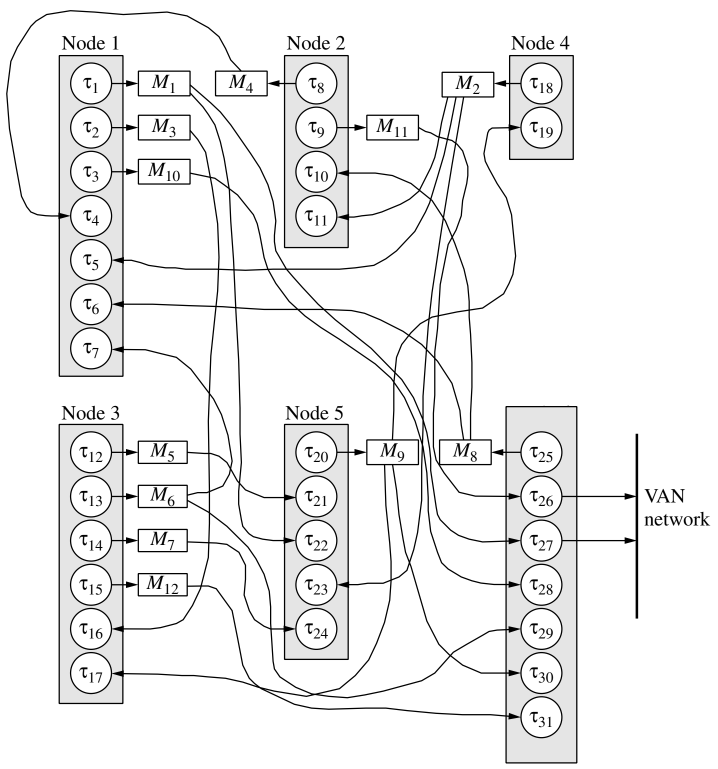

Figure 9.19 Software architecture of the automotive application restricted to the critical real-time communications on the CAN network

We now present the model of the application used in the temporal analysis. We describe all the tasks on each processor, and all the messages on the CAN network. Each task is defined by (r_i, C_i, D_i, T_i) parameters, defined in Chapter 1. In this application, the arrival time r_i is null for any task. Moreover, the tasks are periodic and deadlines are equal to periods. The timing requirements are summarized in Table 9.6, for each processor.

For evaluating the implementation, we assume that all ECUs run under OSEK/VDX OS. Moreover, the actual complex task description has been split into many small basic tasks (in an OSEK/VDX sense). Table 9.7 presents the communication data between tasks for all the messages. The period of a message is trivially inherited from the sender of this message and its deadline is inherited from the task it is addressed to. In our case, deadlines are equal to periods, so deadlines can also be inherited from the sender of the message. For a message, the transmission delay is computed as a function of the number of bytes according to the formula recalled in Section 9.3.2. The messages are listed by priority order.

Table 9.6 Task parameters of the distributed automotive application

Node	Task	Computation time (ms)	Period (ms)
Node 1	τ_1	2	10
	τ_2	2	20
	τ_3	2	100
	τ_4	2	15
	τ_5	2	14
	τ_6	2	50
	τ_7	2	40
Node 2	τ_8	2	15
	τ_9	2	50
	τ_{10}	2	50
	τ_{11}	2	14
Node 3	τ_{12}	1	20
	τ_{13}	2	40
	τ_{14}	1	15
	τ_{15}	2	100
	τ_{16}	1	20
	τ_{17}	2	20
Node 4	τ_{18}	4	14
	τ_{19}	4	20
Node 5	τ_{20}	1	20
	τ_{21}	1	20
	τ_{22}	1	10
	τ_{23}	2	14
	τ_{24}	2	15
Node 6	τ_{25}	2	50
	τ_{26}	2	50
	τ_{27}	2	10
	τ_{28}	2	100
	τ_{29}	2	40
	τ_{30}	2	20
	τ_{31}	2	100

Table 9.7 Message characteristics of the distributed automotive application. The transmission delay computation is based on CAN with a bit rate of 250 Kbit/s

