Kernel support for embedded reactive systems

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF STELLENBOSCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
MC Ackerman
October 1993

Supervised by: Mr P.J.A. de Villiers

Stellenbosch University http://scholar.sun.ac.za

Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: **Date:**

Stellenbosch University http://scholar.sun.ac.za

Abstract

Reactive systems are event driven state machines which usually do not terminate, but remain in perpetual interaction with their environment. Such systems usually interact with devices which introduce a high degree of concurrency and some real time constraints to the system. Because of the concurrent nature of reactive systems they are commonly implemented as communicating concurrent processes on one or more processors. Jeffay introduces a design paradigm which requires consumer processes to consume messages faster than they are produced by producer processes. If this is guaranteed, the real time constraints of such a system are always met, and the correctness of the process interaction is guaranteed in terms of the message passing semantics. I developed the ESE kernel, which supports Jeffay systems by providing lightweight processes which communicate over asynchronous channels. Processes are scheduled non-preemptively according to the earliest deadline first policy when they have messages pending on their input channels. The Jeffay design method and the ESE kernel have been found to be highly suitable to implement embedded reactive systems. The general requirements of embedded reactive systems, and kernel support required by such systems, are discussed.

Stellenbosch University http://scholar.sun.ac.za

Opsomming

Reaktiewe stelsels is toestandsoutomate wat aangedryf word deur gebeure in hul omgewing. So 'n stelsel termineer gewoonlik nie, maar bly in 'n voortdurende wisselwerking met toestelle in sy omgewing. Toestelle in die omgewing van 'n reaktiewe stelsel veroorsaak in die algemeen 'n hoë mate van gelyklopendheid in die stelsel, en plaas gewoonlik sekere intydse beperkings op die stelsel. Gelyklopende stelsels word gewoonlik as stelsels van kommunikerende prosesse geïmplementeer op een of meer prosessors. Jeffay beskryf 'n ontwerpsmetodologie waarvolgens die ontvanger van boodskappe hulle vinniger moet verwerk as wat die sender hulle kan stuur. Indien hierdie gedrag tussen alle pare kommunikerende prosesse gewaarborg kan word, sal die stelsel altyd sy intydse beperkings gehoorsaam, en word die korrektheid van interaksies tussen prosesse deur die semantiek van die boodskapwisseling gewaarborg. Die "ESE" bedryfstelselkern wat ek ontwikkel het, ondersteun stelsels wat ontwerp en geïmplementeer word volgens Jeffay se metode. Prosesse kommunikeer oor asinkrone kanale, en die ontvanger van die boodskap met die vroegste keertyd word altyd eerste geskeduleer. Jeffay se ontwerpsmetode en die "ESE" kern blyk in die praktyk baie geskik te wees vir reaktiewe stelsels wat as substelsels van groter stelsels uitvoer. Die vereistes van reaktiewe substelsels, en die kernondersteuning wat daarvoor nodig is, word bespreek.

Stellenbosch University http://scholar.sun.ac.za

Acknowledgements

My thanks to:

- **my supervisor, Pieter de Villiers, for his ideas, advice and guidance;**
- **Prof Tony Krzesinski for goading me whenever he saw me, none the less remaining patient; and**
- **Charlotte Ackerman for having unwavering faith in my completion of this thesis, constantly supporting me, and for cheerfully putting up with the disruption and stress caused by it.**

Stellenbosch University http://scholar.sun.ac.za

Contents

Abstract iii

Opsomming iv

Acknowledgements v

1 Introduction 1

1.1 The subject of this thesis 2

1.2 Outline of this thesis 2

2 Background 3

2.1 Scheduling algorithms for multiprogramming in a hard-real-time environment 3

2.1.1 Rate monotonic scheduling 4

2.1.2 Deadline driven scheduling 5

2.1.3 Hybrid scheduling algorithms 6

2.1.4 Run-time vs. pre-run-time scheduling 7

2.2 Specification and analysis techniques for real-time systems 8

2.3 Operating systems for real-time systems 10

2.4 The real-time producer/consumer paradigm 12

2.5 Discussion 12

Stellenbosch University http://scholar.sun.ac.za

3 The real-time producer/consumer (RT/PC) paradigm 15
3.1 Introduction . . . 15
3.2 System components and graphical notation . . . 16
3.2.1 Processes and channels . . . 18
3.2.2 Input and output devices . . . 19
3.2.3 Mutual exclusion regions and data repositories . . . 20
3.2.4 Well-formedness of design graphs . . . 21
3.3 Message passing semantics . . . 22
3.3.1 Message transmission rates . . . 22
3.3.2 Restrictions on process construction . . . 22
3.4 Calculating channel input rates . . . 23
3.4.1 Calculating message output rates . . . 23
3.4.2 Calculating message input rates . . . 24
3.5 Scheduling results . . . 24
3.6 Implementing RT/PC designs . . . 26

4 The ESE Kernel 29
4.1 Embedded Reactive Systems . . . 29
4.2 Events . . . 32
4.3 Preemptive and Non-preemptive Scheduling . . . 34
4.3.1 Support for RT/PC . . . 34
4.3.2 Scheduling efficiency . . . 34
4.3.3 Support for state machines . . . 35
4.3.4 Design decision . . . 35
4.4 Periodic vs Sporadic Processes . . . 36

Stellenbosch University http://scholar.sun.ac.za

4.5 Data Repositories and Synchronous Message Passing 38
4.6 Synchronous vs Asynchronous Channels 38
4.7 Mutual Exclusion Regions 39
4.8 Interrupts and Events 40
4.9 Timers 43
4.10 Implementation of Scheduling 44
4.10.1 System clock granularity 48
4.11 Performance of ESE 50

5 Developing Reactive Systems 54
5.1 Introduction 54
5.2 X.25 case study — requirements definition 54
5.3 System design 58
5.3.1 Block design of X.25 system 58
5.3.2 Process decomposition 60
5.4 Design and analysis of RT/PC systems 61
5.4.1 Realisability of a design graph 61
5.4.2 Realisability of the X.25 design 62
5.4.3 Implementing the design graph 67
5.4.4 Implementation mapping of the X.25 example 70
5.4.5 Viability analysis 73
5.4.6 Viability of the X.25 example 74
5.5 RT/PC and ESE in practice — some observations 79
5.5.1 The area of applicability of ESE 81
5.5.2 The effort of designing for RT/PC and ESE 81

Stellenbosch University http://scholar.sun.ac.za

5.5.3 Realisability and viability analysis 81

5.5.4 Performance of ESE implementations 82

5.6 Summary of method 82

5.6.1 Requirements definition 82

5.6.2 System design 82

5.6.3 RT/PC design 83

5.6.4 Realisability analysis 83

5.6.5 Implementation mapping 83

5.6.6 Viability analysis 83

5.6.7 Implementation 84

6 Conclusion **85**

A ESE interface types and commands **89**

A.1 Types 89

A.1.1 Alarm 89

A.1.2 AsyncChannel 89

A.1.3 InputPort 90

A.1.4 Mailbox 90

A.1.5 Message 90

A.1.6 Name 90

A.1.7 Octet 90

A.1.8 Process 90

A.1.9 Thread 91

A.1.10 Time 91

A.1.11 Timer 91

Stellenbosch University http://scholar.sun.ac.za

A.1.12 UserRef **91**

A.2 CreateProcess **91**

A.3 CreateAsyncChannel **92**

A.4 CreateMailbox **92**

A.5 CreateInputPort **92**

A.6 StartSystem **93**

A.7 SendMessageOnAsyncChannel **93**

A.8 PutIntoMailbox **93**

A.9 SignalInputPort **94**

A.10 Receive **94**

A.11 SetTimer **95**

A.12 SetAlarm **95**

A.13 StopTimer **95**

A.14 StopAlarm **96**

Stellenbosch University http://scholar.sun.ac.za

List of Tables

1 Task Switching Overhead 52

2 Timer Processing Overhead 53

3 Viability Condition 1 — Maximum Channel Rate 77

4 Viability Condition 1 — Adjusted Channel Rate 78

Stellenbosch University http://scholar.sun.ac.za

List of Figures

1 Real-time producer/consumer paradigm 16
2 An RT/PC design graph 17
3 Schema for an RT/PC process 19
4 A schema for an RT/PC data repository 21
5 Message transmission rate function 24
6 Input rate definition for MP/SC processes 24
7 Reactive system 30
8 Information frame format 41
9 Fast Interrupting Device: Unrealisable System 41
10 Fast Interrupting Device: Realisable System 42
11 Scheduling with course clock ticks 50
12 Scheduling with inserted fine grain clock ticks 51
13 Task switching cost of channels 52
14 Block diagram of iX.25 hardware 56
15 OSI/X.25/implementation correspondence 59
16 Protocol machine implementations 60
17 Nested cycles 62
18 RT/PC design of X.25 63

Stellenbosch University http://scholar.sun.ac.za

19 Reduced, annotated RT/PC design for X.25 65

20 Refined model of X.25 for implementation 71

21 Input File for Viability Condition 2 Checker 79

22 Output File of Viability Condition 2 Checker 80

23 Output File of Viability Condition 2 Checker: Continued 80

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

The proliferation of cheap microprocessors over the past decade has led to a trend of distributing intelligence in a computer system, and to embed intelligent applications in subsystems. Integrated single chip and chip set solutions has made it cheap and easy to produce relatively powerful embedded and stand-alone platforms. These platforms are used in a variety of applications: from bus expansion cards, such as intelligent communications controllers, to stand-alone devices such as process controllers and hand held computers.

The intelligence available in embedded platforms, today, facilitates very powerful embedded applications. At the same time, economical considerations often dictate that commercial and industrial systems be built on the cheapest possible platform. This usually results in a less powerful processor, and less memory, than the software engineer would have preferred. In contrast, operating systems are becoming more complex, and small embedded platforms can often not support their resource requirements.

An embedded system usually interacts with devices in its environment. Such devices often introduce real-time constraints on the device drivers controlling them. By real-time we mean that the correctness of the system depends on its processing of certain events within a known interval, in other words: before a deadline. In some real-time systems a missed deadline may have catastrophic results. Such systems are usually called *hard real-time* systems. Examples of hard real-time systems are on-board flight control systems, medical control systems, etc. Other real-time systems can recover from missed deadlines, and are know as *soft real-time* systems.

Pnueli [45] describes two basically different views of computerised systems. The first, called *transformational systems*, refers to systems which accept their inputs at the beginning of

Stellenbosch University http://scholar.sun.ac.za

their operation, and yield their outputs at termination. The second, called ***reactive systems***, are systems which typically don't terminate, but remain in perpetual interaction with their environment. Reactive systems are not used to obtain a final result, but rather to control ongoing processes. Embedded systems are often reactive systems.

1.1 The subject of this thesis

Small commercial and industrial embedded systems are often built on uniprocessor platforms. The processor and RAM resources of these platforms usually do not permit the use of sophisticated state of the art operating systems, but their reactive nature is best supported as a system of communicating processes. Furthermore, these systems interact with devices, and therefore often have real-time constraints.

The purpose of this study is to find a design and implementation methodology which is suitable for the type of systems described above. The implementation strategy should ensure that the temporal correctness of a design is preserved in its implementation.

1.2 Outline of this thesis

This chapter introduces the concepts of real-time, embedded and reactive systems, and describes the subject of the thesis.

Chapter 2 describes the state of the art in real-time scheduling techniques, operating systems and design and analysis methods. One method is chosen for an experimental implementation.

Chapter 3 summarises the *RT/PC paradigm*: a design and analysis method for real-time systems. The paradigm, its graphical design notations, scheduling results, and realisability analysis are described.

Chapter 4 examines the design, features and performance of the ESE kernel, which supports implementations of RT/PC designs.

Chapter 5 describes the design, analysis and implementation of a real system. The RT/PC method is used to design the system, which is then implemented on an embedded ESE kernel.

Finally, chapter 6 discusses the conclusions drawn from using RT/PC and the ESE kernel in practice.

Stellenbosch University http://scholar.sun.ac.za

Chapter 2

Background

In chapter 1, the goal of this study was stated to be an investigation into a design and implementation methodology, which is appropriate for uniprocessor based embedded systems. We also saw that the concepts of *real-time* and *reactive systems* are pertinent to this study. The implementation of embedded systems, on the target platforms in which we are interested, will require specific kernel support for reactive, real-time systems. To understand the background to this study we must therefore survey the literature pertaining to real-time scheduling, reactive and embedded systems, analysis and modeling techniques for real-time systems, and operating system support for real-time systems.

2.1 Scheduling algorithms for multiprogramming in a hard-real-time environment

This was also the title of a landmark paper by Liu and Layland [39] in 1973, in which the authors compared two preemptive priority scheduling techniques: a fixed priority and a dynamic priority scheduling algorithm. In the fixed priority algorithm, priorities are assigned to tasks according to the *rate monotonic* priority assignment method. In the dynamic priority algorithm, the priority of a task is determined by its deadline. Liu and Layland's results are for *periodic tasks* scheduled on a *uniprocessor*, and *tasks are independent*. That is, requests for a certain task do not depend on the initiation or completion of requests for other tasks.

Stellenbosch University http://scholar.sun.ac.za

2.1.1 Rate monotonic scheduling

Let $\tau = \{T_1, T_2, \ldots, T_m\}$ be a set of m periodic tasks. Let each task $T_i = (c_i, p_i)$ be characterised by its execution cost, c_i, and its request period, p_i. The *request rate* r_i of a task is the reciprocal of its request period, and represents the frequency at which the task must be scheduled. A *rate monotonic priority assignment* to τ implies that for all $T_i, T_j \in \tau$, the priority assigned to T_i is higher than the priority assigned to T_j iff $p_i < p_j$. Liu and Layland proved that the rate monotonic priority assignment is optimal in the sense that if a feasible priority assignment exists for a task set, the rate monotonic priority assignment is feasible for that task set.

Definition 1 *A* **feasible set of tasks** *can be scheduled on a uniprocessor in such a way that every execution request of every task is guaranteed to have completed execution at or before its deadline.*

Definition 2 *An* **optimal scheduling discipline** *can correctly schedule any task set that is feasible.*

Liu and Layland defined a *processor utilisation factor*, $U_i = \frac{c_i}{p_i}$, for tasks. The total processor utilisation for a set of tasks is given by

$$U_\tau = \sum_{i=1}^{m} \frac{c_i}{p_i}$$

For a given fixed priority scheduling algorithm, the *least upper bound* of the utilisation factor is the minimum of the utilisation factors over all task sets that fully utilise the processor. If the utilisation factor of a task set is below this bound, there exists a feasible priority assignment for it. If the utilisation factor of a task set is above the least upper bound, it is feasible only if the request periods of its tasks are related suitably. Liu and Layland proved that the least upper bound of the utilisation factor for a fixed priority assignment is

$$U \leq m(2^{\frac{1}{m}} - 1),$$

and for large m

$$U \simeq ln2 = 0.69$$

Liu and Layland's original work has been extended significantly. Dhall and Liu [14], for example, studied the performance of the *rate monotonic first fit* and *rate monotonic next*

Stellenbosch University http://scholar.sun.ac.za

fit scheduling algorithms for multiprocessor systems. Sha and Goodenough [50] discuss *rate monotonic analysis* and give necessary and sufficient feasibility conditions for task sets which exceed the least upper bound for fixed priority processor utilisation. They also discuss how *aperiodic* tasks can be supported, and give a sufficient feasibility condition for tasks which synchronise.

When tasks which are scheduled according to the rate monotonic algorithm synchronise, i.e. the requests for a certain task may depend on the initiation or completion of requests for other tasks, a situation called *priority inversion* arises. Priority inversion occurs when tasks with different priorities come into contention for a shared resource, and a higher priority task is blocked by a lower priority task. Davari and Sha [12] examine common sources of priority inversion and outline a number of solutions to the problem.

The rate monotonic scheduling algorithm is popular today for scheduling sets of tasks with hard real-time constraints [36, 60, 6], and *rate monotonic analysis* can be used in conjunction with other scheduling algorithms [6].

2.1.2 Deadline driven scheduling

The effect of using a deadline driven scheduling algorithm is to dynamically adjust the priorities of tasks according to their deadlines. For a periodic task set Liu and Layland [39] derived the following necessary and sufficient feasibility condition for a deadline driven scheduling algorithm:

> *For a given set of m periodic tasks, the deadline driven scheduling algorithm is feasible if and only if*
> $$\sum_{i=1}^{m} \frac{c_i}{p_i} \leq 1$$

This means that the least upper bound for processor utilisation is uniformly 100%. Two deadline driven algorithms are commonly used:

1. The *least slack first* algorithm selects the task with the least amount of slack before its deadline. The difference between the remainder of a task's period (before its deadline) and its computational cost is known as its slack. This algorithm has been proved to be optimal for uniprocessor systems [37].

Stellenbosch University http://scholar.sun.ac.za

2. The *earliest deadline first* algorithm selects the task with the earliest deadline to be executed next. This algorithm has been proved to be optimal for both uniprocessor and multiprocessor systems [37].

Deadline driven scheduling is optimal for both periodic and aperiodic tasks [8, 28, 9, 39]. Jeffay [28] proved necessary and sufficient feasibility conditions for preemptive and non-preemptive *earliest deadline first* scheduling of periodic and aperiodic tasks on a uniprocessor.

2.1.3 Hybrid scheduling algorithms

Hybrids of fixed and dynamic priority algorithms are used for many reasons. Liu and Layland [39] observed that interrupt controllers impose a static priority assignment on interrupt events, even when deadline driven scheduling is used. Processor utilisation cannot be 100% for mixed scheduling algorithms, but on average can be much higher than the least upper bound of the processor utilisation for fixed priority schedulers [39].

In rate monotonic schedulers the priority of critical tasks may be temporarily adjusted to accommodate transient overload conditions [50]. Miller [43] cites the necessity to have functional prioritisation in the presence of deadline scheduling. Schwan and Zhou [49] describe a preemptive earliest deadline first scheduler which performs dynamic feasibility analysis when a task request is received. Since all deadlines cannot be guaranteed in this environment a secondary metric, called *criticalness* is used to determine which task is scheduled when two tasks cannot simultaneously meet their deadlines.

Because deadlines must be met in the worst case behaviour or hard real-time systems, analysis of hard real-time systems is usually based on the theoretical worst case behaviour of the system. The average case behaviour may differ significantly from the worst case, and if schedules are computed based on worst case, under utilisation of resources may result. Hardware and software monitors may be used to measure the real-time behaviour of real-time systems, and adapt the scheduling policy accordingly. Haban and Shin [19] describe issues relating to real-time monitoring, and their experience with using a hardware monitor, while Kenny and Lin [30] describe software monitoring in the Flex language and C++. Some real-time operating systems include monitors to improve average system performance [58].

Stellenbosch University http://scholar.sun.ac.za

2.1.4 Run-time vs. pre-run-time scheduling

Because it is imperative for a hard real-time system to meet its deadlines, task schedules for hard real-time systems are sometimes computed before run-time. A very simple run-time scheduler is then used to select the next task from the pre-calculated schedule. Scheduling a task set before run-time can reduce the run-time scheduling overhead, because feasibility analysis is not done when the task is scheduled, and it ensures that a feasible schedule exists before the task set is run. This approach is called *pre-run-time scheduling*, or *static scheduling*. Another approach is *run-time* or *dynamic scheduling*, where the schedule for a task set is calculated as task requests arrive. Except for a closed task set, run-time scheduling implies that feasibility analysis is required when a task request arrives. Burns [5] reviews current scheduling techniques for hard real-time systems.

Xu and Parnas [65] argue that pre-run-time scheduling is essential to guarantee that a system will meet all its timing constraints. In their experience most tasks in a hard real-time environment are periodic, and any asynchronous (aperiodic) task can be translated into an equivalent periodic task. For instance, let $T_a = (c_a, p_a, d_a)$ be an asynchronous task, with c_a the worst case computational cost, p_a the minimum period between successive requests for T_a, and d_a its deadline. T_a can be translated [65] into an equivalent periodic task, $T_p = (r_p, c_p, p_p, d_p)$, with release time $r_p = 0$, computational cost $c_p = c_a$, deadline $d_a \geq d_p \geq c_p$, period $p_p \leq min(d_a - d_p + 1, p_a)$. In this way it is possible to schedule asynchronous tasks using pre-run-time scheduling. Xu and Parnas [64] give a pre-run-time scheduling algorithm for processes with release times, deadlines, precedence and exclusion relations. Shepard [53] gives a pre-run-time *multiprocessor* scheduling algorithm for the same class of processes. The original algorithm by Xu and Parnas was later also extended by Xu [63] to include multiprocessor architectures.

Schwan and Zhou [49] argue that dynamic hard real-time systems, in their experience, have to cope with unexpected sporadic (aperiodic) processes with hard deadlines. A schedule of periodic tasks computed pre-run-time cannot adapt adequately to frequent unexpected task requests with hard deadlines. Such task sets should be scheduled dynamically, but this means that a feasible executing task set may be rendered infeasible by the arrival of a new task request. Run-time feasibility analysis is therefore required before a new task request can be scheduled. Schwan and Zhou present a preemptive earliest deadline first algorithm with run-time feasibility analysis. A secondary metric, *criticalness* is used to determine which tasks are scheduled when the arrival of a new task request causes an infeasible schedule. When a new task request arrives, only those tasks which are in contention with the new task are

Stellenbosch University http://scholar.sun.ac.za

rescheduled. This keeps the run-time overhead of the scheduler low.

Chetto and Chetto [8, 9] use a combination of static and dynamic scheduling for fault tolerant hard real-time systems. For each task it is possible to schedule either a primary or an alternative process. If a primary process fails, an alternative may be used to recover from the failure. The primary processes are scheduled pre-run-time, and alternatives sporadically when a primary fails.

2.2 Specification and analysis techniques for real-time systems

Embedded systems are hard to develop and test for a number of reasons. Because the hardware platform of an embedded system is usually customised to the specific purpose of the system, the usual development aids such as character I/O and interactive debuggers can often not be used. Because an embedded system is often a subsystem of a larger system, it may not be possible to test in isolation, and more seriously, it may have real-time constraints which are hard or impossible to create outside the actual target environment. In some critical hard real-time systems, such as weapons systems, the problem with testing and correcting errors under operational conditions is self evident.

De Roever [13] urges the development of a design specification which satisfies a "full requirements specification", followed by implementation according to fail-safe, fault tolerant techniques. Rigorous specification development is only possible when a formal specification method is used, and preferably automated. Fault tolerant techniques are becoming more common [8, 9, 33, 37, 41, 54], and many analysis techniques have appeared. Although the definition of the term *real-time* depends on the perspective of the practitioner using it, there are clear categories and trends. The explicit use of time in the specification of real-time systems evokes a wide range of responses. Turski [59] advocates total avoidance of the explicit use of time, while others prefer stepwise addition of timing constraints [23, 35]. At the other extreme are totally time based design and analysis methods, such as rate monotonic analysis.

