1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**RUN-TIME KERNEL FOR
MULTI-ACTIVITY SYSTEMS USING
'PROCESS ACTIVITY LANGUAGE'
(PAL)**

by

Dan Sharon, B.Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Engineering.

Department of Systems and Computer Engineering
Carleton University,
Ottawa, Ontario, Canada

March 22, 1991

© copyright 1991, Dan Sharon

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-68900-5

Canada

The undersigned recommend to to the Faculty of Graduate
Studies and Research acceptance of the thesis

Run-Time Kernel For Multi-Activity Systems Using 'Process Activity Language' (PAL)

submitted by Dan Sharon, B.Eng
in partial fulfilment of the requirements for
the degree of Master of Engineering

Thesis Supervisor

Co-Supervisor

Chairman, Department of Systems
and Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
May 8, 1991

Abstract

Through a design and implementation, this thesis establishes the feasibility of a portable run-time kernel based on Process Activity Language (*PAL*), a formal design specification language that is based on the *Multiactivity* view of event-driven systems. To put the design in perspective, the multiactivity view of systems and its implied architecture is first introduced, along with a discussion of the *PAL* language. An overview of the run-time kernel is then presented followed by a description of the detailed design, including alternatives that were considered for parts of the kernel. Finally, the real-time performance of the system is evaluated, followed by a discussion of some application examples and considerations for performance improvement and future research and development.

Acknowledgements

I would like to thank my supervisor, Dr. M. Krieger (Department of Electrical Engineering, University of Ottawa), for his advice and encouragement over many months of discussion and consultation. Many sincere thanks to my family, for their continuing support and encouragement, and for their infinite patience to help see this project to completion. I gratefully acknowledge the financial support provided by the Ottawa-Carleton Research Institute. I would also like to thank the proof-readers for their time and consideration.

Contents

1	Introduction	1
1.1	Parallelism, Real-Time, and Multiprocessors	1
1.2	Thesis Objective	2
1.3	Summary of Achievements	3
1.4	Thesis Outline	4
2	Multiactivity Systems and Process Activity Language (PAL)	5
2.1	Multiactivity Systems	7
2.1.1	Multiactivity System Design	9
2.2	Centrally Coordinated Architecture	10
2.3	Overview of the PAL Language	13
3	Overview of the Run-Time Kernel	17
3.1	Environment Interface	26
3.1.1	Interaction with the Processes Manager	28
3.2	Processes Manager	29
3.2.1	Describing an Application Process	30
3.2.2	Global Billboard Area	32
3.2.3	Handling of Parallelism	32
3.2.4	Language Simplifications	32
3.2.5	Interaction with the Rest of the System	34
3.3	Activity Scheduler	35
3.3.1	Interaction with the Processes Manager	36
3.4	Message Handler	38
3.4.1	Resolution Of Requests	39
3.4.2	Interaction with the Processes Manager	39
4	Design and Implementation	42
4.1	Design Overview	45
4.2	Environment Interface	48
4.2.1	Event Handling	50
4.2.2	Handling of Queued Events	55
4.2.3	Environment Interface Services	56

4.2.3	Data Storage and Movement	. 60
4.3	Processes Manager	. 61
4.3.1	Graph Traversal	. 66
4.3.2	The Descriptor Table	. 71
4.3.3	Handling of the Language Elements	. 74
4.3.4	Processes Manager Services	. 78
4.4	Activity Scheduler	. 80
4.4.1	Handling of the Activity Requests	. 81
4.4.2	Activity Scheduler Services	. 87
4.5	Message Handler	. 87
4.5.1	Handling of the Messaging Requests	. 89
4.5.2	Message Handler Services	. 93
5	Performance Evaluation	94
5.1	Simulation of the Runtime Environment	. 94
5.2	Sample Applications	. 96
5.3	Measuring the Performance	. 99
5.4	Performance Results	103
5.5	Performance Byproducts	112
6	Multiactivity Applications	115
6.1	Dual Layer Systems	116
6.2	Hierarchical Multi-Layer Systems	117
7	Conclusions	120
7.1	Thesis Summary	120
7.2	Further Research	122
	References	124

List of Figures

Figure 2-1:	Event Driven System	6
Figure 2-2:	Multi-Level Decomposition	7
Figure 2-3:	Centrally Coordinated Architecture	10
Figure 2-4:	Construct Language Elements	14
Figure 2-5:	Sample Process of an Application	16
Figure 3-1:	The Kernel's Executive Processes	21
Figure 3-2:	Process States	23
Figure 3-3:	Structure of an I/O List	25
Figure 3-4:	ENV_IF Process Level View	28
Figure 3-5:	Describing an Application Process	31
Figure 3-6:	PROC_MAN Process Level View	34
Figure 3-7:	ACT_SCHED Process Level View	37
Figure 3-8:	MSG_HAND Process Level View	40
Figure 4-1:	System Code Modules	46
Figure 4-2:	Event Table	52
Figure 4-3:	Wait Table Data Structure	58
Figure 4-4:	Delay Table Data Structure	60
Figure 4-5:	Processes Table	63
Figure 4-6:	Completed Service Requests	64
Figure 4-7:	Nesting of Constructs	68
Figure 4-8:	Construct Language Elements	71
Figure 4-9:	Primitive Language Elements	73
Figure 4-10:	Activity Language Element	74
Figure 4-11:	the Ready-To-Run-Queue	82
Figure 4-12:	Activity Allocation Table	83
Figure 4-13:	Slave Table	84
Figure 4-14:	Channel Table	90
Figure 4-15:	Tx/Rx Request Queue	91