Message	Sender task	Receiver task	Number of bytes	Size (bits)	Propagation delay (ms)	Period (ms)	Priority
M_1	τ_1	τ_{27}, τ_{22}	8	130	0.5078	10	12
M_2	τ_{18}	$\tau_{11}, \tau_5, \tau_{23}$	3	82	0.3203	14	11
M_3	τ_2	τ_{16}	3	82	0.3203	20	10
M_4	τ_8	τ_4	2	73	0.2852	15	9
M_5	τ_{12}	τ_{21}	5	101	0.3945	20	8
M_6	τ_{13}	τ_7, τ_{29}	5	101	0.3945	40	7
M_7	τ_{14}	τ_{24}	4	92	0.3594	15	6
M_8	τ_{25}	τ_{10}, τ_6	5	101	0.3945	50	5
M_9	τ_{20}	$\tau_{19}, \tau_{17}, \tau_{30}$	4	92	0.3594	20	4
M_{10}	τ_3	τ_{28}	7	121	0.4727	100	3
M_{11}	τ_9	τ_{26}	5	101	0.3945	50	2
M_{12}	τ_{15}	τ_{31}	1	63	0.2461	100	1

9.3.4 Detailed temporal analysis

Temporal analysis of nodes considered as independent

As a first step of the temporal analysis, we ignore the communications between nodes. The scheduling analysis of the different ECUs is quite easy because the defined tasks are considered independent. So we can calculate the processor utilization factor U and the scheduling period H for each processor, as defined in Chapter 1 (Table 9.8).

Moreover, on each node, we have a real-time system composed of independent, preemptive periodic tasks that are in phase and have deadlines equal to their respective periods. If we assign the fixed priorities according to the rate monotonic algorithm (tasks with shorter periods have higher priorities), we can check the schedulability of the node only by comparing its utilization factor U to the upper bound of the processor utilization factor determined by Liu and Layland (1973) (see condition (2.12) in Chapter 2). Notice that this schedulability condition is sufficient to guarantee the feasibility of the real-time system, but it is not necessary. This means that, if a task set has a processor utilization factor greater than the limit, we have to carry on and use other conditions for the schedulability or simulate the task execution over the scheduling period.

Table 9.8 Basic temporal parameters of each node

Node	U	Upper bound (Liu and Layland, 1973)	H (ms)
Node 1	0.686	0.729	4200
Node 2	0.356	0.757	1050
Node 3	0.337	0.735	600
Node 4	0.486	0.828	140
Node 5	0.476	0.743	420
Node 6	0.470	0.729	200

From the results presented in Table 9.8, we conclude that each node is weakly loaded, less than 69% for the highest processor utilization factor. Therefore all the task sets, if considered independent, satisfy the sufficient condition of Liu and Layland (1973) and the fixed-priority assignment, according to the rate monotonic algorithm, can schedule these task sets. Neither further analysis nor simulation over a scheduling period is necessary to prove the schedulability of the application.

Anyway, in order to illustrate the scheduling analysis with priority fixed according to the RM algorithm, we present the execution sequences of tasks of two nodes and display the emission and reception of messages by the different tasks. It is assumed hereafter that the messages are sent at the end of the tasks and received at their beginning. Recall that we do not consider message communications. The simulations have been plotted only over a tiny part of the scheduling period: 20 ms. Figure 9.20 deals with the execution of node 3, and Figure 9.21 corresponds to the execution sequence of node 5.

To summarize this section, each node of this automotive application, considered alone, can easily schedule tasks by using a fixed-priority assignment, according to the rate monotonic algorithm. We can widen this result to the case of message communications, if we consider a slack synchronization between tasks. This case occurs when a kind of 'blackboard' is used as a communication technique in a real-time system: the sender writes or over-writes the message at each emission and the writer always reads the last message (sometimes a message may be lost).

The cost of reading and writing a message is included in the task computation times. The access to the 'blackboard' is supposed to be atomic (or at least mutually exclusive, or best, according to a reader–writer synchronization pattern). The slack synchronization means that if the kth value of a message is not available, the receiving task can perform its computation with the previous $(k-1)$th value of the message.

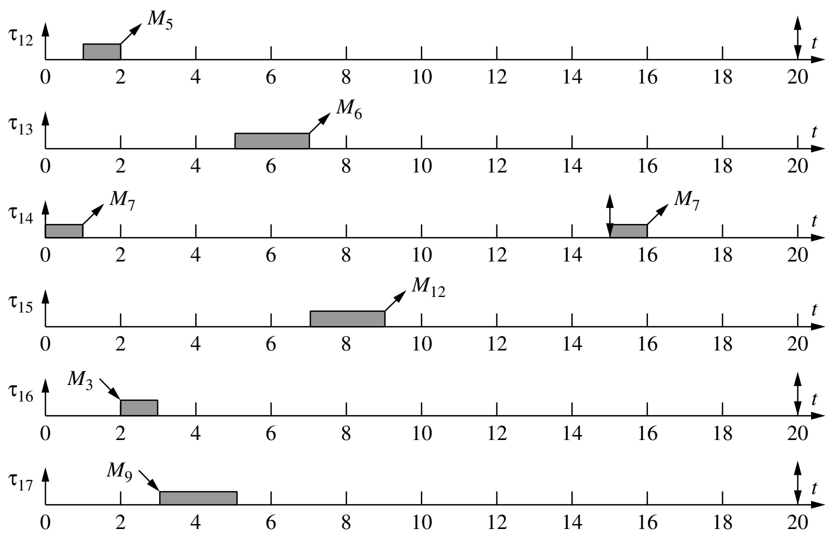

Figure 9.20 Execution sequence of the tasks of node 3 with fixed-priority assignment according to the rate monotonic algorithm