The following papers provide review aspects of design and analysis of real-time systems: Heath [22] argues in favour of multiprocessor design architectures; Joseph and Goswami [29] review formal description techniques used for real-time systems; and Hull, et al [26] compare four methods for real-time system development. We shall now briefly examine some current trends.

Stellenbosch University http://scholar.sun.ac.za

State machines. Harel [20] describes an extension of finite state machines, called *Statecharts*. Statecharts is a visual formalism which allows the specification of concurrent components, as well as their refinement and abstraction. It is well suited to the rigorous specification of reactive systems and since the semantics of statecharts are formally specified, automated analysis of statecharts can be done [21].

Communicating Real-Time State Machines [52] is a notation for specifying the requirements of real-time systems. It allows the definition of a system of state machines that communicate over unidirectional channels.

Temporal logic and model checking have been used successfully in the verification of concurrent systems [10]. Extended temporal logics can be defined [44, 45] to allow quantitative temporal operators over bounded intervals. The resultant logics are used to specify real-time properties for models of finite state systems. These models are extended with a mechanism which allows the expression of bounds on the delays between state transitions. Model checking can be used to verify that the model satisfies the specification.

Graph theory. Graphs can be used to model real-time systems [27, 28]. Real-time constraints can be specified for elements of a graph and used in a mathematical analysis of the temporal behaviour of the system. The design method of the MARS real-time operating system uses graphs to specify real-time transactions [34, 33].

Process algebra. CSP-R [40] is an extension of Hoare's Communicating Sequential Processes [24, 25] by a real-time construct, **wait** *t*. Programs with real-time constraints can be written and statically analysed for temporal correctness. The maximal parallelism approach to real-time programming, taken in CSP-R, is criticised by Kurki-Suonio [35] because of the interdependencies on the relative speed of processes which it introduces into models.

Real-time analysis for programming languages. Fugetta et al [15] extend Haase's [18] extension of Dijkstra's guarded commands by parallel guarded commands (PARCs) for real-time programs. Execution times for PARCs are specified as weakest preconditions.

Shaw [51] describes a methodology similar to that of Fugetta, et al. He extends Hoare logic to prove assertions about deadlines and timing constraints in high level language programs.

Stoyenko et al [56] describe a system which supports guaranteed schedulability of Real-time Euclid programs. The system includes a Real-Time Euclid compiler, schedulability

Stellenbosch University http://scholar.sun.ac.za

analyser, and a kernel which schedules tasks according to the Earliest Deadline First policy on a multiprocessor hardware platform. It is capable of guaranteeing real-time response.

I have tried to give a brief overview of trends in real-time system design. As we shall see in section 2.3, the trend in hard real-time operating systems is towards distributed and multiprocessor architectures.

2.3 Operating systems for real-time systems

A large number of real-time operating systems have been developed over the last decade. The latest systems reflect the state of the art in hardware (distributed and multi-microprocessor architectures) and the theory of analysis and scheduling of real-time systems. We shall briefly review some of the well documented examples.

ARTS [58] is an object oriented distributed real-time operating system kernel and a real-time toolset which consists of a schedulability analyser and a real-time monitor. ARTS has a time driven rate monotonic scheduler and priority inheritance protocol to prevent unbounded priority inversion. The scheduler can guarantee hard periodic tasks, and perform criticality based soft real-time task scheduling, as well as overload control based on value functions of aperiodic tasks.

CHAOS^{arc} [17] is an object based real-time operating system designed for a multiprocessor architecture. It can run on bare hardware, or on an existing operating system [47, 48]. Chaos supports lockable resources, and uses a run-time earliest deadline first scheduling policy. Schedulability analysis is done dynamically and a priority assignment, based on the "criticalness" of the task, is used to determine a feasible task set [4, 48].

Real-time threads [47, 48] is a package which supports the programming of concurrent threads on a UNIX platform. Since the package is built on top of a standard UNIX platform, it is portable. Scheduling is earliest deadline, and schedulability analysis is done when the thread is created. Threads communicate via lockable shared data structures or signals. See also Chaos^{arc}.

FAIRCHILD [42] is a real-time operating system which runs on an embedded Intel 80386 system. It employs a *predictive deadline scheduling policy* [43] and supports multiprocessing, semaphores and event management. In [42] Miller gives performance statistics

Stellenbosch University http://scholar.sun.ac.za

for the system.

HARTS [54] (Hexagonal Architecture for Real-Time Systems) is a mesh of 19 nodes, each consisting of application processors, a network processor and a RISC processor. Each node connects to 6 other nodes via its network processor. A distributed operating system, HARTOS, consists of a single processor kernel running on each application processor. A distributed name service and both blocking and non-blocking inter-process communication is provided. The scheduling policy is priority based and processes can change their priority dynamically.

MARS [34, 33] (Maintainable Real-Time System) is a fault tolerant distributed real-time operating system. The MARS architecture consists of a number of MARS components inter-connected on a synchronous real-time bus. Each component is a computer on which tasks are executed, and components are clustered to manage network complexity. Messages in MARS are not consumed when read. Instead, a message may be read multiple times, and is overwritten when the system state changes. Tasks as well as communication on the MARS bus is scheduled pre-run-time, and hard real-time tasks are periodic. Aperiodic tasks usually have soft deadlines, and are handled in system idle time.

Maruti [38, 37] is an object oriented distributed real-time operating system which supports fault tolerance through replication. Maruti supports reactive systems in that task requests may be entered dynamically when a previously guaranteed schedule is already executing. Run-time schedulability analysis determines whether a new task request can be guaranteed to complete before its deadline. Tasks with no deadline guarantees execute "off-line" when no real-time tasks are ready.

Spring [55] is a distributed real-time operating system kernel which guarantees deadlines. It supports tasks with lockable resources, which are scheduled at run-time. Run-time schedulability analysis determines whether a task's deadline can be guaranteed.

Thoth [7] is a single processor real-time operating system which was designed to be portable. It runs on Data General Nova and Texas Instruments minicomputers. Tasks are scheduled on a priority basis, and a task is not preempted unless a higher priority task becomes ready, or the task is blocked.

Commercially available real-time operating systems. *VRTX* [46] is a commercially available real-time operating system for embedded microprocessor applications. Tasks

Stellenbosch University http://scholar.sun.ac.za

are scheduled by a preemptive priority based scheduler on a uniprocessor. Tasks communicate through signals, mailboxes and queues. Various real-time versions of the *UNIX* operating system have been developed [16]. Intel Corporation's RMK kernel and RMX operating system [1] are also widely used. QNX [31] is a distributed real-time operating system which provides inter-process communication through remote procedure calls.

The review of real-time operating systems provided here is by no means complete. The intention is to reflect the current trends, and to highlight how both deadline driven and priority based scheduling is employed both pre-run-time and at run-time.

2.4 The real-time producer/consumer paradigm

In 1989 Kevin Jeffay published his Ph.D. thesis [28] in which he describes a design method for real-time systems. The method is based on the premise that a real-time system can be understood in terms of producer/consumer relationships on system components. Jeffay's paradigm requires that the kth message sent on any channel must be consumed before the $k+1^{th}$ message is sent. He calls this the Real-Time Producer/Consumer (RT/PC) paradigm.

Jeffay proves that both preemptive and non-preemptive deadline scheduling could be used to realise his paradigm. This makes it well suited to reactive real-time systems which are driven by aperiodic (sporadic) events. The message passing semantics of the RT/PC paradigm makes it possible to reason about time, and make accurate performance predictions.

In his thesis Jeffay describes a graphical design method which supports RT/PC, and determines certain conditions under which an RT/PC design can be implemented with guarantees of temporal correctness. These conditions can be used to support a design and development method for real-time reactive systems which spans requirements analysis to implementation. Given appropriate run-time kernel support, the real-time behaviour of a design can be guaranteed at run-time.

2.5 Discussion

It should be clear from the brief survey of scheduling techniques that there are distinct camps in the field of scheduling for hard real-time systems. Both rate monotonic and deadline driven scheduling can be used to guarantee deadlines. Both techniques can be used in a static as well

Stellenbosch University http://scholar.sun.ac.za

as dynamic scheduling mechanism. Hybrid techniques (mixed priority and deadline driven scheduling, and dynamic priority adaption) are used widely.

The choice of scheduling algorithm (priority based or deadline driven) and whether scheduling is done pre-run-time or at run-time, is dictated by the nature of the real-time environment in which the system will run. In environments where sensors are regularly sampled, a periodic task approach is suitable. In environments where a system has to react to frequent sporadic events, a deadline driven approach is appropriate.

If it is imperative that hard deadlines be met, pre-run-time scheduling can guarantee a feasible schedule. In a closed task set deadlines can be guaranteed by both deadline and rate monotonic scheduling, but rate monotonic is preferred in periodic task environments. If task requests can arrive dynamically, all deadlines cannot be guaranteed — a new task request can render the current schedule infeasible. This means that run-time feasibility analysis must be done. If the new task set is infeasible, some tasks will not meet their deadlines. In a hard real-time environment, critical tasks must therefore be ensured to remain in the feasible schedule. This can be done by ensuring that they have a higher priority in a rate monotonic scheduler, or to use a second priority metric in deadline driven schedulers.

Next generation real-time platforms tend to have multiprocessor or distributed architectures with processors dedicated to specific purposes. Processor nodes sometimes consist of a cluster of special purpose processors: application processor, network processor, off-line scheduler, monitor, etc. The design and features of next generation real-time operating systems depend on the environment in which it is used. Systems which support periodic tasks are used in periodic environments, such as pure control systems. Operating systems which support aperiodic processes are normally used in asynchronous environments, usually with deadline scheduling. Fault tolerance is usually supported through replication. Failures of primary tasks scheduled pre-run-time are recovered by backup tasks which are activated aperiodically, and usually scheduled by deadline.

The goal of this study was stated in chapter 1 to be an investigation into a method of design, analysis and implementation for small, uniprocessor based embedded reactive systems. Run-time support for the method should allow guaranteed real-time behaviour, and the implementation method should use resources such as CPU and RAM economically.

Since the systems in which we are interested consist of closed reactive task sets, deadline scheduling can be used to guarantee real-time behaviour. Jeffay's method suits the environments for which the method is intended very well. In RT/PC messages flow through

Stellenbosch University http://scholar.sun.ac.za

the system from input devices, through communicating processes to output devices. Deadline driven scheduling makes it well suited to aperiodic environments, and RT/PC semantics allow rigorous reasoning about the timing of a system's real-time behaviour. Finally, kernel support for RT/PC can be economically provided on the platforms for which it is intended. Jeffay's method was therefore chosen to be the subject of an experimental kernel support implementation, as well as a nontrivial reactive application. The rest of this thesis describes the kernel implementation, and the application of RT/PC in the design, analysis and implementation of a real reactive application.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

The real-time producer/consumer (RT/PC) paradigm

3.1 Introduction

In this chapter we will examine Jeffay's method of design and analysis for hard real-time systems [28]. Jeffay defines a model of hard real-time systems, which he calls the *real-time producer/consumer paradigm*, and uses it as a semantic basis for a design discipline of real-time systems. The method consists of a graphical design notation for describing RT/PC systems, and a sequence of analysis steps during which the semantic correctness of the system is determined. This chapter discusses the various components of the design method, and the different aspects of analysis. More emphasis is laid in this chapter on the design method than analysis, since the analysis of RT/PC systems is the subject of chapter 5.

Jeffay models a real-time system as a system of *message producers* which communicate with *message consumers* via *unidirectional channels*. Assuming that there is no propagation delay, the consumer of a message receives it immediately after it was sent by the producer. If a producer produces messages faster than the consumer can consume them, then no amount of buffering is sufficient to prevent the loss of messages. To guarantee the correctness of the system the consumer must process messages at least at the rate at which they are produced. Based on this observation Jeffay defines the *real-time producer/consumer paradigm* (RT/PC) as follows:

Definition 3 *The consumer must process the ith output of the producer before the producer*

Stellenbosch University http://scholar.sun.ac.za

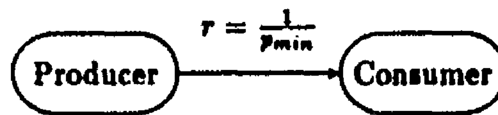


Figure 1: Real-time producer/consumer paradigm

produces the $(i+1)^{st}$ *output.*

Figure 1 illustrates the RT/PC paradigm. *Producer* and *Consumer* are processes, connected by a unidirectional channel. The rate (r) at which messages can flow on the channel between Producer and Consumer, is the reciprocal of the minimum period between any two messages on the channel (p_{min}).

3.2 System components and graphical notation

In this section we shall examine the components of Jeffay's design discipline, and describe his graphical notation. The method is based on the producer/consumer model of communicating processes, and the design of a real-time system may be expressed as a directed graph, $G = (V, E)$. V is the set of vertices, and can represent *processes*, *input devices*, *output devices* or *data repositories*. E the set of edges, and can represent unidirectional synchronous or asynchronous channels between the vertex components. A design can therefore contain[1]:

- *Processes* — machines which process messages. Processes are graphically denoted as ovals.
- *Channels* — unidirectional asynchronous connections between processes, or synchronous connections between processes and data repositories. An asynchronous channel is graphically denoted by a single-headed arrow, a synchronous channel by a double-headed arrow.
- *Data repositories* — machines which control access to data objects which are shared between two or more processes. Data repositories are graphically denoted by two concentric ovals.

[1]I have modified Jeffay's original graphical notation slightly with regard to input and output devices. Jeffay's original graphical notation uses a darkened circle for input devices, and a darkened and shaded circle for output devices. I use a bar for both input and output devices. Input and output devices can still be uniquely identified: input devices by their single outgoing directed edge, and output devices by their single incoming edge. The notational change was necessitated by a lack of graphical tools, and incurs no loss of meaning, power or clarity.

Stellenbosch University http://scholar.sun.ac.za

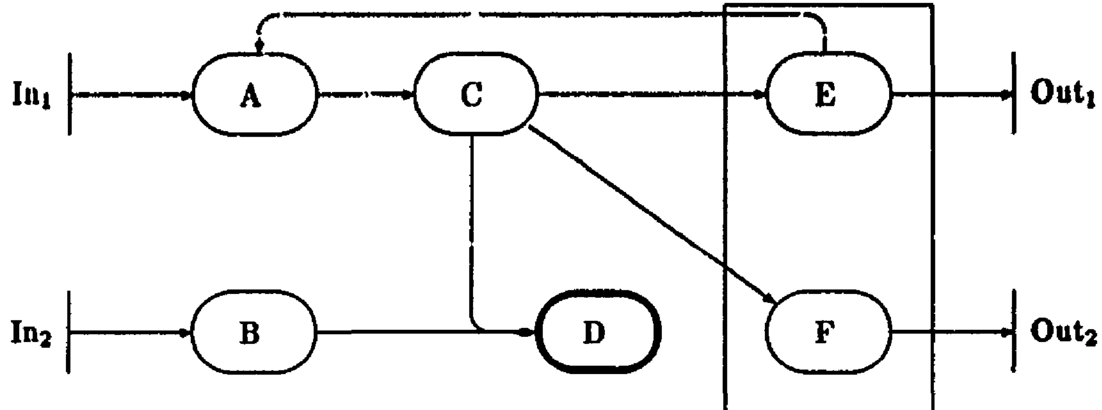


Figure 2: An RT/PC design graph

- *Input devices* — process abstractions of physical devices which cause messages to flow into the system; denoted graphically by a bar.
- *Output devices* — process abstractions of physical devices into which messages flow out of the system; denoted graphically by a bar.
- *Mutual exclusion regions* — sets of processes or data repositories which are implicitly or explicitly required to execute mutually exclusively in time. An explicitly defined mutual exclusion region is graphically denoted by enclosing its members in a box.

Figure 2 is an example of an RT/PC design graph. In the figure we can identify the following components:

- A, B, C, E and F are processes;
- D is a data repository;
- In_1 and In_2 are input devices;
- Out_1 and Out_2 are output devices;
- The directed edges of the graph represent unidirectional channels.
- The box enclosing vertices E and F defines an explicit mutual exclusion region, consisting of E and F.

Stellenbosch University http://scholar.sun.ac.za

3.2.1 Processes and channels

An RT/PC model represents a reactive system which performs certain actions in response to events that occur in its environment. In the model events are represented by messages which are transmitted to processes by input devices or other processes. When a process receives a message it may perform certain operations, and send one or more messages in response to other processes, data repositories or output devices.

An RT/PC process consists of a single execution thread, with a single entry point, which always starts with the reception of a message, and is defined by its behaviour in response to each message it may receive. Each time a process completes its execution thread it blocks until it receives another message.

Figure 3 shows a schema for an RT/PC process. The *ACCEPT* command blocks the process until it receives a message (*msg*). It then executes the body of the process, and may call *A_EMIT* and *S_EMIT* one or more times. A_EMIT is used to send a message to another process or output device on an asynchronous channel. S_EMIT is used to send a synchronous message to a data repository, and to receive a reply from it.

A process can receive messages on only one logical input port, but several asynchronous channels may connect to the same input port. A process may emit messages on a set of output ports, but only one message may be emitted on each asynchronous output channel per execution of the process thread. The input and output ports of a process are statically bound to communication channels for the lifetime of the system.

A channel is a connection between the output port of one process and the input port of another process or data repository. If a process sends a message on a synchronous channel, it awaits and will receive a response. A process which sends a message on an asynchronous channel does not block, and does not receive a response to the message. All messages sent on a channel will be received by the receiving process. As long as a process does not emit two messages on the same asynchronous channel during the same execution of the process thread, and if the system obeys RT/PC, no asynchronous channel will ever overflow. A synchronous channel cannot be overflowed because the sending process is blocked until it receives the reply from the receiver, at which point the receiver is ready to receive another message.

Apart from the RT/PC paradigm, message passing in RT/PC has the following semantics:

- A process can only consume one message at a time; and

Stellenbosch University http://scholar.sun.ac.za


```
Process:
BEGIN
    ACCEPT (msg) ;
    ⋮
    A_EMIT (m1) ; /* Asynchronous emit */
    ⋮
    S_EMIT (m2, reply) ; /* Synchronous emit */
    ⋮
END ;
```

Figure 3: Schema for an RT/PC process

- messages may be sent to a process on several input channels; therefore
- the reception of messages has to be interleaved in time.
- The receiving process cannot select which input channel it wishes to receive a message from. The ACCEPT command will present it with the next message to process.

Because of the above, the temporal behaviour of a process is not determined by how it handles its connections, but by the input rate of its input port (bear in mind that all input channels of a process connect to one logical input port). One can therefore reason about the temporal behaviour of a process in terms of only the rate of its input port.

3.2.2 Input and output devices

Input and output devices are abstractions of the behaviour of the physical devices in the environment of a system. Because an RT/PC system is reactive it is driven by the events generated by devices in its environment. These events enter the system in the form of messages sent by input devices on asynchronous channels. Reactive systems usually control devices, and this is modeled in RT/PC by sending messages on asynchronous channels to output devices.

An input device behaves like a process which intermittently sends messages on an output channel. We assume that there is a nonzero interval between any two messages emitted by an input device, with a known lower bound. It is important to know the worst case (shortest) interval between messages from an input device, because it determines the rate of down tream channels.

Stellenbosch University http://scholar.sun.ac.za

An output device behaves like a process which is always ready to receive a message on an asynchronous channel, but does not emit messages back into the system. The reason for having output devices in a design is to determine the temporal behaviour of the system with regard to each device it controls.

3.2.3 Mutual exclusion regions and data repositories

Some processes which share access to data or devices have critical sections in their execution threads, which require exclusive access to the shared resource in order to execute correctly. RT/PC allows the definition of *mutual exclusion regions* which are sets of processes which must be guaranteed to execute mutually exclusively. A set of processes may be denoted explicitly to constitute a mutual exclusion region, by enclosing them in a box as shown in figure 2.

A *data repository* encapsulates data which are shared between processes. It behaves like a reactive process with a single input port for one or more *synchronous* channels. Processes send requests to the data repository, which processes each request atomically, and replies to the requesting process. In this way it serialises concurrent accesses to shared data, thus ensuring mutual exclusion. A data repository may not emit asynchronous messages — it may only respond synchronously to received synchronous messages. Data repositories therefore do not have any output ports. Figure 4 shows a schema for a data repository.

A process or data repository logically performs a single function. To ensure the predictable completion of each execution of this function, messages from multiple sources are not allowed to be processed simultaneously. Processes and data repositories are therefore allowed only one entry point where messages can be accepted. A process or data repository with multiple input channels forms an implicit mutual exclusion region.

Jeffay sets two restrictions on the use of mutual exclusion regions:

- Each component (process or data repository) may occur in at most one mutual exclusion region; and
- processes and data repositories may not be combined in one mutual exclusion region.

Jeffay does not give reasons for these restrictions, but deadlocks may occur if vertices are freely combined in mutual exclusion regions. A process and data repository in the same mutual exclusion region would be unable to communicate, because their execution is always

Stellenbosch University http://scholar.sun.ac.za


```
DataRepository:
BEGIN
    Accept (msg) ;
    ⋮
    Reply (value)
END DataRepository ;
```

Figure 4: A schema for an RT/PC data repository

interleaved in time.

3.2.4 Well-formedness of design graphs

Jeffay imposes two restrictions on the interconnections of processes:

1. There must exist a path from at least one input device to each process in the system.
2. A process may have any number of synchronous or asynchronous output channels; however, no two asynchronous output channels of a process may have the same receiver.

Jeffay calls a design that satisfies both restrictions *well-formed*. He also states that the first condition simplifies the analysis of RT/PC designs, the second implementation of RT/PC systems.

Every message in an RT/PC system ultimately originates from an input device. Therefore, if condition 1 does not hold, it means that a process in the design graph is unreachable. If a process is unreachable, its input channel rate functions cannot be solved, with the result that it is impossible to determine analytically whether the system will obey the RT/PC paradigm.[2] An unreachable process does not contribute to the system, but complicates the analysis.

Jeffay's statement about condition 2 presumably refers to the amount of buffering required. Recall that all input channels of a process logically connect to a single input port. If condition 2 holds a process cannot send more than one message to another process during the same execution cycle. This would mean that only one message needs to be buffered per input port if condition 2 is met; one per input channel if condition 2 is not met.

[2] It is possible to determine analytically whether a system will obey the RT/PC paradigm if all the channel rate functions can be solved. This will be discussed in sections 3.4 and 3.6, as well as chapter 5.

Stellenbosch University http://scholar.sun.ac.za

3.3 Message passing semantics

Jeffay's design method specifies special temporal semantics for message passing. Every channel has one producer and one consumer of messages. All connected pairs of processes obey the RT/PC paradigm, which requires that whenever a message is produced on a channel, it will be consumed by the receiving process before the next message is produced on the same channel. The time interval in which a consumer may consume a message is therefore defined by the arrival rate of messages on the input port of that consumer.

3.3.1 Message transmission rates

Each channel in a design graph has an associated message transmission rate which is defined in terms of the worst case (shortest) inter arrival time of messages at the receiving process of the channel. If p_{min} is the shortest inter arrival time of messages on a channel, its rate is

$$r = \frac{1}{p_{min}},$$

where p_{min} is an integer multiple of a time unit of suitable granularity.

3.3.2 Restrictions on process construction

In order to ensure that a process can be guaranteed to obey RT/PC, Jeffay defines a number of restrictions on the construction of processes and data repositories in implementations.

- Process and data repositories are implemented in a sequential language in such a way that the execution time of each programming construct can be statically determined.

  In order to determine whether an RT/PC implementation is guaranteed to be temporally correct, the execution cost (timing) of each process must be determined accurately. Ideally the cost of each construct should be automatically determinable, possibly by the compiler for the implementation language. As such tools do not exist for most compilers used in real systems (such as C, Modula-2, etc), one has to revert to other tools to measure the timings of processes. This can be done accurately as long as the implementation language is sequential, and the execution of a thread is not interleaved with other threads.

- Potentially nonterminating constructs, such as loops, must contain a mechanism to ensure termination. An unbounded loop can easily occur when a loop is used to test for a

Stellenbosch University http://scholar.sun.ac.za

specific condition in a hardware device. If the condition never occurs (due to hardware malfunction, for example) the loop will not terminate. The worst case (longest) execution time for a loop must be used in the analysis of RT/PC systems. An upper bound must therefore exist on the number of times any loop may be executed. If a bound is not explicit in the loop condition, a special condition is required. One way of guaranteeing a bounded loop is to increment and test a special loop counter, and terminate the loop if an upper bound is exceeded. Another method would be to set and test a timer.

- During the processing of a message, a process may emit only one message per asynchronous channel. This ensures that the receiver can be scheduled to process the message before another message is sent on the same channel.

3.4 Calculating channel input rates

In order to determine the precise semantics of an RT/PC system one must be able to calculate the transmission rates of all channels accurately. Synchronous channels are only used to connect processes with data repositories. To adhere to RT/PC a process must consume each message it receives within $\frac{1}{r}$ time units, if r is the rate of its input port. If a process communicates with data repositories, all synchronous communications must be completed within $\frac{1}{r}$ time units as well. The rate of synchronous channels are therefore determined by the rate of asynchronous channels.

3.4.1 Calculating message output rates

A channel's transmission rate is the rate at which the sending process connected to it transmits messages on that channel. The transmission rate of the process depends upon the rate at which it processes messages on its own input port. A *transmission rate function* is defined for each asynchronous channel, which maps the rate at which the sender process receives messages to the rate at which it emits messages on that channel. The time required to execute the code to process messages received on its input port, and the rate of the input port of a process determines the transmission rate function of each of the output channels of the process.

Stellenbosch University http://scholar.sun.ac.za

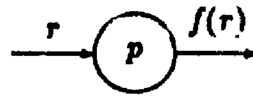
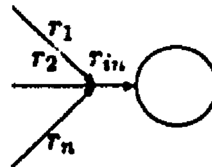


Figure 5: Message transmission rate function



$$r_{in} = \sum_{i=1}^{n} r_i$$

Figure 6: Input rate definition for MP/SC processes

3.4.2 Calculating message input rates

The input rate of a process with only one input channel is equal to the message transmission rate of that channel. A process which receives messages on a set of input channels is required to obey RT/PC on each of those channels. Messages can arrive simultaneously on any number of the input channels of a process. A single buffer is therefore required for each input channel. If messages are buffered for input channels, the input rate of a multi-producer/single-consumer (MP/SC) process is equal to the aggregate rate of its input channels.

The rate of a channel connecting an input device to a process is simply the worst case rate at which the input device can produce messages.

If a design graph is acyclic and well-formed, the transmission rates on all asynchronous channels can always be calculated. These equations may be solved numerically or symbolically.

3.5 Scheduling results

In an implementation of an RT/PC design, the RT/PC system is represented by a set of communicating tasks[3]. To ensure that the implementation satisfies the temporal requirements

[3]To differentiate between a design and its implementation, we refer to processes in a design graph, and tasks in an implementation task set.

Stellenbosch University http://scholar.sun.ac.za

of the design, it is necessary to find a way to schedule the tasks in such a way that the RT/PC paradigm is preserved. Since RT/PC requires that a message is consumed before a certain deadline, a scheduling mechanism which can guarantee deadlines is required.

If a set of tasks is feasible it is possible to find a scheduling discipline which will ensure that every task will always meet its deadline. In order to find an optimal scheduling discipline which can support RT/PC systems, Jeffay examined the earliest deadline first (EDF) scheduling discipline with regard to RT/PC. The repetitive behaviour of RT/PC systems suggests the characterisation of an RT/PC process as a cyclic task, where a cyclic task is one that makes repeated requests for execution.

Definition 4 *A* **cyclic task** *T is a 3-tuple (s, c, p) were*

1. *s = start time (also called release time): the time of the first request for execution of T;*
2. *c = computational cost: the time to execute T to completion on a dedicated uniprocessor; and*
3. *p = period: the interval between requests for execution of task T.*

Jeffay distinguishes between two types of cyclic task: sporadic and periodic. If a task makes requests for execution at regular intervals it is periodic, else sporadic. These two types of tasks are typical of two important characterisations of real-time systems: periodic in time driven and sporadic in event driven state machines [32].

In chapter 3 of his thesis, Jeffay proves feasibility and optimality results for the EDF scheduling discipline with regard to the RT/PC paradigm. The results of his investigation can be summarised as follows. For preemptive scheduling:

- EDF is an optimal discipline (see definition 2) if preemption is allowed at arbitrary points in a process. If a feasible schedule exists for a task set, and every process can be preempted at any point, the EDF discipline will correctly schedule the tasks.
- Feasibility of a task set can be determined analytically for arbitrary release times of both sporadic and periodic tasks.

For non-preemptive scheduling:

- EDF is an optimal discipline for sporadic tasks; but

Stellenbosch University http://scholar.sun.ac.za

- EDF is not an optimal discipline for periodic tasks with arbitrary release times.
- For sporadic tasks, feasibility can be determined efficiently for arbitrary release times; but
- for periodic tasks, feasibility is dependent on knowledge of release times, and the problem of determining the feasibility of a set of periodic tasks with arbitrary release times is intractable.

These results will be used in chapter 4, when the design of a kernel which supports RT/PC is considered.

3.6 Implementing RT/PC designs

The RT/PC paradigm requires that for every channel in a system, the receiver of the channel consumes messages faster than its producer emits messages on that channel. In order to determine whether a system is guaranteed to obey RT/PC, we have to reason about the temporal properties of a design graph. We do this in terms of the message transmission rates of the asynchronous channels of the design graph.

If the set of equations which describe the message transmission rate functions of a design graph can be solved, it means that the maximum input rate of messages on the input port of each process is known. We know from section 3.5 that the *earliest deadline first* (EDF) scheduling discipline is optimal for sporadic tasks. Therefore, given a fast enough processor, we can always schedule the processes of the design graph with the guarantee that each pair of connected processes will obey the RT/PC paradigm. Such a design graph is said to be *realisable*, meaning that there exists a mapping from it to a *feasible* task set.

Definition 5 *An implementation of a design graph is* **temporally correct** *if every pair of vertices connected with an asynchronous channel is guaranteed to adhere to the RT/PC paradigm.*

Definition 6 *A design graph is* **realisable** *on a uniprocessor if it is possible to implement the design on a uniprocessor so that the design will be temporally correct.*

Realisability is an absolute measure of temporal correctness. A realisable design graph can always be implemented given a fast enough processor, while a design graph which is not realisable cannot be implemented with guarantees of temporal correctness.

Stellenbosch University http://scholar.sun.ac.za

Jeffay proved that the following conditions are necessary and sufficient for realisability of design graphs:

1. *Acyclic, well-formed* design graphs are realisable.
2. Design graphs with *disjoint cycles* are realisable iff at least one channel in each cycle has a nonidentity transmission rate function. This will be discussed informally in chapter 5.

   Definition 7 *Two cycles in a design graph are* **disjoint** *if no process appears in both cycles.*

   Definition 8 *If the transmission rate function, $f(r)$, of a channel is defined by $\frac{1}{x}r$, then the* **slope** *of the transmission rate function is $\frac{1}{x}$.*

   We can now formally define the condition for realisability of a graph with disjoint cycles as follows:

   Definition 9 *Let* G *be a design graph with a cycle* C *of* n *distinct processes and asynchronous channels. Let $\frac{1}{x_1}, \frac{1}{x_2}, \ldots, \frac{1}{x_n}$ be the slopes for the transmission rate functions for the channels in* C. *If* C *is disjoint from other cycles in the graph, then the transmission rate functions for the channels in C can be solved iff*

   $$\prod_{i=1}^{n} x_i > 1$$

3. Design graphs with *non-disjoint cycles* are realisable only if the following two conditions are met:

   (a) At least one channel in each cycle has a nonidentity transmission rate function.

   (b) There must exist a process or sequence of processes in one of the cycles in a non-disjoint set of cycles, in which at least three messages must be sent to the first process in the sequence before a message can be emitted from the last process in the sequence. Formally:

   Definition 10 *A* **simple cycle** *in a design graph is a cycle in which all vertices are disjoint.*

   Definition 11 *Let* G *be a design graph with non-disjoint cycles. Let C_1 and C_2 be two non-disjoint, simple cycles. Assume that C_1 and C_2 contain* n *and* m *processes respectively. If processes in C_1 are interconnected with channels whose transmission*

Stellenbosch University http://scholar.sun.ac.za

rate functions have slopes $\frac{1}{x_1}, \frac{1}{x_2}, \ldots, \frac{1}{x_n}$, and processes in C_2 are interconnected with channels whose transmission rate functions have slopes $\frac{1}{y_1}, \frac{1}{y_2}, \ldots, \frac{1}{y_m}$, then the transmission rate functions for the channels in C_1 and C_2 can be solved only if

$$(\prod_{i=1}^{n} x_i - 1)(\prod_{j=1}^{m} y_j - 1) > 1$$

We do not have sufficient conditions for when the transmission rate functions can be solved for arbitrary patterns of non-disjoint cycles. However, if each pair of simple cycles has at most a single process that receives messages from a distinct process in each cycle, then the necessary condition given above is also sufficient for solving the transmission rate functions.

Realisability analysis is called *processor independent* analysis because a realisable design graph can always be implemented on a fast enough uniprocessor. *Temporal correctness* is a measure of the correctness of an implementation of a design graph. The following chapters describe an experiment to investigate how realisable RT/PC design graphs can be implemented to be temporally correct.

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

The ESE Kernel

In chapter 3 we discussed Jeffay's method of developing real-time systems with guaranteed temporal behaviour. He showed that his RT/PC[1] paradigm can be used to design systems with hard real-time constraints in such a way that all deadlines can be guaranteed. He also showed that RT/PC is realisable by developing a kernel which supports RT/PC design graphs, and by implementing a number of systems with hard real-time constraints on it [28].

Jeffay's system runs on the UNIX operating system on a Sun workstation, and interacts with devices at a very high level. I propose that RT/PC is also suitable for the design and implementation of general embedded reactive systems. In order to investigate this, I developed a nontrivial example on an industrial embedded platform. In chapter 5 we shall examine the implementation of the case study. To support RT/PC systems on an embedded platform I developed a kernel, called *Embedded Systems Executive* (ESE — pronounced "easy"), which runs on Intel 80x86 based hardware platforms. In this chapter we shall examine the design of this kernel.

4.1 Embedded Reactive Systems

Embedded systems are usually reactive. A *reactive system* is idle until an event occurs in its environment. The system performs certain actions in response to the event, and then awaits the next event. A state machine can be used to determine the reactive system's response to a particular event, given the current state of the system. The action taken by the reactive system in response to an event can change the state of the system. We can define a reactive

[1] Real-Time Producer/Consumer

Stellenbosch University http://scholar.sun.ac.za

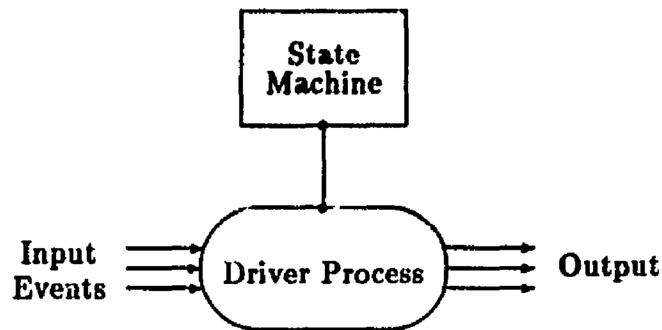


Figure 7: Reactive system

system as follows:

Definition 12 *A reactive system is an event driven state machine.*

Figure 7 shows the components of a reactive system. Input events are generated by devices in the environment of the system. The driver process receives the events, and executes the state machine. The execution of the state machine typically causes output to be generated. The output may be control signals to a device, data written to a persistent data store, or messages sent to other processes in the system.

An ***embedded system*** is an autonomous component of a larger system. It can run on an autonomous hardware platform embedded in the hardware of the larger system. Examples of such systems are intelligent communications adapters, disk subsystems, etc.

Autonomous stand-alone systems often suffer the same problems as the embedded systems: hard real-time constraints, limited CPU, RAM and other resources, etc. Examples of such systems are robots with autonomous control systems.

A software system can also be embedded within a larger software system — in other words, run on the same hardware platform of the larger system. Examples of such systems are real-time systems which are embedded in general purpose operating systems.

The main characteristics of embedded systems are:

- Devices introduce concurrency — embedded systems usually interact with several autonomous devices. These devices introduce concurrency which is best modeled as a system of communicating processes.

Stellenbosch University http://scholar.sun.ac.za

- Devices have real-time requirements — devices usually have to be serviced within specified temporal limits, otherwise information is lost. Many problems in embedded systems are timing related: a missed event because the system was engaged in another action, an overflowed buffer because a message consumer was too slow, etc.

- Reactive systems — an event driven reactive system can be modeled very conveniently as a system of communicating state machines. This requires inter process communication and the guarantee of atomicity of state machine commands.

  Because of their "black box" nature, embedded systems can often be tested only in terms of their response to events from their actual environment. Certain sequences of events may be timing dependent, and very difficult to reproduce. It can therefore be very hard to find errors in embedded systems.

- Memory and CPU requirements are usually tight in embedded systems, and resources have to be used optimally.

Conventional multi tasking operating systems do not adequately address the problems of embedded systems. The memory requirements of a general purpose operating system (UNIX, for example) is usually prohibitive, and the processing overhead it introduces, unacceptable. Furthermore, the process scheduling policy of general purpose operating systems is usually a variant of time sliced round robin scheduling with multiple levels of priority. This has the following implications:

- Real-time constraints cannot be guaranteed;

- Priority levels must be tailored to accommodate processes with higher processing requirements;

- The order in which processes (which communicate via asynchronous message passing) will be scheduled may be unpredictable. Inter process message buffers must therefore provide for the worst case of outstanding messages.

The ESE kernel is designed to meet the requirements of embedded reactive systems which are designed according to RT/PC. I assume that the number of processes and channels in an embedded system is fixed, and new channels or processes cannot be dynamically created during the lifetime of the system. Jeffay's methodology can therefore be used to analyse the system, and to determine a priori whether all deadlines can always be guaranteed.

Other principles which influenced design decisions were the following:

Stellenbosch University http://scholar.sun.ac.za

- Portability — different processors are often used for embedded systems. ESE should be portable across different architectures with the minimum of effort;
- State machines — A state machine is a very suitable formalism for the description of reactive systems. ESE must be suitable for the implementation of state machines.

The rest of this chapter describes how ESE supports RT/PC and the requirements of embedded reactive systems.

4.2 Events

Events are fundamental to most models of concurrent systems. In CSP [25], for example, a process is defined entirely in terms of the possible sequences of events which may occur during its execution. A state machine is defined in terms of the state transitions which it makes in response to the occurrence of events. We define an event as follows:

Definition 13 *An event is an atomic occurrence. It may cause a state machine to perform an action, or it may generate input to a process.*

The granularity of an event is determined by the level of abstraction at which a process observes it. Consider the following example:

> A transport layer protocol sends a data unit to a remote destination. Because the data unit is larger than the maximum network layer datagram size, the network layer protocol fragments the transport layer data unit into two network layer data units. These fragments are received by the destination, reassembled by the network layer protocol and passed to the transport layer protocol.

Let us now examine the sequence of events which describes the arrival of the data at its destination. Our first vantage point is at the link layer. We see a sequence of events generated by the physical layer, representing the arrival of the data:

$$
\begin{aligned}
&\mathit{LinkLayerFrameArrival} = \\
&\quad \mathit{OpeningFlag} \rightarrow \mathit{DataByte} \rightarrow \cdots \rightarrow \mathit{DataByte} \rightarrow \mathit{CRC} \rightarrow \mathit{ClosingFlag} \rightarrow \\
&\quad \mathit{OpeningFlag} \rightarrow \mathit{DataByte} \rightarrow \cdots \rightarrow \mathit{DataByte} \rightarrow \mathit{CRC} \rightarrow \mathit{ClosingFlag}.
\end{aligned}
$$

Stellenbosch University http://scholar.sun.ac.za

The link layer passes the data encapsulated in the link layer frames to the network layer. The events seen by the network layer are:

$$NetworkLayerDatagramArrival = \\ \quad Fragment1Arrives \rightarrow Fragment2Arrives.$$

The network layer reassembles the transport layer data unit encapsulated in the two network layer datagrams, and passes it to the transport layer. The event seen at the transport layer is:

$$TransportLayerDataUnitArrival = DataUnitArrives.$$

The previous example shows that an event at a higher level of abstraction is the result of a process at a lower level. What is atomic at one level is structured at another.

Physical events occur in the environment of RT/PC systems — for example:

- a message arrives on a data communications physical medium;
- a quantity measured by a sensor reaches a predefined threshold;
- a timer expires; or
- an operator issues a command.

An event could manifest itself to an RT/PC system in various ways. It could be in the form of an interrupt; the raising of a condition which is polled; or an incoming message on a communication link (for instance a transputer link). A physical event causes an RT/PC input device to send a message to an RT/PC process. The process performs certain actions in response to the message, and may send messages to other processes and output devices.

An RT/PC system is therefore a reactive system — it reacts to input from the devices which constitute its environment, and controls devices in the environment by sending messages to them. All actions are taken directly or indirectly in response to events (in the form of messages from input devices).

The view that RT/PC systems are event driven reactive systems, has been the guiding principle in the design of the ESE kernel.

Stellenbosch University http://scholar.sun.ac.za

4.3 Preemptive and Non-preemptive Scheduling

Jeffay proved that a realisable RT/PC design can be scheduled either preemptively or non-preemptively, and still be guaranteed to meet all its deadlines. As long as the *earliest deadline first* (EDF) scheduling policy is adhered to, the choice of preemptive and non-preemptive scheduling is not determined by RT/PC. The choice must be made on implementation and other requirements.

The choice of scheduling policy for ESE was made on the design principles stated previously. The guiding principle is that ESE is intended for embedded reactive systems, designed according to RT/PC. The requirements for ESE's scheduling policy are:

- it must be capable of supporting RT/PC;
- due to real-time requirements and limited resources, it must be as efficient as possible; and
- it must support state machines.

4.3.1 Support for RT/PC

We know already that both preemptive and non-preemptive EDF scheduling will support RT/PC. An important RT/PC requirement which impacts the choice of scheduling policy is mutual exclusion. Both explicitly defined mutual exclusion regions and data repositories require kernel support for mutual exclusion. If the scheduling policy is preemptive, the kernel must implement primitives to ensure that mutual exclusion is maintained in the critical sections of mutual exclusion regions and data repositories. A non-preemptive scheduling policy, on the other hand, implies that all threads always run to completion. This means that mutual exclusion is guaranteed in critical sections by the scheduling policy, and no special primitives are required for its implementation.

4.3.2 Scheduling efficiency

The efficiency of preemptive and non-preemptive EDF scheduling can be compared in terms of their respective memory requirements and context switching overhead.

A preemptive scheduling policy requires that the complete context of each process must be maintained separately. In the implementation of ESE this would imply a separate stack for

Stellenbosch University http://scholar.sun.ac.za

each process, and that all processor registers must be saved when a process is preempted.

If the scheduling policy is non-preemptive it is possible to run all processes on the same stack. The amount of memory saved in this way can be significant in an embedded system. If sporadic processes are scheduled non-preemptively, it is possible to implement the scheduler in such a way that no context has to be saved during a context switch. This makes non-preemptive scheduling more efficient than preemptive scheduling, in terms of both memory usage and context switching overhead.

4.3.3 Support for state machines

For a state machine to be deterministic, its actions must be atomic. If it is possible for a state machine action to be interrupted by any other action which may change the machine state, the result of the state machine's actions will not be deterministic.

A preemptive scheduling policy makes it possible for a state machine action to be preempted and for another, which may change the system state, to be scheduled. Atomicity of state machine actions must therefore be ensured by the programmer, if preemptive scheduling is used.

A non-preemptive scheduling policy ensures that state machine actions will always run to completion before another process is scheduled. There is therefore no requirement for the programmer to ensure the atomicity of state machine actions. A whole class of errors, which may be introduced by the inherent concurrency of most reactive systems, is therefore eliminated by the scheduling policy. The importance of this safety becomes even greater when the reactive system is embedded in an environment where debugging is difficult.

4.3.4 Design decision

Non-preemptive EDF scheduling was implemented for ESE for the following reasons:

- non-preemptive EDF scheduling supports RT/PC;
- non-preemptive scheduling is more efficient than preemptive scheduling in terms of both memory required and context switching overhead; and
- non-preemptive scheduling ensures mutual exclusion and the atomicity of state machine actions implicitly, thus relieving the programmer of the onus of ensuring mutual exclusion.

Stellenbosch University http://scholar.sun.ac.za

4.4 Periodic vs Sporadic Processes

The decision to make support for reactive systems a design principle has an important result: *processes in a reactive system are always sporadic.* This is due to our definition of reactive systems — the system is idle until an event causes it to perform a state machine action. A periodic process can be supported in a reactive system by defining a periodic timer event which causes a sporadic process to be executed periodically.

If processes are scheduled non-preemptively, it becomes unnecessary to execute each process exclusively on its own stack. But in order for all processes to run on one stack, the activation record of the current process must be removed from the stack before the next process can be activated. This implies that the release of a ready process is equivalent to a procedure call, and that the process must return from that procedure on completion. Processes which loop forever cannot be supported because they will never be descheduled.