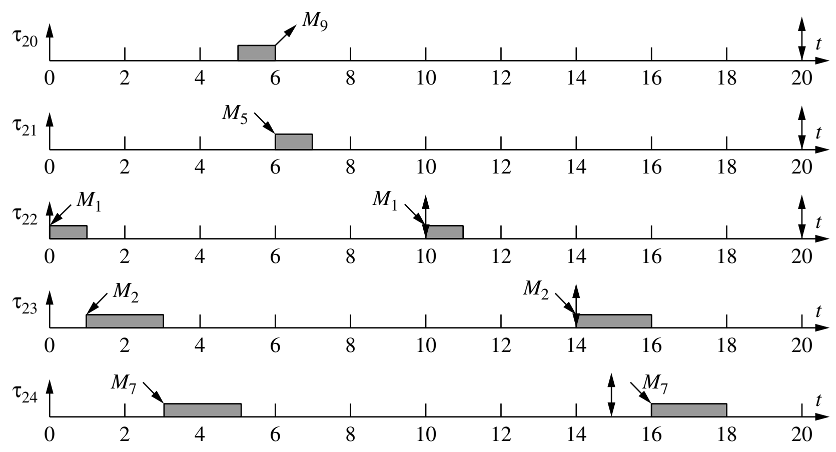

Figure 9.21 Execution sequence of the tasks of node 5 with fixed-priority assignment according to the rate monotonic algorithm

Temporal analysis of the distributed application

When distributed systems are considered with tight synchronizations, the tasks are mutually dependent because they exchange messages. The analysis must take into account the synchronization protocol of the communicating tasks, and also the scheduling policies of the messages through the network. The network is a shared resource for each communicating task. For example, between the previously analysed nodes (3 and 5), we have three communication relationships:

- τ_{12} (node 3) sends message M_5 to τ_{21} (node 5);
- τ_{14} (node 3) sends message M_7 to τ_{24} (node 5);
- τ_{20} (node 5) sends message M_9 to τ_{17} (node 3).

In distributed systems, a dysfunction can occur if a message is sent after the receiver task execution. This fact is illustrated in Figures 9.20 and 9.21: task τ_{20} sends the message M_9 to task τ_{17} after this task has completed its execution. A simple solution to this problem lies in the use of two-place memory buffers related to each communication message. The message emitted at the kth period is used at period $k + 1$. The first request of τ_{17} must be able to use a dummy message. This is possible if the calculation of task τ_{17} remains valid within this message time lag. But this solution needs hardware and/or software changes in order to manage this specific buffer. So we want to stay in a classical real-time system environment.
A solution can be found following two methods:

- Method 1 assumes the use of synchronization primitives (e.g. lock and unlock semaphores) in the task code in order to produce the right sequence with a fixed-priority assignment (this solution is used in the rolling mill signal acquisition presented as the first case study).

- Method 2 modifies the task parameter r_i, keeping the initial priority in accordance with the method presented in Section 3.1.

In the first method, the schedulability analysis is based on the response time analysis method for distributed systems called holistic analysis (Tindell and Clark, 1994). This is an *a priori* analysis for distributed systems where the delays for messages being sent between processors must be accurately bounded. In this modelling, the network is considered as a non-preemptive processor. When a message arrives at a destination processor, the receiver task is released, and can then read the message. We can say that the receiver task inherits a release jitter J_r in the same way that a message inherits release jitter from the sender task corresponding to its worst-case response time $TR_s : J_r = TR_s + d_{\mathrm{CAN}}$ where d_{CAN} is the transmission delay of the message (Section 6.4.3 gives an example of computation of d_{CAN} delay). A solution to the global problem can be found by establishing all the scheduling equations (worst-case response time for each task on every node and the release jitters induced by the message communication). Then it is possible to solve the problem and find the maximum execution time bounds, which must be lower than deadlines. We can summarize by saying that this method validates the application by evaluating the worst-case response times of all the tasks of the distributed application. With synchronization primitives, the dysfunction, explained above in Figures 9.20 and 9.21, cannot occur because when task τ_{17} starts running, it is blocked waiting for the message M_9. So this method permits us to validate this application with the RM priority assignment (Richard et al., 2001).