Because ESE's processes are sporadic and scheduled non-preemptively, processes are predictable (deterministic) in both their logical and temporal behaviour. Each process has a single entry point, and always runs to completion. A process is scheduled only in response to the occurrence of an event, and will branch deterministically in response to the event. No other process can interrupt an executing process and modify the state of the system. The predictability of ESE processes makes systematic testing possible, and the real-time behaviour of a process can be measured accurately for each possible event.

ESE's non-preemptable sporadic processes are suitable for the implementation of systems of connected state machines. Each state machine is implemented as a process. It receives its events in the form of messages from other processes, and its output may be the input events to other state machine processes. An example of such a system is a protocol stack. Each protocol layer is a state machine which may be encapsulated in a process. Each layer receives messages from, and sends output messages to its next higher and lower neighbouring layers.

```
PROCEDURE                       AProcess () ;
VAR
    msg                             : POINTER TO MessageType ;
    msg_size                        : CARDINAL ;
    msg_src                         : ESE.UserRef ;
BEGIN
    ESE.Receive
        (msg,
```

Stellenbosch University http://scholar.sun.ac.za


```
          msg_size,
          msg_src) ;
    CASE msg_src OF
        SOURCE_A :
            ProcessASourceAMessage (msg, msg_size)
    |   SOURCE_B :
            ProcessASourceBMessage (msg, msg_size)
    END
END AProcess ;
```

AProcess, above, is an example of a skeleton of an ESE process. The process entry point is always at the start of the body of the procedure which implements it. The *Receive* call does not block, and will not deschedule the process. Each process may call *Receive* only once between its entry point and termination, and a ready process will be scheduled even if it does not call *Receive*. The only way in which a process can acquire the message for which it is scheduled, is by calling *Receive*, therefore it is usually the first operation in the thread of a process. Receive also returns the user reference of the active channel (for which the current process was scheduled). *AProcess* can receive messages from two channels, and uses the user reference of the active channel to branch to the routine which is used to process a message for that channel. There is no restriction on the number of input channels to a process.

For every process in an RT/PC system, ESE stores a process record which contains a symbolic name for the process, as well as a reference to the procedure which implements it.

```
    ProcessRecord = RECORD
        name        : Name ;
        thread      : Thread ;
    END ;
```

Name is an ASCII string, and will be described in appendix A. *Thread* is a procedure type which contains the entry point for a process.

```
    Thread = PROCEDURE () ;
```

Stellenbosch University http://scholar.sun.ac.za

4.5 Data Repositories and Synchronous Message Passing

Jeffay's data repositories encapsulate persistent data which are shared between processes. In order to ensure the integrity of the data, a data repository is accessed only via a single synchronous message passing port. A data repository never emits asynchronous messages, but only replies to requests. In this way access to the data repository is serialised.

Since ESE processes are scheduled non-preemptively, access to persistent shared data is serialised by the scheduler. It is therefore not necessary to implement a synchronous message passing channel to data repositories. The synchronous access to a data repository degenerates to a procedure call.

4.6 Synchronous vs Asynchronous Channels

The purpose of the ESE kernel is to support systems designed according to the RT/PC method. All RT/PC systems are reactive, and messages flow from input devices through processes to output devices. Unlike client-server systems there is therefore no synchronous message flow back to the source of a message, except in the case of data repositories. We have already seen that ESE's non-preemptive scheduling policy and sporadic processes make kernel support for data repositories unnecessary. We therefore need only asynchronous channels to support RT/PC.

ESE is also required to support state machine implementations of reactive systems. A state machine implemented on ESE will receive its input events in the form of messages on channels, and both synchronous and asynchronous message passing could be used to send an event message to a state machine. However, a state machine does not always send the output, generated in response to an event, back to the source of the event. Instead, the output usually goes to a device controller or another state machine. In neither case is there any synchronous message flow back to the source of an event.

When two processes communicate on a synchronous channel, the sender is preempted while the receiver consumes the message, and remains blocked until the receiver replies. This would require preemptive scheduling and therefore a separate stack would have to be maintained for each process to store its context when preempted.

All ESE message passing channels are implemented in the memory of a single processor. Channels are therefore highly reliable, and it is unnecessary to acknowledge the delivery of a

Stellenbosch University http://scholar.sun.ac.za

message. Asynchronous channels are suitable to convey the events and messages which flow in RT/PC systems. They can be implemented very efficiently and allow a scheduler implementation which executes all processes on the same stack. Given our design principle that ESE must support embedded RT/PC systems, only asynchronous message passing channels were implemented in ESE because of the gain in processing efficiency and memory usage.

For each asynchronous channel in a system, ESE maintains a channel record with the following [illegible]:

```
  Async
 ChannelRecord    = RECORD
       name               : Name ;
       user_ref           : CARDINAL ;
       receiver           : Process ;
       sender             : Process ;
       period             : Time ;
       deadline           : Time ;
       message            : Message ;
       max_message_size   : CARDINAL ;
   END ;
```

The *user_ref* is a number chosen by the programmer to represent that channel. ESE will use the user reference to identify the channel to the receiving process when it is scheduled. ESE's asynchronous channels identify both their *sender* and *receiver* processes. The receiver process identification is used by the scheduler to activate the receiver process, and the sender identification is used by the ESE *Send* command to check that each channel connects only two processes (as required by RT/PC). The *period* is stored to compute a *deadline* when a message is sent on the channel. When a message is sent on a channel it is stored in a field called *message*, and *max_message_size* is used to check that it is not lager than the buffer of the channel.

The types, *Process*, *Time* and *Message* are described in appendix A.

4.7 Mutual Exclusion Regions

Some processes which share data have critical code sections in which each process must be assured of exclusive access to the shared data. If not, the critical section may be executed

Stellenbosch University http://scholar.sun.ac.za

incorrectly. RT/PC allows the specification of *mutual exclusion regions* to ensure the serialisation of critical accesses.

ESE's non-preemptive scheduling policy implicitly serialises the execution of all processes, and no special kernel support for mutual exclusion regions is therefore required.

4.8 Interrupts and Events

ESE (and RT/PC) systems are driven by messages flowing in channels. A process will be scheduled only if it has a message pending on an input channel, and every message in an RT/PC system can be traced back to the occurrence of an event in the environment. Environmental events usually manifest themselves as interrupts. For each type of interrupt there is an interrupt handler which must convert the information conveyed by the interrupt into a message, so that it can be processed by a process which manages the interrupting device.

For an interrupt handler to be able to send a message to its device driver process each time it receives an interrupt, the shortest interval between any two consecutive interrupts of that type must be longer than the time required to send a message, schedule its receiver and process the message. Interrupts often occur in bursts which exceed the rate at which the receiving device driver processes can be scheduled. When this occurs in an RT/PC graph, the design is not realisable. In practice, however, there is usually a known upper bound on the length of a burst of an interrupt.

Consider, for example, the device driver process of a communications device which provides a physical layer service to a link layer protocol[2]. The shortest possible inter frame delay can occur when any frame is followed immediately by an information frame with only one data byte. An information frame with one data byte consists at the physical frame level of 7 bytes, composed as depicted in figure 8. The worst case interrupt behaviour (assuming a connection with no idle time) is therefore a frame followed seven byte transfer intervals later by an information frame. At a transmission rate of 64000 bits/s this translates to an inter frame period of 875μs. In the case study described in chapter 5 an X.25 protocol stack is implemented on an Intel 80188 processor running at 10MHz. On this platform the system would be hard pressed to service a sustained burst of short frames, if scheduled strictly according to the semantics of RT/PC.

[2] This example uses the frame format of the balanced link access procedure protocol (LAPB) of CCITT's X.25 protocol stack.

Stellenbosch University http://scholar.sun.ac.za

Figure 8: Information frame format

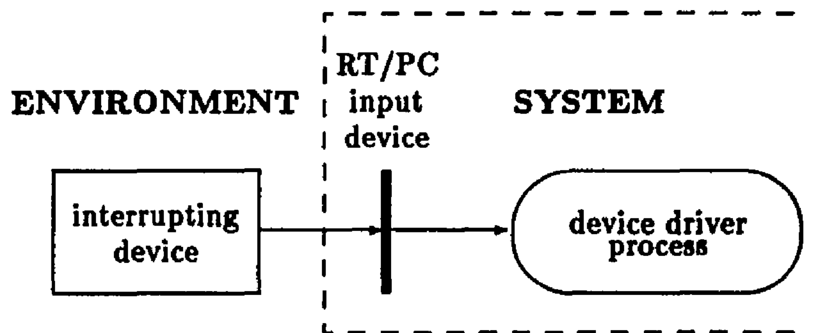


Figure 9: Fast Interrupting Device: Unrealisable System

If the link layer protocol implements a reliable service with window flow control, we know that a sustained burst of frames cannot be longer than the available window size. We also know that another burst will not be received until some of the received information frames have been acknowledged. This means that incoming frames can be buffered, and because the upper bound on the length of a burst is known, the receive buffer can be made large enough so that no incoming frame is ever lost due to buffer overflow.

When the length of an interrupt burst is bounded as described above, the interrupt handler can buffer messages from input devices, and release them into the system at a rate at which the design is realisable. In effect the boundary between the system and its environment is shifted. Instead of the physical device determining the rate of the input device, the input from the physical device is buffered. The rate at which messages are released into the system is determined by the buffer, which is logically a part of the environment.

Figure 9 shows a system with a device which generates interrupts too fast for the system to be realisable. Figure 10 shows the same system, with an interrupt buffering mechanism which releases events into the system at a realisable rate. The figure shows how the interrupt buffering mechanism becomes logically part of the environment rather than the system.

In ESE an interrupt handler signals the occurrence of an interrupt on an *input port*, which is an asynchronous channel which does not convey data — only the fact that an event has occurred. An input port counts the number of signals it receives and its receiving process is scheduled

Stellenbosch University http://scholar.sun.ac.za

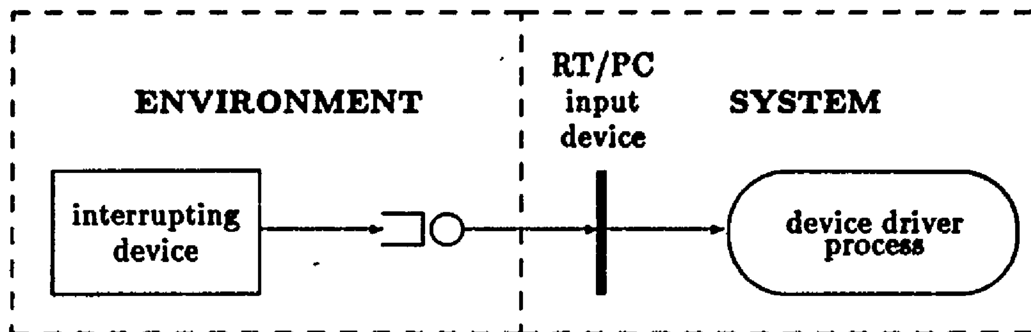


Figure 10: Fast Interrupting Device: Realisable System

once for every signal received. Each input port is assigned a *period*, and its receiving process is scheduled according to the same EDF policy which applies to asynchronous channels.

```
InputPortRecord          = RECORD
    name                 : Name ;
    user_ref             : CARDINAL ;
    receiver             : Process ;
    period               : Time ;
    level                : CARDINAL ;
    deadline             : Time ;
END ;
```

Each time that an interrupt handler signals on an input port, the *level* of the port is incremented. Each time that the receiving process of the input port is scheduled, the level of the port is decremented. The receiving process of an input port is ready when the level of the port is non-negative ($level \geq 0$). A *deadline* is calculated for an input port when it receives a signal while its level is zero, or when its receiving process is scheduled and the new level is still nonzero.

The period of an input port has the following scheduling implications:

- if the level of an input port is positive, and no other input ports or asynchronous channels are ready, then the receiving process of the input port will be scheduled immediately;
- if another asynchronous channel or input port has an earlier deadline, the receiving process of a ready input port will be scheduled no later than its deadline; and

Stellenbosch University http://scholar.sun.ac.za

- if several occurrences of an interrupt have been buffered, the receiving process of the input port will be scheduled once for each level of the port, and the delay between each release of the receiving process will be no greater than the period of the input port.

The period of an input port therefore determines the *minimum frequency* at which its receiving process will be scheduled. If the system is not heavily loaded, and there are no other channels with earlier deadlines, the receiver will be scheduled faster than the minimum frequency.

A system may recognise other events than just interrupts. If a process has an event to signal to another process, it may do so by sending a message to the other process. Such a message would not contain any data, since it merely indicates that an event had occurred. The signal command of ESE can be easily generalised to support another kind of port: one which one process uses to signal to another process (rather than an interrupt handler to a device driver). ESE therefore supports two port types: an *input port* for interrupt handlers, and a *general port* for processes.

A general port identifies its signaling process as well as its receiving process. The signal command checks whether the current signaling process is the declared signaling process of the port. This guarantees that ports, like asynchronous channels, are unidirectional connections between only two processes. An input port is logically a connection between a physical device and a process, and because interrupt handlers are not activated by the scheduler, the identity of the signaling process cannot be checked for input ports.

4.9 Timers

Many reactive systems require timers to implement periodic processes, to verify the completion of tasks, or to check that certain events occur within a specified interval. ESE supports timers which can be set to expire after a specified interval, after which it sends a message to a process on an asynchronous channel. Associated with each timer is a *user reference*, a *channel* to send *notification* of expiry on, and a *deadline*.

```
TimerRecord            = RECORD
    user_ref             : CARDINAL ;
    notification_channel : AsyncChannel ;
    deadline             : Time ;
END ;
```

Stellenbosch University http://scholar.sun.ac.za

AsyncChannel identifies an asynchronous channel in ESE's channel table. It will be described in section A.

When a timer is started, ESE calculates its deadline based on the current system time and the period specified in the request to start the timer. A special process is periodically scheduled to check whether any timers have expired. If the current system time is later than a timer's deadline, the timer is stopped and a notification sent on its notification channel. The notification consists of a message which contains the user reference which was specified when the timer was set.

4.10 Implementation of Scheduling

To the scheduler there is no difference between an asynchronous channel and a port. Both have a name, user reference, receiver, period and deadline. For the rest of this section asynchronous channels and ports will both be referred to as channels, except where it is necessary to distinguish between the two. "Channel" should be read as "channel (or port)", and "message" should be read as "message (or signal)".

```
ChannelRecord                    = RECORD
    name                         : Name ;
    user_ref                     : CARDINAL ;
    receiver                     : Process ;
    period                       : Time ;
    deadline                     : Time ;

    CASE channel_type : ChannelType OF
        ASYNC_CHANNEL :
            sender               : Process ;
            message              : Message ;
            max_message_size     : CARDINAL ;
    |   INPUT_PORT :
            level                : CARDINAL ;
    END
END ;
```

ESE maintains a table of channels for scheduling purposes:

Stellenbosch University http://scholar.sun.ac.za


```
channel_table        : RECORD
    channels         : ARRAY [0..MAX_CHANNELS-1] OF ChannelRecord ;
    no_of_channels   : CARDINAL ;
END ;
```

Because ESE processes are sporadic, a process will only be scheduled when there is a message for it to process — processes are not scheduled periodically. Jeffay showed that all messages in a feasible RT/PC design will always be processed before their deadlines, if their receiving processes are scheduled according to the earliest deadline first (EDF) policy. Because deadlines are associated with messages, a process is scheduled to process a message on a specific channel: the one with the earliest deadline — the process cannot decide which channel to accept a message from. In contrast, a process in a general purpose operating system can be scheduled whenever it is not blocked on a specific communication command, and can usually determine which message to process first. When no channel has a pending message, the ESE idle process is scheduled. The idle process is also a sporadic process and runs to completion like all ESE processes. The *Select* procedure of the ESE scheduler determines which process to activate next.

```
PROCEDURE                          Scheduler () ;
VAR
    next_channel                      : ChannelIndex ;
BEGIN
    LOOP
        Select (next_channel) ;
        Release (next_channel.receiver)
    END
END Scheduler ;
```

Because of the policy of scheduling according to deadlines associated with channels, ESE does not maintain a ready queue of processes. Instead, channels are sorted in increasing order of deadlines. When the scheduler is executed, the channel with the earliest deadline is found in the channel table. It contains a reference to its receiving process, where the scheduler can find its entry point to schedule. The scheduler will always schedule the receiving process of the channel with the earliest deadline, and that process will not be preempted. All processes run to completion, and after initially scheduling the first process, the scheduler is only invoked every time a process completes execution.

Stellenbosch University http://scholar.sun.ac.za

A deadline is calculated for a channel when a message is sent on that channel. The deadline is equal to the current system time plus the period of the channel. A channel's deadline is therefore the earliest time at which another message can be sent on it. The receiving process of a ready channel must be scheduled before its deadline to process the current message, otherwise message collision will occur.

Scheduling the next process involves finding the channel with the earliest deadline, and calling the thread of its receiving process. To find the channel with the earliest deadline, the channels must be ordered by their deadlines. The channel ready queue can be sorted when one of the following events occurs:

1. *When a message is sent on a channel.* A list of channels with pending messages must be maintained and ordered by deadline. When a message is sent on a channel a deadline is calculated for the channel, and it is inserted into the correct position in the ordered list.

   - If the ordered ready queue of channels is implemented as a linked list, the complexity of the insertion of a channel into the queue is of order $O(n)$, where n is the number of channels.
   - If the ordered ready queue of channels is implemented as a randomly accessible queue, a binary search can be used to find the position for a new channel. The complexity of a binary search is of order $O(log\ n)$, where n is the number of channels. In order to insert a new element into the queue, its existing elements must be moved in time proportional to n.

   In both cases, sending a message introduces an *unpredictable delay* with a *known upper bound* into the execution time of a sending process.

2. *When the current process completes execution, and the scheduler selects the next process.* In this case no ordered queue needs to be kept if all channels are examined during the scheduling to determine the one with the earliest deadline. This incurs a constant $O(n)$ overhead on the cost of scheduling, and the ordering of the ready channels has no impact on the execution time of a process sending process.

As both the approaches described above have a know worst case behaviour, either can be used in the implementation of the EDF scheduling policy. ESE employs the second approach for the following reasons:

Stellenbosch University http://scholar.sun.ac.za

- No ready queue for channels is required. This simplifies the implementation, and saves data and code space. No dynamic data structures (pointers) are required for the scheduler, raising confidence in the reliability of the scheduler.
- A simple iterative routine finds the earliest deadline. The cost of this operation is constant for a given number of channels, and can be determined accurately.
- No unpredictable delay is incurred by sorting the channel ready queue when a process sends a message on a channel. The execution time of a process can be predicated accurately.

The first channel in the channel table is initialised as the input channel to the idle process. All channels receive an initial deadline of MAX (Time).

```
WITH channel_table.channels [0] DO
    name     := 'Idle Process Input Channel' ;
    user_ref := IDLE_PROC_CHANNEL ;
    receiver := idle_process ;
    deadline := MAX (Time) ;
END ;
channel_table.no_of_channels := 1 ;
```

Select selects the first channel with the earliest deadline (more than one channel may share the earliest deadline) as the next channel. A ready channel will have a deadline of less than MAX (Time). If no channel is ready Select will default to the input channel of the idle process.

```
PROCEDURE                           Select
    (VAR next_channel                   : ChannelIndex) ;
VAR
    c                                   : ChannelIndex ;
BEGIN
    next_channel := 0 ; (* Input channel to idle process *) ;
    c := 1 ;
    WHILE c < channel_table.no_of_channels DO
        IF
            channel_table.channels [c].deadline <
            channel_table.channels [next_channel].deadline
```

Stellenbosch University http://scholar.sun.ac.za


```
        THEN
            next_channel := c
        END ;
        INC (c)
    END ;
    channel_table.channels [next_channel].deadline := MAX (Time)
END Select ;
```

4.10.1 System clock granularity

ESE requires a hardware clock to determine deadlines accurately. If the system clock granularity is much coarser than the period of input events, more than one process may be scheduled between two clock ticks. When this happens, the deadlines of channels cannot be determined accurately, and the deadline of a channel may be set earlier than it should be.

Figure 11 shows a system where such a problem can occur. It shows three channels, each with the same period, and with the same execution cost for each receiving process. The system clock granularity is five times the execution time of the receiving processes. The period between events that occur in a burst can be much smaller than the timer granularity. For this reason messages on the input channels of processes 1, 2 and 3, which occur in the same interval between two system clock ticks, will receive the same deadline.

If a number of ready channels share the earliest deadline ESE's scheduler will always select the first one it encounters. If the clock granularity is coarse enough, two successive messages on the same channel can receive the same deadline. The scheduler may then repeatedly select a channel with a low channel number even though there is a channel with a higher channel number, and the same deadline.

In figure 11 three channels share a common release time, and therefore a common deadline. The blackened circles show the release times of the channels (in real time) and the unblackened circles show the deadline of each channel (in system time). The filled boxes on the ready intervals of the channels indicate the period for which the receiving process of that channel was scheduled to process the pending message. The figure shows how process 3 will not be scheduled before the deadline of its input port, because processes 1 and 2 were repeatedly scheduled before it.

From the example it should be clear that a system clock granularity, of at most the minimum period of the most frequent event, is required to ensure that the deadline scheduling policy

Stellenbosch University http://scholar.sun.ac.za

is correctly applied. The provision of such a fast clock can however place a heavy interrupt handling overhead on the system, or may be physically impossible in some systems. For example:

> The ESE clock granularity is configurable, but for the platform used in the case study in chapter 5 it defaults to a 50*ms* tick. Systems implemented on ESE on that platform have however had bounded burst signal periods of down to around 500μs. A timer tick of 500μs would have placed an intolerable interrupt handling burden on the system.

One solution to the problem is to keep an ordered deadline queue, and always insert a new channel immediately before the next later deadline. This would ensure a temporal ordering on channels with the same deadline, but would incur more overhead in finding the correct position for a new channel in the queue. An even more serious problem is the inaccuracy of the clock, and the fact that channels which receive the same deadline in this scheme may not have had the same deadline had the clock granularity been finer.

The ESE scheduler addresses the problem by artificially inserting fine grain clock ticks into the clock, thereby providing a finer grain clock. A deadline is calculated each time a message or signal is sent on a channel or port, so ideally message events must occur at different instances of the system clock. In ESE the system clock is therefore incremented by a fixed amount each time a message is sent, or a port is signaled. This has the effect of inserting fine grain clock ticks. The system clock drift caused by the inserted ticks is controlled in two ways:

1. The amount by which the clock is incremented with each communication event is chosen so that the sum of such increments will not exceed the length of one ESE timer tick interval in one such interval. This can be accurately determined because the rates of all channels are known.

2. At each hardware clock tick the clock is incremented to one tick interval later than the previous hardware clock tick. The effect of the inserted fine grain clock increment is therefore cancelled at each clock tick, and the ESE clock remains as accurate as the hardware timer which provides the tick.

Although the ESE clock will never be accurate at the fine grain, it is guaranteed not to drift from the hardware timer. If the constant with which the clock is incremented with each communication operation is chosen carefully, the behaviour of the clock is accurate enough to ensure correct scheduling.

Stellenbosch University http://scholar.sun.ac.za

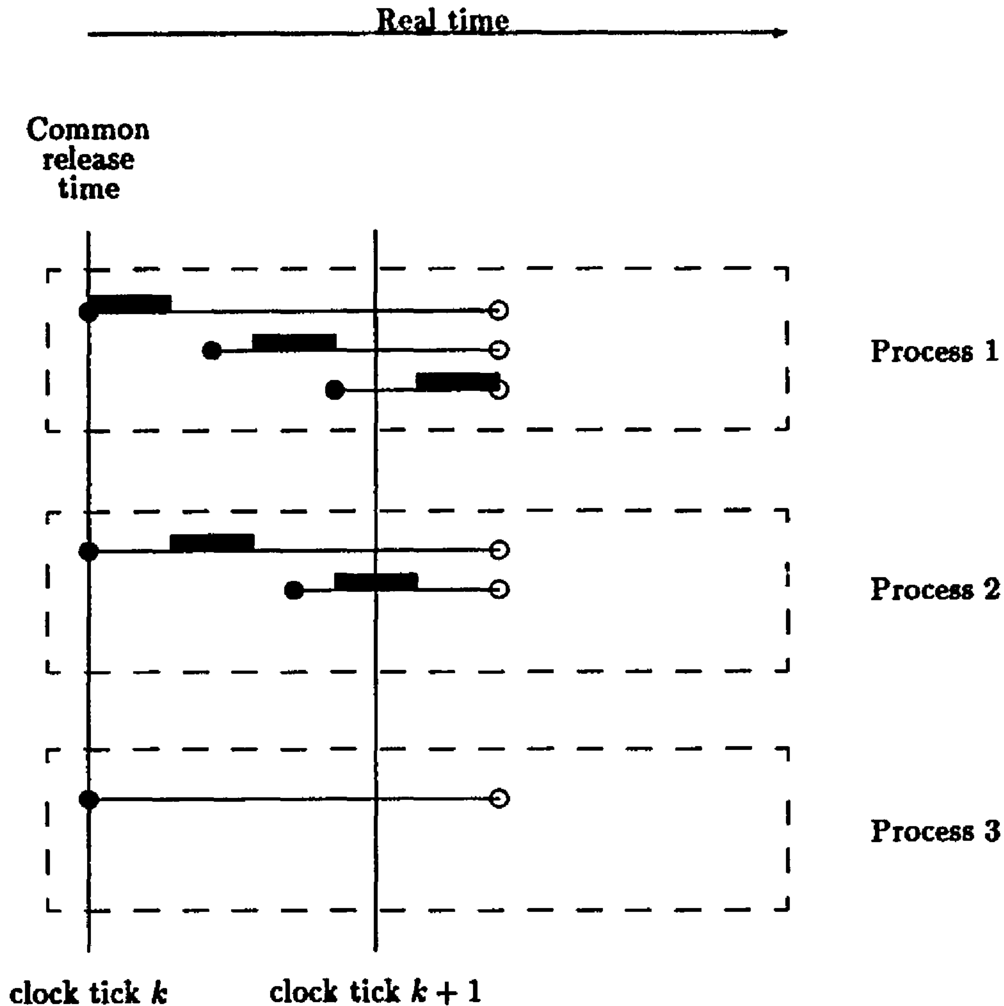


Figure 11: Scheduling with course clock ticks

Figure 12 shows the same events as the previous figure scheduled with inserted clock ticks. All deadlines are now met.

4.11 Performance of ESE

To measure the efficiency of ESE's scheduler two processes were programmed to exchange a 2 byte message 120000 times. This means that the scheduler was executed 120000 times to switch between the two processes. The test was executed on a 50 MHz Intel 80486-based PC. Table 1 shows the results of this test. Note that the context switching overhead is constant for a specific configuration of channels. This predictability is important in hard real-time systems.

Stellenbosch University http://scholar.sun.ac.za

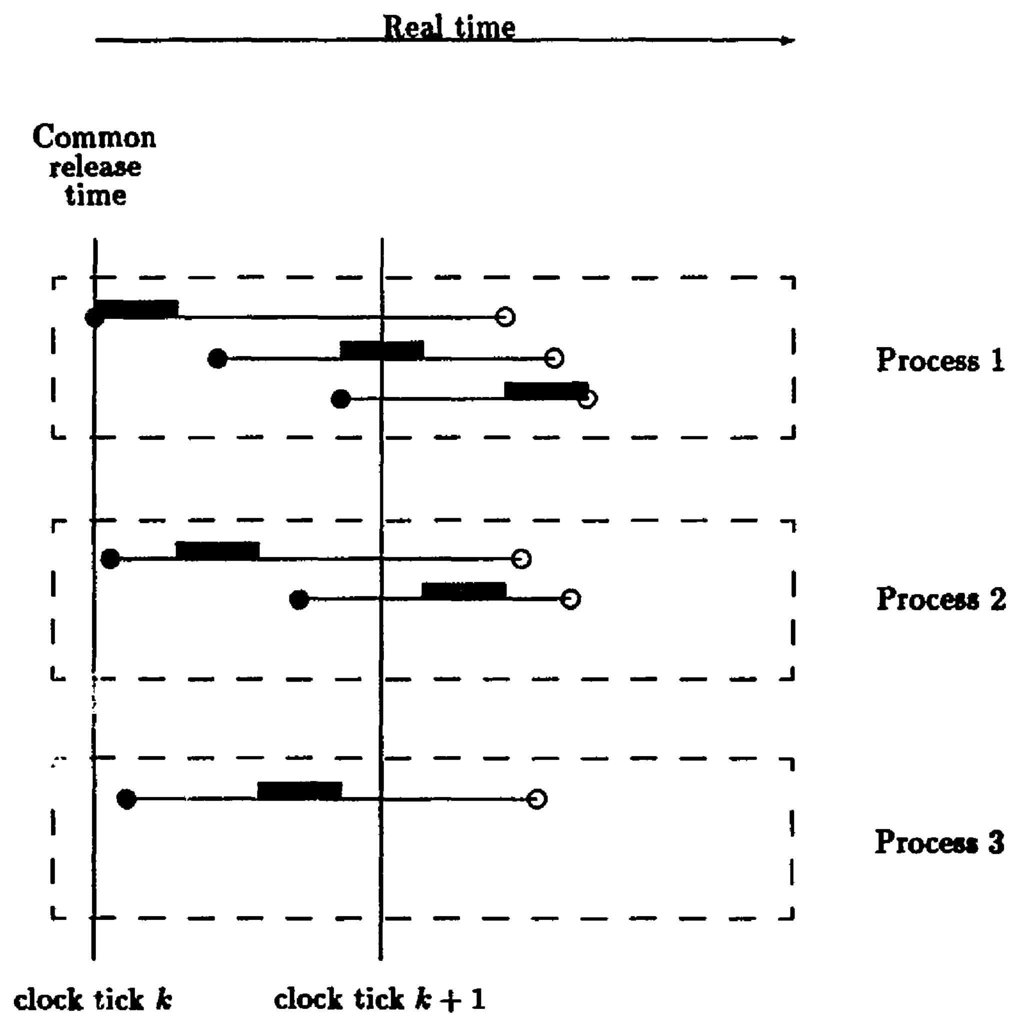


Figure 12: Scheduling with inserted fine grain clock ticks

Stellenbosch University http://scholar.sun.ac.za

No of Channels (n)	Total Time	Task Switch (t)	C = (t-28.4)/n
2	3.59s	30μs	0.8μs
3	3.6s	31μs	0.8μs
4	3.81s	32μs	0.8μs
5	3.86s	33μs	0.8μs
10	4.36s	36μs	0.8μs
20	5.22s	43μs	0.8μs
30	6.17s	51μs	0.8μs
40	7.11s	59μs	0.8μs
50	8.4s	70μs	0.8μs
100	13.45s	112μs	0.8μs
200	23.42s	195μs	0.8μs

Table 1: Task Switching Overhead

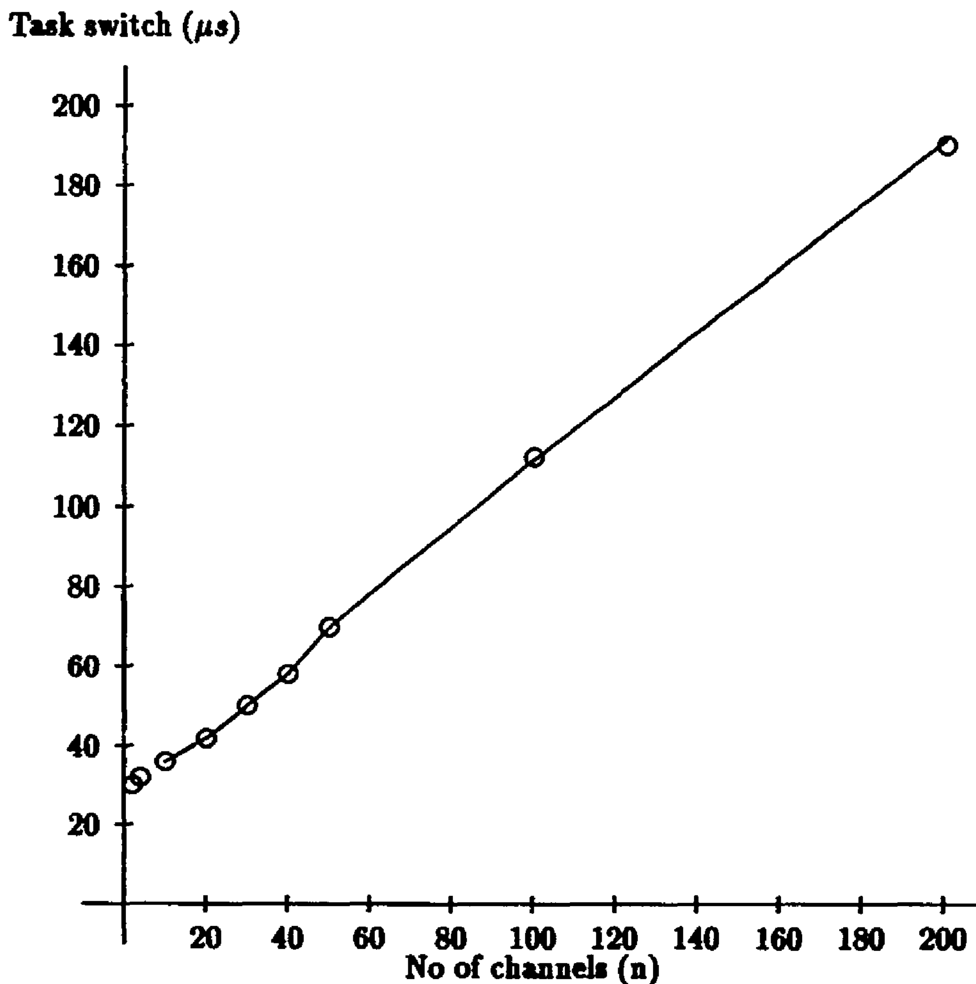


Figure 13: Task switching cost of channels

Stellenbosch University http://scholar.sun.ac.za

No of Timers	Total Time	Task Switch
1	3.59s	29.9μs
10	3.59s	29.9μs
100	3.59s	29.9μs
1000	3.64s	30.3μs

Table 2: Timer Processing Overhead

The cost of the ESE scheduler consists of two components: a fixed setup component (K), and a variable component (C) which is proportional to the number of configured channels. The cost of a task switch (t) is determined by the following formula:

$$t = K + n \times C,$$

where n is the number of configured channels. For the hardware used for these tests, the fixed component is approximately 28.4μs for all channel configurations, and the variable component 0.8 μs per channel. In other words, for any given configuration of channels on this platform, all task switches will take approximately

$$(28.4 + n \times 0.8)\mu s$$

to execute. Figure 13 shows a graphical representation of the context switching overhead incurred for different numbers of channels.

Timers are used extensively in typical reactive systems. To measure the overhead incurred by timers, different numbers of timers were configured, and the context switching rate measured. Table 2 shows the results of this test.

As can be seen from tables 1 and 2, the number of configured timers has a negligible effect on the context switching overhead, while the number of channels created for timers has a significant effect (see table 1). When a large number of timers is required, the context switching overhead incurred by creating a channel for each timer can be considerable. In the next chapter we shall examine a solution to this problem.

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

Developing Reactive Systems

5.1 Introduction

The goal of this chapter is to provide a step by step description of the process of design, analysis and implementation when one uses Jeffay's RT/PC paradigm [28] and the ESE kernel. A real example is used throughout to illustrate each step in the process. We shall see that two extensions were required to the basic ESE kernel to support industrial reactive systems: mailboxes and alarms. Having described the development process we shall discuss some observations derived from using RT/PC and ESE in practice. Finally we shall summarise the development process.

The main steps in the development process are: requirements definition, system design, RT/PC design and analysis, and implementation. At the start of each section which describes a step in the development process, we shall describe the purpose and output of that step.

5.2 X.25 case study — requirements definition

The purpose of the requirements definition step in the development process is to state clearly what the functional and performance requirements of the finished system are. The ouput of this step is a specification document which describes the requirements. We shall discuss the functional and performance requirements of the X.25 implementation briefly, but since it is not a part of Jeffay's method, we shall not develop a complete requirements specification here.

Stellenbosch University http://scholar.sun.ac.za

The requirements definition is, however, an important step in the development process. The application of Jeffay's RT/PC method requires accurate information about the rate at which events occur in the environment of the system, as well as the performance requirements of the system.

My case study is an implementation of CCITT's recommendation X.25 protocol stack, embedded on an intelligent ISA bus expansion card. The requirements of the embedded software are:

- an embedded implementation of both the link layer (LAPB) and packet level protocols of CCITT's 1988 recommendation X.25 [62];
- the packet level protocol must support up to 200 standard two way logical channels;
- a user interface to the X.25 packet level protocol, which provides access to all X.25 facilities; and
- line speeds up to 64Kbps must be supported.

Figure 14 shows a block diagram of the hardware platform on which the X.25 protocol stack is to be implemented. It is a PC/AT (ISA) bus expansion card with the following components:

Intel 80188 μprocessor, running at 10 MHz, for protocol processing. The processor has integrated interrupt and DMA controllers. This facilitates full duplex DMA access to and from the network interface controller.

256Kbyte Local Memory. *The protocol stack must execute from local RAM, and all X.25* buffers will be kept there.

Intel 82C30-10 Serial Communications Controller (SCC) which supports full duplex synchronous communication, and performs framing, bit stuffing/stripping and CRC computation.

PC Interface consisting of 32 KByte dual port RAM, an interrupt each for the card and the PC, and an I/O port each for the card and the PC. Interface primitives are exchanged between the host PC and the card through the dual port RAM window. The I/O port is used by the PC to reset the card and to generate an interrupt to the card's on board processor. The card uses the I/O port to generate an interrupt to the PC.

The stated system requirements make strenuous demands on the card's processor and available RAM. Consider the following examples:

Stellenbosch University http://scholar.sun.ac.za

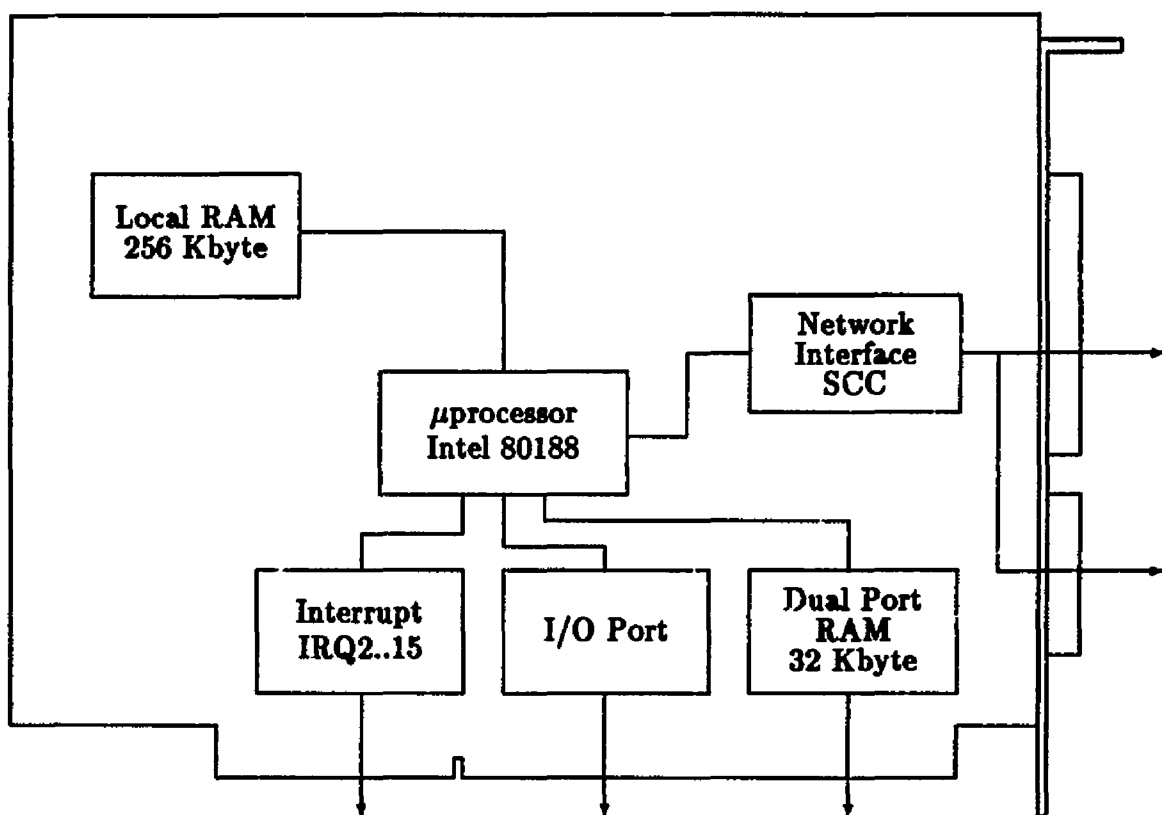


Figure 14: Block diagram of iX.25 hardware

Stellenbosch University http://scholar.sun.ac.za

1. The buffer space required to support 200 standard logical channels can be calculated as follows:
   Standard packet size = 128 bytes
   Standard packet level window size = 2 packets
   Each two way logical channel has an incoming and outgoing window = 4 packets
   200 logical channels therefore require

   $$200 \times 4 \times 128 = 102400 bytes (100KB)$$

   The link layer buffer space requirement is given by

   $$2 \times window\ size \times$$
   $$(max\ packet\ size + packet\ header + LAPB\ header + CRC)$$

   If the packet size is negotiable up to the X.25 maximum of 4096 bytes, the LAPB window size is 7, and sequence numbering of both LAPB and packet level is standard, the link layer buffer space required is

   $$2 \times 7 \times (4096 + 3 + 2 + 2) = 57422 bytes (\pm 56KB)$$

   The total buffer space requirement for the protocol stack is therefore $\pm 156KB$. This leaves plus minus 100 KB of the card's 256 KB for the code, stack and data segments of the entire implementation.

2. LAPB can receive a burst of information frames up to its maximum window size. The smallest information frames which can occur in a burst, is a burst of packet level command frames: for instance, one RR packet for each of a number of logical channels. At 64Kbps, the transfer time for one byte is:

   $$\frac{1}{64000} \times 8 = 125\mu s$$

   Each LAPB frame will consist of an opening flag, an address byte, a command byte, 3 information bytes (the packet level RR), two CRC bytes and a closing flag= 9 bytes. The minimum transfer time for a frame is therefore

   $$9 \times 125\mu s = 1.125ms$$

   Incoming frame events can therefore occur at roughly $1.125ms$ intervals.

   The throughput of the host/card interface is roughly 2Mbps. Assume that the host may send a sustained burst of data requests to X.25, each 130 bytes long (including data and

Stellenbosch University http://scholar.sun.ac.za

control). The messages from the host may therefore arrive at intervals of approximately $65\mu s$.

As can be seen from the above, the rate at which events can occur is very high. On the target Intel 80188 processor ESE processes cannot be scheduled, or consume events, at these rates. The system can therefore not deal synchronously with each event.

5.3 System design

In this section we shall create a top level design for the X.25 implementation. The purpose of this section of the development process is to understand the functionality of the X.25 protocol stack, and to decompose it into manageable blocks. At this stage we do not yet concern ourselves with Jeffay's method, but we translate the statement of requirements into a functional specification/design. We shall not develop a detailed specification here, as it is not a part of Jeffay's method.

5.3.1 Block design of X.25 system

The requirements stated in section 5.2 suggest a design consisting of three modules: a user interface, the X.25 packet level protocol, and the LAPB protocol. A fourth module, the interface to the network interface hardware, is also required. We shall call these modules *NL3IF*, *PLvl*, *LAPB* and *SCC Driver*, respectively. Figure 15 shows the modules of our design in relation to the OSI 7 layer stack and the X.25 specification.

The ISO's 7 layer model for open systems interconnection (OSI) [61] specifies 7 protocol layers, as well as service interfaces between each pair of protocol layers. For instance, the Transport layer (TL4) and the Network layer (NL3) share a service interface: the Network layer Interface. In OSI this is referred to as the n-interface between the n-layer and n+1-layer protocols. The X.25 specification [62] spans the lower three layers of OSI: Physical, Link and Network, but it does not specify any service interfaces. Our statement of requirements adds to the X.25 specification a Network layer interface. In our case study the Physical layer is implemented in hardware. In our software design, the Physical layer is represented by the device driver which interfaces with the physical layer: the serial communications controller (SCC) device driver.