In the second method, the release time of each task receiving a message is modified in order to take into account the execution time of the sender task and the message communication delay. These two delays correspond to the waiting times of the sender task (respectively message) due to higher priority tasks of the same node (respectively higher priority messages in the network). It is of an utmost importance to integrate in calculations the occurrence number of higher priority tasks (respectively higher priority messages) arriving during the period of the sender task (respectively message). An example of these results is shown in Table 9.9 only for the nodes 3 and 5 corresponding to the previously analysed sequences. In Figures 9.22 and 9.23, it is quite clear that task τ_{20} sends the message M_9 to task τ_{17} before its execution. It is also obvious that the

Table 9.9 Modifications of task parameters of the distributed automotive application according to the second method

Node	Task	Period	RM priority	Modified r_i
Node 3	τ_{12}	20	5	0
	τ_{13}	40	2	0
	τ_{14}	15	6	0
	τ_{15}	100	1	0
	τ_{16}	20	4	10
	τ_{17}	20	3	9
Node 5	τ_{20}	20	2	0
	τ_{21}	20	1	5
	τ_{22}	10	5	3
	τ_{23}	14	4	5
	τ_{24}	15	3	3

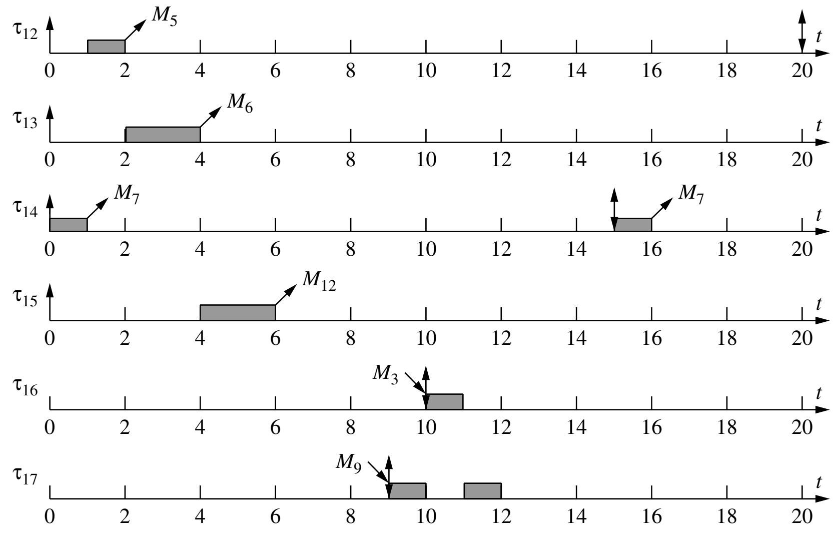

Figure 9.22 Execution sequence of the tasks of node 3 with the RM priority assignment and modified release times (see Table 9.9) in the case of the second method

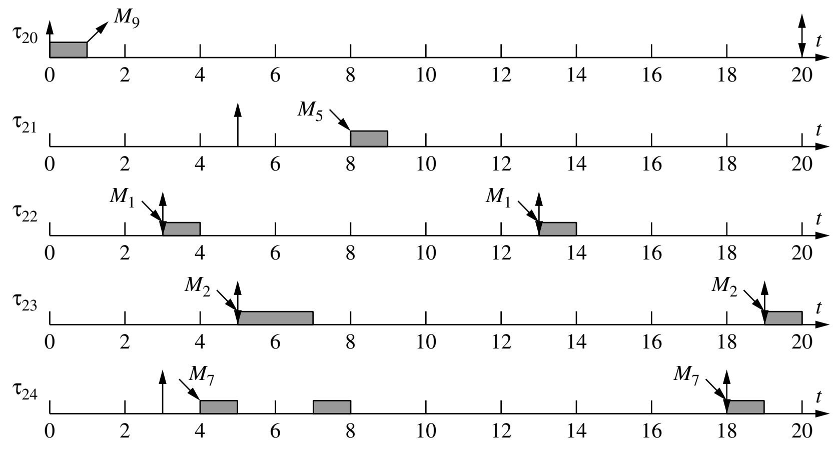

Figure 9.23 Execution sequence of the tasks of node 5 with the RM priority assignment and modified release times (see Table 9.9) in the case of the second method

whole application remains schedulable since only the release times have been changed; the processor utilization factor and the deadlines are the same. In this context, the system of independent, preemptive tasks with relative deadlines equal to their respective periods, on each node, is schedulable with an RM priority assignment because the schedulability condition does not depend on the release times.