Stellenbosch University http://scholar.sun.ac.za

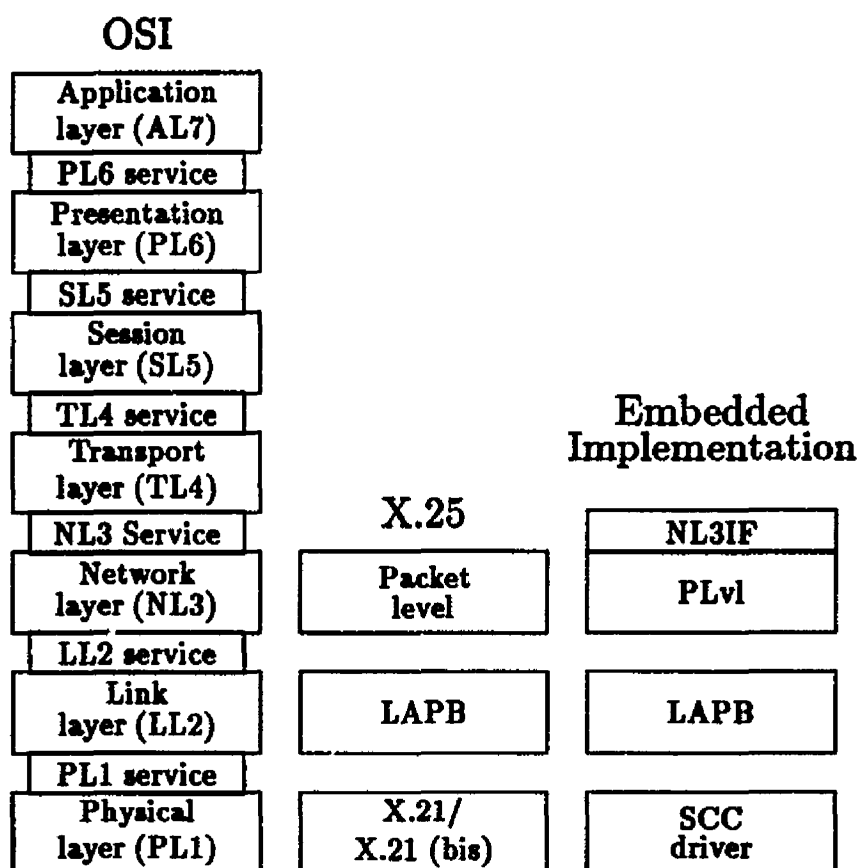


Figure 15: OSI/X.25/implementation correspondence

Stellenbosch University http://scholar.sun.ac.za

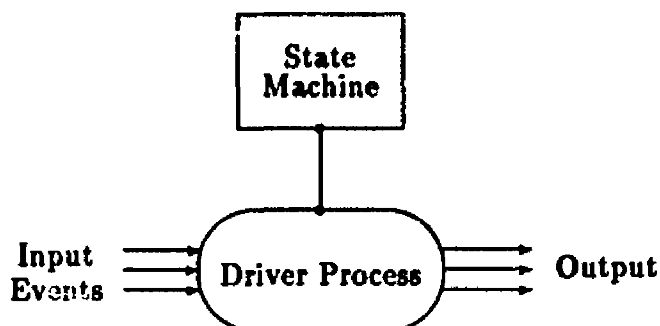


Figure 16: Protocol machine implementations

5.3.2 Process decomposition

A process is a useful abstraction for an autonomous component of a system. If processes communicate via message passing they do not need to share data structures, and concurrent access problems are eliminated. A process is also a clean way of dealing with asynchronous events in reactive systems. When an event occurs, a process is scheduled to process it.

We use the following criteria to decompose a system into processes. Reactive systems are driven by events in their environment. These events enter the system in the form of interrupts. An interrupt handler then signals the occurrence of the event to a process, which is scheduled by the scheduler. Any component of a system which is decomposed into an autonomous module, and which also processes events, is a candidate to be a process.

Of the four modules already identified in our X.25 design, three will process events: NL3IF, PLvl and LAPB. The SCC driver will consist of library routines to initiate transmission and reception, and interrupt handlers. When it is decomposed into the three processes above, our design becomes a system of communicating reactive systems. Each process receives events on its input channels, and generates output in response to these events. When the output is a message to another process, this message constitutes an input event to the reactive system represented by that process.

Figure 16 shows the structure of each process in our design. Since each process implements a protocol machine (a service protocol in the case of NL3IF) each emulates a state machine. Each protocol has a single driver process which receives all events (signals and messages) for that protocol layer. This process activates a state machine which processes the events, and generates output messages to devices and other processes (protocol layers).

Stellenbosch University http://scholar.sun.ac.za

5.4 Design and analysis of RT/PC systems

The purpose of this section is to describe how Jeffay's analysis techniques for RT/PC systems can be applied to a real system. The goal of Jeffay's method is to design and implement a system of communicating processes in such a way that the implementation is known to be *temporally correct* (Definition 5, section 3.6).

The method involves two steps: the first is to develop a *realisable* design graph (Definition 6), the second to map the realisable design onto a *viable* implementation of tasks[1] and channels.

A realisable design is one that can be implemented. If the worst case rate of each asynchronous channel in an RT/PC graph is known, it is theoretically possible to schedule its processes in such a way that the system adheres to the RT/PC paradigm. A design graph is therefore realisable if its channel rate functions can be solved.

Definition 14 *An implementation of a set of tasks is* viable *if every execution of every task is guaranteed to complete before its deadline.*

Viability as a measure of the correctness of a system of tasks is relative to the implementation strategy employed for those tasks, and is dependent upon the processor used for the implementation. In section 5.4.5 we shall identify specific conditions under which a realisable design graph has a viable implementation on the ESE kernel.

5.4.1 Realisability of a design graph

Jeffay determined conditions[2] for the realisability of three classes of design graphs: acyclic design graphs, graphs with disjoint cycles and graphs with non-disjoint cycles.

1. An acyclic design graph is realisable if there exists a path from an input device to each process in the system.

2. The transmission rate functions of a *cycle*, which is *disjoint* from all other cycles in a design graph, can be solved iff at least one channel in the cycle has a non-identity transmission rate function.

[1]We shall use the term, *task*, when we refer to a process in an implementation, and *process* when we refer to a process in the design graph. In the analysis that follows, this helps us to distinguish between the design and its implementation.

[2]Section 3.6 contains formal versions of the necessary and sufficient conditions for design graphs with cycles.

Stellenbosch University http://scholar.sun.ac.za

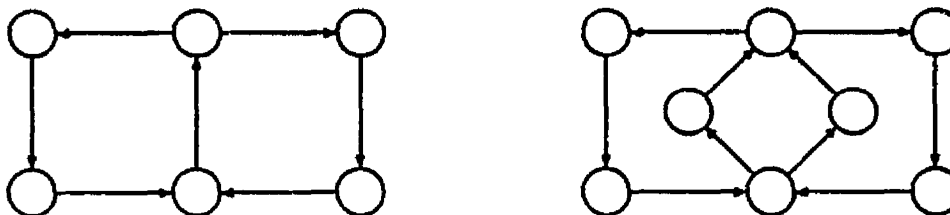


Figure 17: Nested cycles

3. The transmission rate functions of a pair of non-disjoint, simple cycles in a design graph can be solved iff:

   - there is only one process which receives messages from a distinct process in each cycle;
   - each cycle has at least one channel with a non-identity transmission rate function;
   - one of the cycles has a sequence of processes in which at least three messages are required to be input to the first process in the sequence, before a message can be emitted from the last message in the sequence.

   Figure 17 shows two examples of non-disjoint cycles. The realisability conditions for non-disjoint cycles are sufficient for the pair of cycles on the left, but not for those on the right.

5.4.2 Realisability of the X.25 design

Figure 18 shows an RT/PC design graph of our X.25 system. It has a process for each of the two protocol layers, LAPB and packet level; and one for the network layer interface. The physical layer is implemented in hardware, and is represented in the graph by input devices. The design contains two *output* devices (not shown in the design graph): the physical layer (for frame transmission) and the host interface. Due to the flow control mechanism of the X.25 protocol stack, the output devices cannot be flooded. Since they are not required in the realisability analysis, the output devices are omitted from the graph. Should any reasoning about timing involving the output devices be required, they can be added without any impact on the realisability analysis of the system. *Asynchronous channels* in the system are labeled $a \cdots j$.

The design graph contains six *input* devices:

Stellenbosch University http://scholar.sun.ac.za

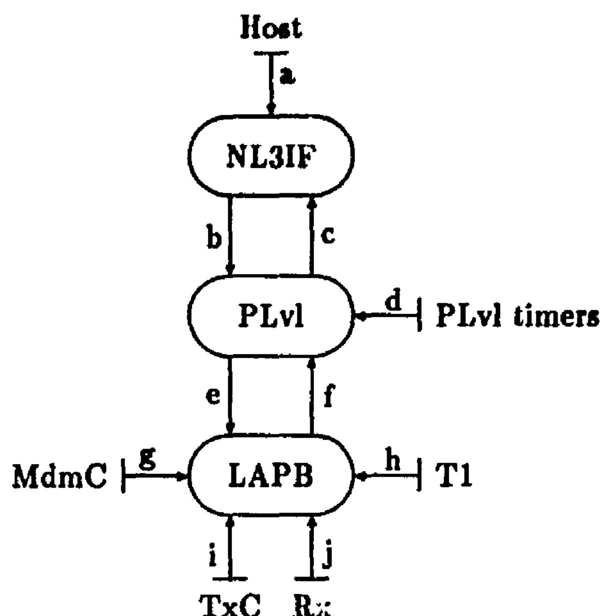


Figure 18: RT/PC design of X.25

1. *host* — control and data messages sent to the embedded protocol stack by the host PC.
2. *PLvl timers* — The packet level protocol maintains several timers which signal certain events.
3. *MdmC* — An event indicating a change in the modem status to LAPB.
4. *TxC* — An event indicating the completion of a frame transmission to LAPB.
5. *Rx* — An event indicating that an incoming frame has been received.
6. *T1* — LAPB maintains a single timer which signals the occurrence of certain events.

Channel rate functions

The rate at which messages flow through the X.25 system is determined by the rate at which the host sends data, and the rate at which messages are received on the physical layer interface. The *worst case* rate occurs when a sustained burst of messages is sent in both directions between an X.25 DTE and DCE.[3] If the DTE and DCE are correctly tuned, each received LAPB frame will contain both a packet level data packet and a LAPB frame

[3]The CCITT's recommendation X.25 defines the protocol between a DTE (Data Termination Equipment) and a DCE (Data Connection Equipment). The DTE represents the X.25 user's equipment, the DCE the point where the user's link connects to the X.25 packet switched network.

Stellenbosch University http://scholar.sun.ac.za

acknowledgement. In response to such a frame, *LAPB* will then send two messages to *PLvl*: a data and a control primitive. In this worst case scenario, each packet level data packet will contain a data message for the host, as well as a packet level acknowledgement. In response to each data packet, *PLvl* will therefore send two primitives to the user via *NL3IF*: a data and a control primitive. Based on the worst case behaviour of the system, we can now determine the following channel rate functions for the system:

- $b = a$ — All primitives from the host are passed to *PLvl* by *NL3IF*.
- $f = 2j$ — Each received frame is an information frame which causes *LAPB* to send a data primitive and a control primitive to *PLvl*.
- $c = 2 \times \frac{f}{2} = f$ — Half of all primitives which *PLvl* receives from *LAPB* ($\frac{f}{2}$) are data packets. Each packet contains data for the host as well as packet acknowledgement. For each data packet received, one data indication primitive and one control primitive is sent to the host via *NL3IF*.
- $e = 2 \times \frac{f}{2} = f$ — Each control message sent to *PLvl* on f causes it to send the next queued information packet to *LAPB* ($\frac{f}{2}$). Each data primitive which *PLvl* receives on f causes it to send a control primitive on e ($\frac{f}{2}$).

The behaviour above can be achieved only if the host interface is at least as fast as the frame reception rate — $a \geq j$. If $a > j$ the X.25 flow control mechanism will ensure that host data is transmitted only at the rate at which frames are acknowledged. If $a < j$ the message flow rate in the system will be lower, and the worst case behaviour described here will not occur.

The input devices, *MdmC*, *TxC*, *T1* and *PLvl Timers* cause very limited message flow in the system. In the worst case behaviour described here, only *TxC* and *MdmC* generate any events. Neither of these events cause any message flow out of *LAPB*, and therefore have no impact on the realisability of the system. *T1* and *PLvl timers* will only generate events when communication timeouts occur — in other words, when the flow of data through the system is much lower than the worst case described above. We can therefore ignore these input devices for the purpose of our realisability analysis, and use the reduced design graph shown in figure 19.

Figure 19 shows that even though the RT/PC design of X.25 contains non-disjoint cycles, the channel rate functions are easily solved. This is because the channel rate functions depend upon the interactions of the X.25 protocols, rather than cumulative input. The design graph in figure 19 *is realisable because all the channel rate functions can be solved.*

Stellenbosch University http://scholar.sun.ac.za

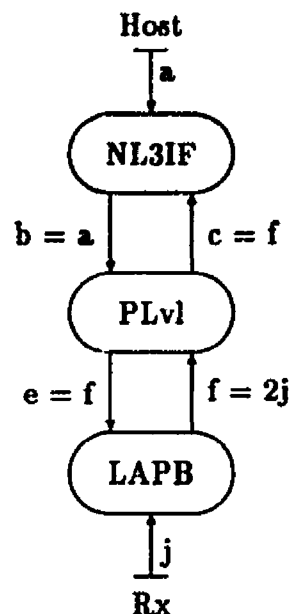


Figure 19: Reduced, annotated RT/PC design for X.25

It is possible to refine the design graph in figure 19 by refining the processes and channels shown into sets of processes and channels. For example, the data and control flows can be separated into separate channels and receiving and sending processes. In this way cycles in the design graph can be removed. However, the added complexity will not simplify the analysis. The structure of the system is such than the effective message flow in the refined graph is exactly the same as in the abstracted graph shown here.

ESE kernel extension — mailboxes and alarms

Two practical issues, which arose in using RT/PC and the ESE kernel in practice, prompted the addition of asynchronous channels, which can buffer multiple messages, to the basic ESE kernel.

The first situation occurs when one input message (event) can cause its receiver process to send more than one output message to the same destination. Consider the worst case behaviour of the X.25 system, described above. Each *LAPB* frame received causes two messages to be sent to PLvl. If both messages were sent on the same channel, the channel would overflow, because ESE would not preempt *LAPB* so that *PLvl* could process the first message. There are a number of solutions to this problem:

Stellenbosch University http://scholar.sun.ac.za

1. Both primitives could be packed into one message, and sent together. However, this would violate the conventions of the OSI n-service interface, which requires separate control and data primitives. It would also require the receiver of each message to parse it for the primitives it contains, and a separate thread to be executed for each different message. But this approach does not suit the design of a reactive system, which processes events one at a time.

2. Separate channels can be created for different types of messages exchanged between two processes. This is a viable solution, since the deadline scheduling policy can ensure that messages are processed in the order they were sent. Care has to be taken, however, of the number of channels created. Section 4.11 shows that the number of channels in a system has a significant impact on the fixed scheduling overhead. This becomes especially significant when a less powerful embedded processor, such as an Intel 80188, is used.

3. Messages can be buffered, and a signal used to activate a receiver process once for every buffered message. When the buffering and signaling functions are combined, the result is equivalent to an asynchronous channel which can buffer multiple messages.

The second motivation for buffered asynchronous channels involves the use of timers. In some systems provision has to be made for a large number of timers. The X.25 packet level, for instance, requires a separate timer for each logical channel. In our implementation, which supports up to 200 logical channels, this would imply 200 extra channels, and that would have a significant impact on the cost of scheduling. The answer to this problem is to have a second type of timer in ESE: one which signals expiry on a buffered asynchronous channel rather than the standard ESE asynchronous channel. Several of these timers can then share one buffered channel. For the reasons outlined above, *Mailboxes* and *Alarms* were added to the basic ESE kernel.

A *mailbox* is an asychronous channel which can buffer up to a predefined number of messages. Like an ESE *input port*, the receiving process of the mailbox will be scheduled once for each buffered message. The semantics of mailboxes are exactly the same as for input ports, and the receiving process of a mailbox will be scheduled at least at the rate corresponding to the period of the mailbox. Since a mailbox can receive messages from a number of tasks, it does not identify its sender task. The buffer slots required by a mailbox are allocated when the mailbox is created. This is done to fix the cost of sending a message to a mailbox. If the buffer space for a slot is allocated when the send occurs, there may be an unpredictable delay, or in the worst case the system may be unable to allocate the required space.

Stellenbosch University http://scholar.sun.ac.za

When a message is put into a mailbox, it is copied into the buffer space maintained by the ESE kernel. If it is the only message in the mailbox, a deadline is calculated for the mailbox. ESE's scheduler and *Receive* command treat mailboxes as part of the generic set of channels, which consists of asynchronous channels, input ports and mailboxes.

An *alarm* is a timer which signals its expiry on a mailbox rather than an asynchronous channel. It is started and stopped in the same way as an ESE *timer*. Because alarm expiry notifications are buffered in a mailbox, an alarm cannot be used to provide an accurate time tick to a process. When accurate timing is required, an ESE *timer* must be used. When a timer is used only to indicate that an event (timeout) had occurred, and this does not require accurate timing, an alarm can be used. The overhead involved in sending to and receiving from a mailbox is marginally higher than for an asynchronous channel. Since there is no semantic difference between timers and alarms, alarms should therefore only be used when the number of timers required would add a significant number of channels to the system.

5.4.3 Implementing the design graph

An RT/PC design graph models a system in terms of graph vertices and edges. Edges correspond to message passing channels (both synchronous and asynchronous); vertices to processes, in/output devices, data repositories, etc. To implement a design one must map a design graph onto a set of ESE tasks and channels in such a way that the temporal correctness of the implementation can be determined.

In his thesis [28], Jeffay specifies necessary and sufficient conditions for the viability of a set of sporadic tasks which communicate on asynchronous channels. The ESE kernel has the following features which determine how a design graph is mapped onto a set of sporadic tasks, and the viability conditions that apply to such a mapping:

- *Non-preemptive scheduling* ensures that mutual exclusion is always ensured for any set of tasks that share data structures and resources. Because of the non-preemptive scheduling policy no receiver will ever consume a message immediately when it is sent — the sending task must always complete its execution cycle before another process can be scheduled. Each ESE channel must therefore buffer one message until the receiver of that channel is scheduled.

- *No data repositories or other mutual exclusion mechanisms* are required in ESE, because mutual exclusion is guaranteed by the non-preemtive scheduling policy.

Stellenbosch University http://scholar.sun.ac.za

- *No lockable shared resources* are supported by the ESE kernel. All ESE tasks are therefore single phase tasks [28] and no ready process can ever be blocked by a locked resource.
- *The ESE kernel scheduler selects the channel with the earliest deadline*, and activates the receiving process of that channel. Because tasks are scheduled non-preemptively a task's deadline is the same as that of the channel whose message it consumes at any point in time. Due to the non-preemptive scheduling policy a task with multiple input channels will behave exactly like a set of tasks: one for each channel.

Mapping a design graph onto an ESE implementation

A design graph can be mapped to an ESE implementation as follows:

1. *ESE tasks*

   Jeffay recommends an implementation strategy of creating a task for each asynchronous channel. This is because a process with multiple input channels would have a period of $\frac{1}{r_{in}}$ where r_{in} is the sum of the rates of all input channels of the process. For tasks with shared resources it is desirable to have the largest possible period, because the more blockage a task can endure, the less it imposes on other tasks. If a task is created for each channel, the period of each task will be longer than the $\frac{1}{r_{in}}$ of the corresponding multi-input process [28].

   In ESE we can create a *task* for each process in the design graph without the problem described above. This is due to the fact that ESE does not support lockable shared resources, and schedules tasks non-preemptively on the deadlines of ready channels. Because ESE determines which message (from which channel) a scheduled process can consume, and because tasks are single threaded, the effect is the same as creating a task for each asynchronous channel in the design graph.

2. *ESE channels*

   ESE supports only asynchronous channels. Synchronous channels reduce to standard procedure calls due to the non-preemptive scheduling policy of single phase ESE tasks. An *asynchronous ESE channel* is therefore created for each asynchronous channel in the design graph. The sender and receiver tasks of a channel are those created for the corresponding sender and receiver processes in the design graph.

3. *ESE signal ports*

   A signal port is created for every event that originates from an interrupt handler. This

Stellenbosch University http://scholar.sun.ac.za

enbles ESE to schedule a device driver task which will process the event.

4. *ESE mailboxes*
   A mailbox can be created, instead of two or more channels, if it is desirable to reduce the number of channels in a system. It can also be created for a set of alarms to signal their expiry on. A mailbox may also be used to reduce the flow rate of messages on a certain path in a system. The mailbox can be used to buffer and release messages at the desired rate. In this way an otherwise non-viable system may be made viable.

Rules for the use of mailboxes

1. The number of unprocessed messages to be buffered by a mailbox must be bounded. If the system does not impose some form of flow control over the processes communicating through the mailbox, the mailbox will overflow.

2. A mailbox must be created with enough slots to buffer the largest possible number of unprocessed messages which may be sent to it, otherwise it will overflow.

3. The period of the mailbox determines a lower bound on the rate at which the messages in the mailbox will be consumed. If no other channels have earlier deadlines, the mailbox receiver will be scheduled faster than the rate corresponding to its period.

4. Because mailbox slots are allocated when the mailbox is created, each slot must be large enough to hold the largest possible message that can be sent to the mailbox. If a mailbox is created to combine two channels which have a large difference in maximum message size, the mailbox must provide buffer space for the smaller messages in slots large enough to hold the largest message. This wastes space, and in such a case it may be more desirable to create two separate asychronous channels. A mailbox should be used to combine channels only if they have similar maximum message sizes.

Refining design graph asynchronous channels

The purpose of the design graph is twofold: to model a system as clearly and simply as possible; and to determine the realisability of the design. The first purpose requires the abstraction of unnecessary detail; the second, sufficient detail to calculate the message flow rates accurately.

In some designs it is possible to combine a number of asynchronous channels into one in the design graph. In the implementation one may then create more than one channel for a

Stellenbosch University http://scholar.sun.ac.za

corresponding asynchronous channel in the design graph. The refined design would still be realisable, since the effective message flow rate between the processes remained the same.

Example In the X.25 design both the LAPB and packet level protocols can generate both a data and a control message to each other in one execution cycle. Because of non-preemptive scheduling this will cause channel overflow if both messages are sent on the same channel. To solve this, we create separate control and data channels, and if the original design was realisable, the refined one will be as well. The channel overflow problem is eliminated in the refined design, and the temporal ordering of messages is preserved by the deadline scheduling of receiver processes.

In this example a mailbox could also have been used to solve the problem. This would have required buffer space to be allocated for the largest message which the mailbox could receive. Since the largest control message is several times smaller than the largest data message, it is more memory efficient to create separate channels in this case. If all messages have the same size, a mailbox would be a better solution because it would reduce the number of channels. In this way the scheduling overhead is reduced.

5.4.4 Implementation mapping of the X.25 example

In section 5.4.2 we determined the realisability of a model of an X.25 system (figure 19). That model was made deliberately abstract to illuminate the logical structure of the system, and to avoid implementation detail. To reason about the viability of a system, we need a model which represents the planned implementation in the exact detail of tasks, channels and mailboxes. Figure 20 shows the refined model which represents our planned implementation.

Tasks

The implementation model has an ESE *task* for each process in the design graph of the previous section: *NL3IF*, *PLvl* and *LAPB*.

Interrupt handlers and signal ports

The input devices of figure 19 have been refined to input devices which generate events to *interrupt handlers*. The interrupt handler signals the occurrence of the event on an ESE *signal port*. The input device represents the physical device — it is not implemented in software.

Stellenbosch University http://scholar.sun.ac.za

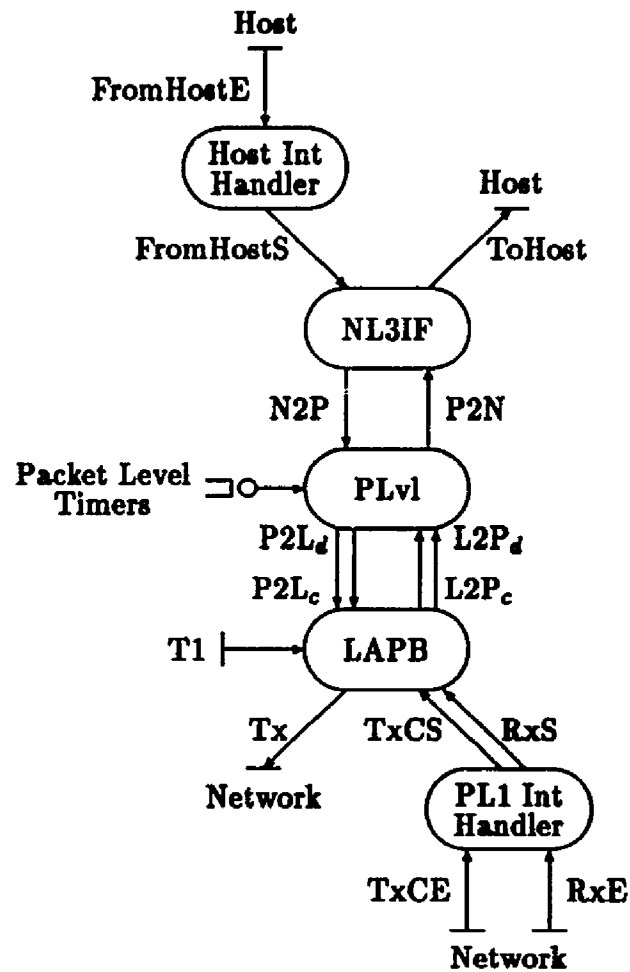


Figure 20: Refined model of X.25 for implementation

Stellenbosch University http://scholar.sun.ac.za

The reason for this refinement is to model the cost of processing an event from an input device accurately. Interrupt handlers in figure 20 are shown as processes. This is because an interrupt handler is semantically equivalent to an ESE process, except for the fact that it is not scheduled by ESE's scheduler.

Host	The host PC
FromHostE	The event which causes an interrupt to the X.25 system
Host Int Handler	The interrupt handler for host interrupts
FromHostS	A signal port on which Host Int Handler signals the arrival of a host message to NL3IF
Network	The physical layer hardware
TxCE	The event which causes a transmission completion interrupt to the system
RxE	The event which causes a frame reception interrupt to the system
PL1 Int Handler	The interrupt handler for physical layer interrupts
TxCS	A signal port on which PL1 Int Handler signals an transmission completion interrupt to LAPB
RxS	A signal port on which PL1 Int Handler signals a frame reception to LAPB

Asynchronous channels

The channels of the abstract design graph have been refined in figure 20 to accurately model the message flow which will occur in the implementation, and the channels between *PLvl* and *LAPB* have been refined into pairs of data and control channels.

N2P	NL3IF to PLvl channel: all primitives
P2N	PLvl to NL3IF channel: all primitives
$P2L_d$	PLvl to LAPB channel: data primitives only
$P2L_c$	PLvl to LAPB channel: control primitives only
$L2P_d$	LAPB to PLvl channel: data primitives only
$L2P_c$	LAPB to PLvl channel: control primitives only

Stellenbosch University http://scholar.sun.ac.za

Timers and mailboxes

Both PLvl and LAPB require timeouts to signal certain events. LAPB's single timer, T1, is created as an ESE timer, and signals expiry on a channel. PLvl requires many timers, and since ESE's scheduling overhead increases with the number of channels in the system, these timers are implemented as ESE alarms which signal expiry on a single mailbox.

T1	A timer, and a channel called T1
Packet Level Timers	An alarm for each PLvl timer, and a mailbox called Packet Level Timers. The mailbox has one slot for each alarm which may expire.

5.4.5 Viability analysis

The first step towards implementation of the RT/PC design graph is a mapping of the design onto a set of tasks and channels, supported by the ESE kernel. This was the subject of sections 5.4.3 — 5.4.4. The purpose of the viability analysis is to determine whether the implementation of that mapping will be temporally correct. Viability analysis takes into account the implementation strategy, as well as the processor on which the system is to be implemented. A viable implementation of a design graph will always be temporally correct. This means that all execution cycles of all tasks in the implementation are always guaranteed to complete before their deadlines. We use an implementation strategy of single phase, non-preemptive, non-resource consuming sporadic tasks (for the purpose of analysis the CPU is not a resource). ESE processes are *single-phase* because ESE does not support lockable resources, and because they are not preempted.

According to Jeffay the viability conditions for this implementation strategy are:

Definition 15 *A* **sporadic task** *T is a tuple (c, p) where:*

c = *computational cost: the time to execute task T to completion on a dedicated uniprocessor, and*

p = *period: the shortest possible interval between successive requests for execution of task T.*

Definition 16 *A set of sporadic tasks, $\tau = \{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by period, can be scheduled non-preemptively without inserted idle time for all possible release times iff:*

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1 \qquad (1)$$

Stellenbosch University http://scholar.sun.ac.za

and

$$\forall k, 1 \leq k \leq n : p_k \geq \max_{i:p_i > p_k} \left(c_i + \max_{0 < l < p_i - p_k} \left(-l + \sum_{j=1}^{i-1} \lfloor \frac{p_k + l - 1}{p_j} \rfloor c_j \right) \right) \tag{2}$$

The meaning of the two conditions are:

1. For any task, $\frac{1}{p}$ gives the rate of that task. The execution cost multiplied by the rate ($c \times \frac{1}{p}$) gives the fraction of the standard time unit that the task can consume in the worst case. The sum for all tasks of $\frac{c_i}{p_i}$ can therefore not exceed one standard time unit.
2. For any task, the remainder of its period less its execution cost must be large enough to cover any delay the task may experience due to other ready tasks being scheduled before it.

Note: When all tasks have the same period only condition 1 needs to be tested.

How do we apply these conditions to an ESE implementation? Since ESE tasks are sporadic, and scheduled non-preemptively, we can treat each channel as if it has a dedicated process which processes only messages from that channel. We therefore use the channels of a design as "tasks" in the viability analysis, with the cost of processing a message on each channel as the cost of that "task".

For a feasible set of tasks, condition 1 is easily verified. Jeffay showed that determining condition 2 has $O(n^2 p_n)$ complexity, where n is the number of tasks, and p_n is the period of the task with the lowest rate of execution requests. However, since the number of channels ("tasks") in a typical implementation of this kind will be reasonably low (typically 30 or less), condition 2 is determined reasonably quickly (see section 5.5).

5.4.6 Viability of the X.25 example

In order to determine the viability of the X.25 example we need accurate values for two parameters: the worst case (shortest) period of execution requests for each task, and the worst case (highest) cost of executing each task. For each channel in the system we consider a separate task — the period of each task is the same as the period of its corresponding channel, and its cost is equal to the cost of processing a message on its corresponding channel. Table 3

Stellenbosch University http://scholar.sun.ac.za

shows the cost ($c_i \equiv C(i)$) and the channel rate ($r_i \equiv R(i)$) for each channel. The execution costs (c_i) were measured with an in-circuit emulator.

Table 3 does not show any timer channels. This is because the timers will not expire during the worst case behaviour of X.25 analysed here. The channel rate functions in table 3 are solved for the worst case rate of incoming 128 byte X.25 data packets. The total length of a link layer frame containing such a packet is:

$$\begin{array}{rll} 2 & flag\,bytes & + \\ 2 & FCS\,bytes & + \\ 2 & LAPB\,header\,bytes & + \\ 3 & packet\,level\,header\,bytes & + \\ 128 & data\,bytes & \\ \hline 137 & bytes & \end{array}$$

This means that 137 bytes must be transmitted for each frame. At 64 Kbps we can therefore transmit

$$\frac{64000bits/s}{8bits/byte} \times \frac{1}{137bytes/frame} = 58.4frames/s$$

For the sake of the initial viability analysis we round the number of frames per second up to 60. If we use the values of table 3 to calculate viability condition 1, we arrive at the following result:

$$\sum_{i=1}^{14} \frac{c_i}{p_i} = 2.42 > 1$$

The system is therefore not viable for the worst case behaviour of 128 byte data packets transmitted at 64Kbps, and we have to reduce the input rate to render the system viable.

Table 4 shows the system with the rate of incoming frames adjusted to 20. For this solution of the channel rate functions

$$\sum_{i=1}^{14} \frac{c_i}{p_i} = 0.97 \leq 1$$

and viability condition 1 is satisfied.

The analysis above was done using a commercially available spreadsheet package. This is an ideal way of solving the equations and determining viability condition 1, since it allows one to interactively adjust the rates of input devices, and to see the resultant message flow in the system.

Stellenbosch University http://scholar.sun.ac.za

The analysis of viability condition 1 tells us that the hardware is not capable of handling the worst case frame transmission rate at 64 Kbps. How does this influence our implementation? At a line speed of 19200 bps, 17.5 frames, each containing 137 bytes, can be transmitted per second. This means that the system can be scheduled according to the EDF policy, and that all deadlines will be met, at line speeds up to 19200 bps. At higher line speeds the higher message rate from input devices will cause channels to overflow. This situation is remedied by setting the period of the ESE *signal port*, which is used to signal frame arrival (*RxS*), to the maximum tolerable rate of 20 frames per second. The result is that ESE will schedule the receiving process of this signal no faster than 20 times per second when other processes with earlier deadlines are ready. Frames which arrive at a rate faster than 20 per second must be buffered or lost. Since the X.25 link layer protocol (LAPB) uses a window flow control mechanism, frames will not be lost — frames can be buffered up to the maximum window size, at which point the flow control mechanism will prevent the remote X.25 stack from sending any further frames. The net effect is a reduction of the rate of the input device, *RxE*, to the viable rate (20). As soon as at least one of the buffered frames has been processed and acknowledged, the remote X.25 stack can transmit again. The system can therefore operate at line speeds higher than 19200 bps, and ESE will ensure that bursts of frames which arrive too quickly to be scheduled will not overflow asynchronous channels.

Viability condition 2 is determined by a special program. Figure 21 shows the input file to this program for the X.25 case study. Each line in the input file contains the name, minimum period and maximum execution cost of a channel. Period and cost are given in microseconds.

Figure 22 and figure 23 show the output file generated by the program in response to the input file shown in figure 21. It first prints the channel information from the input file sorted by non-decreasing period of the channels, and then checks viability condition 2 for each channel. If a channel satisfies viability condition 2, it is marked, OK. If a channel does not satisfy viability condition 2, it is marked, FAILED.

As can be seen from the output file, our X.25 implementation model satisfies viability condition 2. The channel periods of the implementation on the ESE kernel can now be adjusted to those used in the successful viability analysis. The ESE kernel is guaranteed to schedule the viable implementation in such a way that all tasks always meet their deadlines — all consumers are guaranteed to consume every message before another message is received on the same channel.

The X.25 implementation model described here was implemented on the hardware platform described in section 5.2. The measured performance (in terms of maximum throughput) of

Stellenbosch University http://scholar.sun.ac.za

	Channel	c_i	$r_i = \frac{1}{p_i}$	
1	$FromHostE$	0.0005	$2 \times R(RxE)$	= 120
2	$FromHostS$	0.001282	$\frac{1}{\frac{1}{R(FromHostE)}+C(FromHostE)}$	= 113
3	$ToHost$	0.001933	$\frac{1}{\frac{1}{R(P2N)}+C(P2N)}$	= 59
4	$N2P$	0.008562	$\frac{1}{\frac{1}{R(FromHostS)}+C(FromHostS)}$	= 99
5	$P2N$	0.001031	$\frac{2}{\frac{1}{R(L2P_d)}+C(L2P_d)}$	= 63
6	$P2L_d$	0.005431	$\frac{1}{\frac{1}{R(L2P_c)}+C(L2P_c)}$	= 34
7	$P2L_c$	0.001381	$\frac{1}{\frac{1}{R(L2P_d)}+C(L2P_d)}$	= 31
8	$L2P_d$	0.006696	$\frac{1}{\frac{1}{R(RxS)}+C(RxS)}$	= 40
9	$L2P_c$	0.004321	$\frac{1}{\frac{1}{R(RxS)}+C(RxS)}$	= 40
10	Tx	0.000089	$\frac{1}{\frac{1}{R(P2L_d)}+C(P2L_d)}$	= 29
11	$TxCS$	0.001	$\frac{1}{\frac{1}{R(TxCE)}+C(TxCE)}$	= 28
12	RxS	0.00738	$\frac{1}{\frac{1}{R(RxE)}+C(RxE)}$	= 56
13	$TxCE$	0.00053	$R(Tx)$	= 29
14	RxE	0.001161	K	= 60

$$\sum_{i=1}^{14} \frac{c_i}{p_i} = 2.42$$

Table 3: Viability Condition 1 — Maximum Channel Rate

Stellenbosch University http://scholar.sun.ac.za

	Channel	c_i	$r_i = \frac{1}{p_i}$	
1	$FromHostE$	0.0005	$2 \times R(RxE)$	$= 40$
2	$FromHostS$	0.001282	$\frac{1}{\frac{1}{R(FromHostE)} + C(FromHostE)}$	$= 39$
3	$ToHost$	0.001933	$\frac{1}{\frac{1}{R(P2N)} + C(P2N)}$	$= 30$
4	$N2P$	0.008562	$\frac{1}{\frac{1}{R(FromHostS)} + C(FromHostS)}$	$= 37$
5	$P2N$	0.001031	$\frac{2}{\frac{1}{R(L2P_d)} + C(L2P_d)}$	$= 31$
6	$P2L_d$	0.005431	$\frac{1}{\frac{1}{R(L2P_c)} + C(L2P_c)}$	$= 16$
7	$P2L_c$	0.001381	$\frac{1}{\frac{1}{R(L2P_d)} + C(L2P_d)}$	$= 15$
8	$L2P_d$	0.006696	$\frac{1}{\frac{1}{R(RxS)} + C(RxS)}$	$= 17$
9	$L2P_c$	0.004321	$\frac{1}{\frac{1}{R(RxS)} + C(RxS)}$	$= 17$
10	Tx	0.000089	$\frac{1}{\frac{1}{R(P2L_d)} + C(P2L_d)}$	$= 15$
11	$TxCS$	0.001	$\frac{1}{\frac{1}{R(TxCE)} + C(TxCE)}$	$= 15$
12	RxS	0.00738	$\frac{1}{\frac{1}{R(RxE)} + C(RxE)}$	$= 20$
13	$TxCE$	0.00053	$R(Tx)$	$= 15$
14	RxE	0.001161	$R(K)$	$= 20$

$$\sum_{i=1}^{14} \frac{c_i}{p_i} = 0.97$$

Table 4: Viability Condition 1 — Adjusted Channel Rate

Stellenbosch University http://scholar.sun.ac.za


```
# iX.25 Viability Analysis
# Input file for Viable2.EXE
# Test for Jeffay's Second Viability Condition

#ChannelName    Period    Cost
#############################
FromHostE       25000     500
FromHostS       25641     1282
ToHost          33333     1933
N2P             27027     8562
P2N             32258     1031
P2LD            62500     5431
P2LC            66667     1381
L2PD            58824     6696
L2PC            58824     4321
Tx              66667     89
TxCS            66667     1000
RxS             50000     7380
TxCE            66667     530
RxE             50000     1161
```

Figure 21: Input File for Viability Condition 2 Checker

the implementation is within 4% of the throughput predicted in the viability analysis above.

5.5 RT/PC and ESE in practice — some observations

Jeffay's RT/PC paradigm and the ESE kernel have been applied to a number of reactive systems: the embedded X.25 application described in this chapter; an embedded implementation of the Frame Relay protocol; a MAC[4] layer bridge; and various DOS based user interface programs with a menu driven interface. It is currently being used in the implementation of subsystems of a satellite. The following are some observations from two years of experience with the paradigm in practice.

[4]The Medium Access Control sublayer for broadcast networks [57], for example the IEEE 802 family of specifications.

Stellenbosch University http://scholar.sun.ac.za


```
Viability Analysis: Condition 2
Output from Viable2.EXE

Channels sorted by non decreasing period
##########################################
FromHostE       25000          500
FromHostS       25641         1282
M2P             27027         8562
P2M             32258         1031
ToHost          33333         1933
RxS             50000         7380
RxE             50000         1161
L2PD            58824         6696
L2PC            58824         4321
P2LD            62500         5431
P2LC            66667         1381
Tx              66667           89
TxCS            66667         1000
TxCE            66667          530
```

Figure 22: Output File of Viability Condition 2 Checker

```
Delay check per channel
#######################
FromHostE  p = 25000 max delay = 15696 OK
FromHostS  p = 25641 max delay = 16337 OK
M2P        p = 27027 max delay = 17723 OK
P2M        p = 32258 max delay = 22074 OK
ToHost     p = 33333 max delay = 23149 OK
RxS        p = 50000 max delay = 39816 OK
RxE        p = 50000 max delay = 39816 OK
L2PD       p = 58824 max delay = 48640 OK
L2PC       p = 58824 max delay = 48640 OK
P2LD       p = 62500 max delay = 50021 OK
P2LC       p = 66667 max delay = 1000 OK
Tx         p = 66667 max delay = 1000 OK
TxCS       p = 66667 max delay = 530 OK
TxCE       p = 66667 max delay = 0 OK
```

Figure 23: Output File of Viability Condition 2 Checker: Continued

Stellenbosch University http://scholar.sun.ac.za

5.5.1 The area of applicability of ESE

RT/PC and the ESE kernel can be used effectively in any reactive application where the number of processes and channels remain constant. If processes and channels can be added or removed during the lifetime of the system, the channel rate functions will change, and realisability and viability analysis become impossible.

RT/PC and ESE are particularly useful in applications where accurate reasoning about time is required. Since deadlines can be guaranteed, hard real-time systems can be built this way, provided the system is viable on the target processor.

ESE is a suitable platform to develop embedded systems on. It is efficient in terms of processor and RAM requirements, thus maximising the amount of resources (CPU and RAM) available to the application. The RT/PC paradigm allows systems to be designed with the minimum of buffering. Mailboxes and signal ports provide mechanisms to control the rate at which events enter the system. This is important in systems in which a burst of events may render a system nonviable on a specific processor.

ESE can be used fruitfully in non real-time environments. If all channels are created with the same period, ESE effectively schedules ready tasks on a round robin basis. General event driven systems, such as user interface programs are then easily implemented with the minimum of analysis and tuning.

5.5.2 The effort of designing for RT/PC and ESE

Since I originally implemented ESE and the embedded X.25 application, five other developers have used ESE as a platform to implement various embedded and DOS based applications. All report that designing for ESE is at least as simple as designing for any other multitasking kernel used by these programmers, and that the implemented systems were very efficient. The analysis that is possible for RT/PC systems facilitates accurate performance and requirements predictions.

5.5.3 Realisability and viability analysis

Determining the realisability of a design is simple if one applies Jeffay's guidelines. An RT/PC design graph is realisable if its channel rate functions can be solved. It was found in practice that the realisability of the system was usually apparent upon inspection of the channel rate

Stellenbosch University http://scholar.sun.ac.za

functions, and that it was unnecessary to resort to the categories of realisable design graphs defined by Jeffay.

Viability condition 1 can be calculated with any commercially available spreadsheet package. A special program determines viability condition 2 for any set of sporadic tasks. For the X.25 example it computed viability condition 2 in less than 2 seconds. Computation of the viability conditions therefore presents no problem.

5.5.4 Performance of ESE implementations

A good example of the performance attainable by systems implemented on ESE, is an embedded implementation of the Frame Relay protocol [2] on the same hardware platform as the embedded X.25 implementation described in this chapter. The scheduling overhead in this implementation constitutes only 6.2% of the processing time for each message that flows through the system.

5.6 Summary of method

The purpose of this section is to give a guide to development with Jeffay's RT/PC paradigm and the ESE kernel. It also includes the steps of requirements definition and system design to complete the development process.

5.6.1 Requirements definition

This is a very important step in the development process. Since RT/PC involves a priori analysis of real-time behaviour, a small change in the requirements can render the implementation non-viable. It is therefore important to be clear about the requirements before the design and analysis of the system is started.

5.6.2 System design

The purpose of the system design phase is to develop a clear specification of the functionality and performance of the modules that will implement the system. The system is decomposed in terms of data flow and autonomous modules, and a process decomposition is identified.

Stellenbosch University http://scholar.sun.ac.za

5.6.3 RT/PC design

Once the system is decomposed into a system of communicating processes, an RT/PC design graph is developed which shows the interconnection of the processes via synchronous and asynchronous communication channels, and the input and output devices which are the source and sync of events. Mutual exclusion regions are identified, and the graph is annotated with channel rate functions.

5.6.4 Realisability analysis

An RT/PC design is realisable, i.e. schedulable according to the RT/PC paradigm, if the channel rate functions can be solved. The realisability of a design graph depends on the presence and nesting of cycles in the graph. A graph without cycles is always realisable — given a fast enough processor, the system can always be scheduled in such a way that no process ever misses a deadline. Jeffay gives necessary and sufficient conditions for the realisability of graphs with disjunct and nested cycles.

5.6.5 Implementation mapping

An RT/PC design graph is an abstract system represented by the RT/PC design icons. In order to determine whether a realisable design can in fact be supported by a given hardware platform, it has to be mapped to an implementation strategy supported on the platform. For the ESE kernel we map processes to tasks, asynchronous graph channels to asynchronous ESE channels or mailboxes, synchronous channels to procedure calls and input devices to interrupt handlers and signal ports.

5.6.6 Viability analysis

For any number of channels in an implementation we can accurately determine the scheduling overhead incurred by ESE, for a given hardware platform. We also measure the cost of processing each type of message that can flow in the system. Using this input we can determine whether an implementation of an RT/PC design is viable — in other words, whether the implementation as mapped from the RT/PC design can be scheduled on the given hardware platform in such a way that deadlines are guaranteed.

Since the ESE kernel employs a non-preemptive earliest deadline first scheduling policy, the

Stellenbosch University http://scholar.sun.ac.za

viability analysis is done as if a separate task exists for each channel. Because ESE does not support lockable resources, the two viability conditions, 1 and 2, are sufficient for viability. Condition 1 is computed using a spreadsheet, condition 2 by a special purpose program.

5.6.7 Implementation

If the viability analysis was done correctly, we are assured that the corresponding implementation is guaranteed to be temporally correct. Exact reasoning about the real-time behaviour of the system is possible as long as the implementation follows the viable set of tasks and channels exactly.

Stellenbosch University http://scholar.sun.ac.za

Chapter 6

Conclusion

The purpose of this study was to determine a design and implementation methodology suitable for small embedded reactive systems. We therefore surveyed the current trends in real-time systems development, and found a basic dichotomy: a time driven approach and an event driven approach [3, 32]. The time driven approach is popular in hard real-time systems which can be implemented as sets of periodic processes [65]. Rate monotonic analysis and scheduling are widely used in periodic systems [6], and pre-run-time schedulir is used to guarantee deadlines [65]. The event driven approach is popular in systems have to cope with unexpected events with hard deadlines. Such systems are typically implemented as sets of asynchronous processes, and scheduled by a deadline driven algorithm [49].

The trend in real-time operating systems is towards distributed, fault tolerant systems which employ pre-run-time scheduling, or run-time schedulability analysis to guarantee deadlines [11, 17, 38]. Hybrid techniques combine priority based scheduling with deadline driven scheduling to handle overload conditions, priority inversions and fault tolerance [9, 49, 50].

The systems in which we are interested are event driven, and have real-time requirements and strict resource constraints. We therefore require a design and implementation paradigm which supports sporadic processes and incurs the least possible overhead on the system. Jeffay's RT/PC design method [28] provides a design and analysis method which can guarantee deadlines for reactive systems. The ESE kernel, which I implemented, supports implementations of RT/PC systems, and guarantees their correct temporal behaviour.

Jeffay's RT/PC paradigm has been found to be a practical design method for the systems implemented on ESE. It is an intuitive model for reactive systems, and provides a formal basis for rigorous analysis and reasoning about time. RT/PC not only allows the designer

Stellenbosch University http://scholar.sun.ac.za

to guarantee deadlines in the design, but also to make accurate predictions about run-time performance. The automated viability analysis procedure, which is based on a formal model, ensures the temporal correctness of embedded systems.

The ESE kernel proves that it is possible to support RT/PC efficiently at run-time. The scheduling implementation of the ESE kernel is highly efficient, and the run-time overhead incurred by using the ESE kernel is minimal. Since deadline driven scheduling allows a theoretical processor utilisation factor of 100% [39], RT/PC systems implemented on the ESE kernel can use the processor optimally.

RT/PC systems implemented on the ESE kernel are suitable for embedded reactive systems. Because the RAM and processor overhead incurred by using ESE is minimal, it is ideal for systems with limited resources. The process abstraction provided by RT/PC can model the concurrency inherent in systems which interact with devices, and hard deadlines of devices can be guaranteed. Another significant advantage of using RT/PC is that the amount of buffering required in a system can be minimised. An RT/PC system can be designed in such a way, that the consumer of messages on a channel will always consume them faster than they are produced. This means that no buffering is required between the processes.

The ESE kernel is implemented in Modula-2, and all hardware dependencies are isolated in one module. The size of the hardware dependent module is 323 lines of source code for the embedded Intel 80188 version, and 218 lines of source for DOS. ESE requires 4737 bytes of RAM for code on an Intel 80x86 processor, and one hardware timer to operate. Provided a Modula-2 compiler is available, porting ESE to a new hardware platform is a quick, simple exercise.

ESE's non-preemptive sporadic processes make it suitable to embed in another operating system. Since ESE will only execute processes when they have messages to process, an ESE system embedded in another operating system will consume no processor cycles until an event causes it to schedule a process. Because ESE processes run to completion, there is a known upper bound on the amount of time for which an ESE system will remain active before returning control to the host operating system. Provided that the host operating system can guarantee a bounded delay between an event arriving for ESE and ESE being activated, hard deadlines can still be guaranteed. One example of using ESE embedded under another operating system is a version of ESE which can be used for DOS *terminate but stay resident* programs.

Jeffay [28] posed a number of questions which could only be answered after using RT/PC in

Stellenbosch University http://scholar.sun.ac.za

practice:

- *Can real-time systems be naturally and elegantly designed with RT/PC?* RT/PC is ideally suited to modeling event driven systems. Systems with periodic tasks can, however, also be modeled by sending a periodic timer message to a sporadic process.

- *What is the benefit of using RT/PC over more traditional methods?* The first benefit of using RT/PC is that deadlines are guaranteed. Since an ESE design is a closed system, there will be no unexpected run-time overload conditions which can cause deadlines to be missed. The deadline driven scheduling algorithm avoids priority inversions which may also cause missed deadlines.

  For small embedded systems a major benefit is the fact that RT/PC allows the designer to minimise the amount of buffering that is required. When communicating processes are not guaranteed to adhere to RT/PC, some buffering mechanism is required, which must provide for the worst case of unprocessed messages. This inter-process buffering is unnecessary for RT/PC processes.

  Jeffay proves that the RT/PC discipline is deadlock free [28]. An RT/PC system is driven by events, and a process is only scheduled when it has a message to consume. In ESE there are no lockable shared resources which may block a process, and since all inter-process communication is asynchronous, deadlocks cannot occur.

  The small size of the ESE kernel means that it can be used in environments where kernels with more sophisticated features would be too expensive to support. This means that a whole new class of embedded systems can benefit from process based design, and guaranteed deadlines.

- *It must be demonstrated that it is possible to implement the RT/PC paradigm.* The ESE kernel and the various implementations done on it demonstrate that the paradigm can indeed be implemented.

- *Is the cost of scheduling and supporting RT/PC too great?* ESE demonstrates that the cost of scheduling can be extremely low. When non-preemptive earliest deadline first scheduling is used, all processes can be executed on the same stack, and the selection of the next process can be done in time linearly proportional to the number of channels ($O(n)$ for a set of n channels). Section 4.11 gives measured performance figures.

- *Can task sets in practice satisfy the viability conditions?* The class of applications, for which the RT/PC method and the ESE kernel is being used, is always a closed, reactive

Stellenbosch University http://scholar.sun.ac.za

set of sporadic tasks. For these systems the viability conditions were not found to be restrictive.

RT/PC and the ESE kernel are currently in use by three commercial companies and a research institute for applications ranging from data communications to satellite subsystems. The design method has been found practical and of benefit in the development of small embedded systems. The correctness of systems has been improved by eliminating timing errors, and the reduction of buffering has been an important benefit. The absence of timing errors, and the fact that process scheduler priorities and buffering do not have to be tuned during system testing, has been found to reduce the development time of embedded systems.

Stellenbosch University http://scholar.sun.ac.za

Appendix A

ESE interface types and commands

The basic ESE kernel supports processes which communicate over asynchronous channels, and input devices which signal device driver processes on input ports. It also provides timers which signal expiry by sending a message to a process over an asynchronous channel. The following sections describe the parameters and semantics of each of the commands of the basic ESE kernel.

A.1 Types

The following types are used in the interface to ESE.

A.1.1 Alarm

An opaque type in the interface which identifies an alarm record in the timer table.

```
Alarm                = POINTER TO AlarmRecord ;
```

A.1.2 AsyncChannel

An opaque type in the interface which identifies an asynchronous channel record in the channel table.

```
AsyncChannel         = POINTER TO AsyncChannelRecord ;
```

Stellenbosch University http://scholar.sun.ac.za

A.1.3 InputPort

An opaque type in the interface which identifies an input port record in the channel table.

```
InputPort            = POINTER TO InputPortRecord ;
```

A.1.4 Mailbox

An opaque type in the interface which identifies a mailbox record in the channel table.

```
Mailbox              = POINTER TO MailboxRecord ;
```

A.1.5 Message

A pointer to a buffer which contains a message sent on a channel.

```
Message              = POINTER TO ARRAY [1..MAX_MESSAGE_SIZE] OF Octet ;
```

A.1.6 Name

Various objects in ESE receive symbolic names to identify them for debugging purposes. A name is an ASCII string.

```
Name                 = ARRAY [1..MAX_NAME_SIZE] OF CHAR ;
```

A.1.7 Octet

An Octet is an 8 bit type which can store numbers in the range $[0, 2^8 - 1]$.

A.1.8 Process

An opaque type in the interface which identifies a process record in the process table.

```
Process              = POINTER TO ProcessRecord ;
```

Stellenbosch University http://scholar.sun.ac.za

A.1.9 Thread

The entry point of a process, represented by a procedure type with no parameters.

A.1.10 Time

A 64 bit type is used to store time values. Time is measured in elapsed microseconds, so a time interval of $2^{64} - 1\mu s$ can be handled. This translates to 584942 years, which is sufficient to implement linear system time. A 32 bit time value represents an interval of just over 1 hour, if time is measured in μs. This would require the system time to wrap to 0 periodically, introducing possibly inconsistent deadlines.

A.1.11 Timer

An opaque type in the interface which identifies a timer record in the timer table.

```
Timer                    = POINTER TO TimerRecord ;
```

A.1.12 UserRef

A number, chosen by the user, by which ESE will identify channels and timers to the user in the *Receive* command.

A.2 CreateProcess

CreateProcess is used to create a sporadic process which will be scheduled if it receives messages on its input channels. *CreateAsyncChannel*, *CreateMailbox* and *CreateInputPort* are used to create the input channels to a process.

```
PROCEDURE                              CreateProcess
   (    name                              : Name ;
        thread                            : Thread ;
    VAR process                           : Process) ;
```

Stellenbosch University http://scholar.sun.ac.za

A.3 CreateAsyncChannel

CreateAsyncChannel is used to create a unidirectional asynchronous communication channel between only two processes. *CreateProcess* is used to create the processes.

```
PROCEDURE                          CreateAsyncChannel
    (      name                        : Name ;
           user_ref                    : UserRef ;
           period                      : Time ;
           sender                      : Process ;
           receiver                    : Process ;
           max_message_size            : CARDINAL ;
       VAR channel                     : AsyncChannel) ;
```

A.4 CreateMailbox

CreateMailbox is used to create an asynchronous channel which can buffer one or more messages. A mailbox has one receiver and one or more senders. *CreateProcess* is used to create the processes.

```
PROCEDURE                          CreateMailbox
    (      name                      : Name ;
           user_ref                  : UserRef ;
           receiver                  : Process ;
           no_of_slots               : CARDINAL ;
           slot_size                 : CARDINAL ;
           period                    : Time ;
       VAR mailbox                   : Mailbox) ;
```

A.5 CreateInputPort

CreateInputPort is used to create an input port on which an interrupt handler can signal to a device driver process. *CreateProcess* is used to create the processes.

Stellenbosch University http://scholar.sun.ac.za


```
PROCEDURE                              CreateInputPort
    (    name                             : Name ;
         user_ref                         : UserRef ;
         period                           : Time ;
         receiver                         : Process ;
     VAR port                             : InputPort) ;
```

A.6 StartSystem

StartSystem is called by the system to start the ESE scheduler once the process/channel network has been created. Once the *StartSystem* has been called ESE's scheduler remains in control and control never returns to the thread from which it was called.

```
PROCEDURE                              StartSystem
    () ;
```

A.7 SendMessageOnAsyncChannel

SendMessageOnAsyncChannel is used to send a message on an asynchronous channel.

```
PROCEDURE                              SendMessageOnAsyncChannel
    (channel                              : AsyncChannel ;
     message                              : Message ;
     size                                 : CARDINAL) ;
```

A.8 PutIntoMailbox

PutIntoMailbox is used to put a message into a mailbox. Mailboxes differ from asynchronous channels in that message k does not have to be processed before message $k+1$ is sent on the same channel. The mailbox buffers messages, and the receiver process of the mailbox is scheduled at a minimum rate equal to the period of the mailbox.

Stellenbosch University http://scholar.sun.ac.za


```
PROCEDURE                PutIntoMailbox
    (mailbox               : Mailbox ;
     message               : Message ;
     size                  : CARDINAL) ;
```

A.9 SignalInputPort

SignalInputPort is used by an interrupt handler to signal to a device driver process. Ports differ from asynchronous channels in that signal k does not have to be processed before signal $k+1$ is sent on the same channel. The port buffers signals, and the receiver process of the port is scheduled at a minimum rate equal to the period of the port.

```
PROCEDURE                SignalInputPort
    (port                  : InputPort) ;
```

A.10 Receive

Receive is called by a scheduled process to retrieve the user reference of the channel for which it was scheduled, as well as the message sent on that channel. If the channel is an input port only the channel user reference is returned.

```
PROCEDURE                Receive
    (VAR channel           : UserRef ;
     VAR message           : Message ;
     VAR size              : CARDINAL) ;
```

The message in the active channel is not copied out of the internal buffer by the *Receive* command. Since ESE employs non-preemptive scheduling, the sender of the channel cannot be scheduled during the execution of the receiver. The receiver is therefore guaranteed that the message will not be overwritten during its execution. The only exception occurs when a process sends messages to itself on a channel. In this case the process would have to copy the message to a local buffer before sending another.

Stellenbosch University http://scholar.sun.ac.za

A.11 SetTimer

SetTimer is used to start a kernel timer which will expire after the specified period, and send an expiry message to the receiver process of the specified notification channel.

```
PROCEDURE                          SetTimer
   (    user_ref                      : UserRef ;
        notification_channel          : AsyncChannel ;
        period                        : Time ;
    VAR timer                         : Timer) ;
```

A.12 SetAlarm

SetAlarm is used to start a kernel timer which will expire after the specified period, and put an expiry message into the specified notification mailbox.

```
PROCEDURE                          SetAlarm
   (    user_ref                      : UserRef ;
        notification_mailbox          : Mailbox ;
        period                        : Time ;
    VAR alarm                         : Alarm) ;
```

A.13 StopTimer

StopTimer is called to stop a timer which was started earlier by a *SetTimer* command, and whose expiry message has not been received yet.

```
PROCEDURE                          StopTimer
   (timer                             : Timer ;
    user_ref                          : UserRef ;
    notification_channel              : AsyncChannel) ;
```

Stellenbosch University http://scholar.sun.ac.za

A.14 StopAlarm

StopAlarm is called to stop an alarm which was started earlier by a *SetAlarm* command, and whose expiry message has not been received.

```
PROCEDURE                              StopAlarm
   (alarm                                 : Alarm ;
    user_ref                              : UserRef ;
    notification_mailbox                  : Mailbox) ;
```

It is possible for a timer or alarm to expire, and to be reused by another process, before the receiver process is notified of the first expiry. If the receiver process is scheduled in response to another event, which causes it to stop the timer (alarm), it will still have a valid timer (alarm) reference, but possibly for a timer (alarm) set by another process. Both the user reference and notification channel (mailbox) of the timer (alarm) are therefore compared to the parameters of the *StopTimer* (*StopAlarm*) call. If they do not match, the timer (alarm) referred to in the call had already expired.

If a process attempts to stop an expired timer (alarm), ESE removes the expiry message from the notification channel (mailbox), after checking that the user reference of the expiry message matches the parameter of the *StopTimer* (*StopAlarm*) call.

Stellenbosch University http://scholar.sun.ac.za

Bibliography

[1] iRMK^{TM} Version 1.1 Real-Time Kernel, and iRMX^{TM}286 Release 2.00 Operating System. *Intel Corporation, Santa Clara, CA*, June 1987.

[2] Revision 1.0 ANSI T1S1. Frame Relay Specification with Extentions. Technical report, DEC, Northern Telecom, Inc., Stratcom, Inc., 1990.

[3] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *IEEE Proceedings*, 79(9):16–21, September 1991.

[4] B.A. Blake and K. Schwan. Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System. *IEEE Transactions on Software Engineering*, 17(1):34–44, January 1991.

[5] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[6] L.R. Carter. An Introduction to Rate Monotonic Analysis. *Notes of IFIP WG2.4 African Autumn School, University of Pretoria*, 29 March to 2 April 1993.

[7] D.R. Cheriton et al. Thoth, a Portable Real-Time Operating System. *Communications of the ACM*, 22(2):105–115, February 1979.

[8] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[9] H. Chetto and M. Chetto. An Adaptive Scheduling Algorithm for Fault-Tolerant Real-Time Systems. *Software Engineering Journal*, 6(3):93–100, May 1991.

[10] E.M. Clarke et al. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. *Proceedings 10th Annual ACM Symposium on Principles Programming Languages*, 1983.

Stellenbosch University http://scholar.sun.ac.za

[11] A. Damm et al. The Real-Time Operating System of MARS. *ACM Operating Systems Reviews*, 23(3):141–157, July 1989.

[12] S. Davari and L. Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Reviews*, 26(2):110–120, April 1992.

[13] W-P. de Roever. Foundations of Computer Science: Leaving the Ivory Tower. *EATACS Bulletin*, 44, June 1991.

[14] S.K. Dhall and C.L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, January–February 1978.

[15] A. Fugetta et al. Some Consideration on Real-Time Behaviour of Concurrent Programs. *IEEE Transactions on Software Engineering*, 15(3):356–359, March 1989.

[16] B. Furht et al. Performance of REAL/IX^{TM} — Fully Preemptive Real Time UNIX. *ACM Operating Systems Reviews*, 23(4):45–52, October 1989.

[17] A. Geith and K. Schwan. CHAOS^{arc}: Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications. *ACM Transactions of Computer Systems*, 11(1):33–72, February 1993.

[18] V.H. Haase. Real-Time Behaviour of Programs. *IEEE Transactions on Software Engineering*, 12(9), September 1981.

[19] D. Haban and K.G. Shin. Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. *IEEE Transactions on Software Engineering*, 16(12):1374–1389, December 1990.

[20] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[21] D. Harel et al. STATECHARTS: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[22] W.S. Heath. Software Design for Real-Time Multiprocessor VMEbus Systems. *IEEE Micro*, 7(6):71–80, December 1987.

[23] C.R. Hehner. Real-Time Programming. *Information Processing Letters*, 30(1):51–57, January 1989.

Stellenbosch University http://scholar.sun.ac.za

[24] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[25] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1986.

[26] M.E.C. Hull et al. Development Methods for Real-Time Systems. *Computer Journal*, 34(2):164–172, 1991.

[27] F. Jahanian and A.K-L. Mok. A Graph-Theoretic Approach for Timing Analysis in Real-Time Logic. *Proceedings of Real-Time Systems Symposium (IEEE), New Orleans, LA*, pages 98–108, December 2–4, 1986.

[28] K. Jeffay. *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, 1989.

[29] M. Joseph and A. Goswami. Formal description of realtime systems: a review. *Information and software technology*, 31(2):67–76, March 1989.

[30] K.B. Kenny and K-J. Lin. Measuring and Analyzing Real-Time Performance. *IEEE Software*, 8(5):41–49, September 1991.

[31] F. Kolnick. *The QNX Operating System*. Basis Computer Systems Inc, Ontario, Canada, 1989.

[32] H. Kopetz. Event-Triggered versus Time-Triggered Real-Time Systems. *In Operating Systems of the 90's and Beyond, Lecture Notes in Computer Science 563, Springer-Verlag*, July 1991.

[33] H. Kopetz et al. Distributed fault-tolerant real-time system: The MARS approach. *IEEE Micro*, 9(1), February 1989.

[34] H. Kopetz et al. The Design of Real-Time Systems: From Specification to Implementation and Verification. *Software Engineering Journal*, 6(3):72–82, May 1991.

[35] R. Kurki-Suonio. Stepwise Design of Real-Time Systems. *IEEE Transactions on Software Engineering*, 19(1), September 1993.

[36] J.P. Lehoczky and L. Sha. Performance of Real-Time Bus Scheduling Algorithms. *ACM Performance Evaluation Review*, 14(1), May 1986.

Stellenbosch University http://scholar.sun.ac.za

[37] S. Levi and A.K. Agrawala. *Real Time System Design*. Computer Science Series. McGraw-Hill, 1990.

[38] S. Levi et al. The MARUTI Hard Real-Time Operating System. *ACM Operating Systems Reviews*, 23(3):90–105, July 1989.

[39] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[40] L.Y. Liu and R.K. Shyamasudar. Static analysis of real-time distributed systems. *IEEE Transactions on Software Engineering*, 16(4):373–388, April 1990.

[41] K. Marzullo and M. Wood. Making Real-Time Systems Reliable. *ACM Operating Systems Reviews*, 25(1):45–48, January 1991.

[42] F.W. Miller. The Performance of a Mixed Priority Real-Time Scheduling Algorithm. *ACM Operating Systems Reviews*, 24(4):52–63, October 1990.

[43] F.W. Miller. Predictive Deadline Multi-Processing. *ACM Operating Systems Reviews*, 24(4):52–63, October 1990.

[44] J.S. Ostroff. *Temporal logic for real time systcms*. Advanced software development series. Research Studies Press Ltd, 1989.

[45] A. Pnueli. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends — in Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*. Springer-Verlag, Editor — De Bakker, JW et al edition, 1986.

[46] J. Ready. VRTX: a real-time operating system for embedded microprocessor applications. *IEEE Micro*, pages 8–17, August 1986.

[47] K. Schwan et al. Real-Time Threads. *ACM Operating Systems Reviews*, 25(4):35–46, October 1991.

[48] K. Schwan et al. Multiprocessor Real-Time Threads. *ACM Operating Systems Reviews*, 26(1):54–65, January 1992.

[49] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[50] L. Sha and J.B. Goodenough. Real-Time Scheduling Theory and Ada. *IEEE Computer*, 23(4):53–62, April 1990.

Stellenbosch University http://scholar.sun.ac.za

[51] A.C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[52] A.C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[53] T. Shepard and J.A.M. Gagné. A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, 17(7):669–677, July 1991.

[54] K.G. Shin et al. A Distributed Real-Time Operating System. *IEEE Software*, 9(5):58–68, September 1992.

[55] J.A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, pages 62–72, May 1991.

[56] A.D. Stoyenko et al. Analysing Hard-Real-Time Programs for Guaranteed Schedulability. *IEEE Transactions on Software Engineering*, 17(8):737–749, August 1991.

[57] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall International, second edition, 1988.

[58] H. Tokuda and C.W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Reviews*, 23(3):29–53, July 1989.

[59] W.M. Turski. Time Considered Irrelevant for Real-Time Systems. *BIT*, 28(3):473–486, 1988.

[60] C. Warren. Rate Monotonic Scheduling. *IEEE Micro*, June 1991.

[61] Recommendation X.200. Reference model of open systems interconnection for ccitt applications. Blue book, volume viii — fascicle viii.4, CCITT, 1988.

[62] Recommendation X.25. Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for Terminals Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuit. Blue book, volume viii — fascicle viii.?, CCITT, 1988.

[63] J. Xu. Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.

[64] J. Xu and Parnas D.L. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, February 1990.

Stellenbosch University http://scholar.sun.ac.za

[65] J. Xu and D.L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.