Figure 5-1a:	Serial Process 1		. 97
Figure 5-1b:	Serial Process 2		. 97
Figure 5-1c:	Parallel Process 1		. 97
Figure 5-1d:	Parallel Process 2		. 97
Figure 5-2:	4 Process Configuration		103
Figure 5-3a:	System Overhead, w/ Communications		105
Figure 5-3b:	System Overhead, no Communications		105
Figure 5-4a:	Event Response, w/ Communications		108
Figure 5-4b:	Event Response, no Communications		108
Figure 5-5a:	C.C. Bottleneck, w/ Communications		110
Figure 5-5b:	C.C. Bottleneck, no Communications		110
Figure 5-6a:	Slave Utilization, w/ Communication		113
Figure 5-6b:	Slave Utilization, no Communication		113
Figure 6-1:	Hierarchical Multi-Layer Systems		118

Chapter 1

Introduction

1.1 Parallelism, Real-Time, and Multiprocessors

With the advances in sophisticated Computer Aided Software Engineering (CASE) tools and code generators, the task of designing and implementing large software systems has become easier. The methodologies embodied in these tools allow designers to create software that is more structured and therefore more reliable, maintainable, and reusable. Some CASE tools incorporate graphical notations to aid in visualizing the system - an approach that is not new, but is widely accepted in both theory and practice [1,2,4,14].

This thesis is concerned with the branch of computer engineering known as event-driven, embedded and/or real-time systems, and specifically in the area of such systems that are implemented as multi-processors. Advances in this field are the results of general developments in many other areas of software engineering. The combination of Operating Systems theory and market-driven rapid evolution of powerful microprocessors has led to the development of many real-time executives [3, 9, 16] for the handling of multiple resources in concurrent environments. While an executive (or kernel) can provide increased productivity at the implementation stage, design tools in specific support of concurrent embedded systems are not very common but do exist [4, 14, 16]. Design tools that provide an estimate of the expected performance in the real-time multi-processor environment are even more rare.

Recent research at the Real-Time and Distributed Systems Group of the Ottawa Carleton Institute for Electrical Engineering has lead to the development of a set of design and analysis tools that are specifically address the issues of concurrency and the visualization of parallelism. The research of interest here revolves around the **Multiactivity** paradigm, a philosophy that promotes the separation of **coordination** from the actual **execution** of the activities comprising an application.

The philosophical premise and expected result of the multiactivity approach is that, given a proper set of development tools, a computer literate applications engineer can design and test a functional prototype with minimal programming effort, possibly bypassing the computer specialist at the design prototyping stage. This is possible because the functional specification is developed using a high level programming (executable) language called *Process Activity Language* (PAL) [11]. Research in the multiactivity domain, has so far been limited to the development of several PAL based off-line tools: a *PAL Verifier* [10] for checking the temporal correctness of a concurrent system specified as PAL expressions, an *Extended PAL Editor* [5] for the entry and editing of PAL expressions (including lexical and syntax checking), and a *PAL Simulator* [17] for functional and design prototyping.

To continue to promote the multiactivity paradigm as a viable design methodology, the next step was to implement a run-time kernel that would support PAL-based multiactivity applications, and to investigate its performance and limitations.

1.2 Thesis Objective

Rather than serving as a comparative study of alternate methodologies and operating systems, the primary objective of this thesis is to demonstrate that multiactivity can be efficiently supported at run-time by designing a kernel that directly interprets the language elements of PAL. Specifically, the kernel must be shown to be portable, small and efficient enough to allow PAL based systems to be implemented on a wide range of platforms, and for a wide range of applications. Since the requirements of many embedded applications are simple enough to be implemented on 8-bit microcontrollers, the kernel, and hence, multiactivity must be shown to be viable on these platforms.

To further confirm its viability, the run-time kernel must be measured for a variety of performance criteria, and must yield predictable and acceptable results when configured with the various parameters of a given application. By reviewing the performance of the kernel, it should be possible to establish the applications domain of multiactivity based systems, and the limitations imposed by their centrally coordinated architectures.

The implementation of the run-time kernel had several secondary objectives as well.

First, because PAL is still evolving, the design had to be easily extensible to allow new language elements to be integrated into the kernel. Second, the design had to be modular and cleanly structured to allow portions of the code to be reused in other PAL development (the simulator *SIMPAL*, was largely based on the algorithms and data structures of this kernel). Third, the performance self-test portion had to be efficient enough so as not to perturb the behaviour of the system.

1.3 Summary of Achievements

The major developments resulting from this thesis research include:

- the design and implementation of an efficient and small (49KB combined code and data) run-time kernel that directly supports multiactivity applications expressed in PAL. Written in C with no dependencies on any specific operating system, the kernel represents a highly portable implementation;
- testing of the kernel with a variety of configurations, showing that the kernel exhibits good performance characteristics, making it (and hence multiactivity systems) viable for a broad range of applications.

Each of these points is described below:

Design and Implementation

Elements of the PAL language allow the creation of multi-thread applications in which the concurrent nature of the system is described by a graph structure, which is 'traversed' by the kernel at run-time. By implementing data structures and algorithms that directly support the PAL language elements, it is shown by the performance of the system that such a kernel can be very efficient both in terms of code space and execution overhead.

The stand-alone C-based implementation makes the PAL run-time kernel highly portable and ROM'able. In embedded systems, therefore, it could form the first layer of software running on a bare hardware platform, with the PAL-based application representing the second (application) layer.

While concurrent languages exist and have been successfully used to implement concurrent systems [12, 15], several problems were seen with this approach. First, the languages are not supported on a wide variety of platforms, including the low-level microcontrollers being considered here. Second, they often rely on underlying executives to achieve concurrency, where the executives themselves are not portable between systems. Third, the concurrent entities embodied by these languages map onto a process or task level, whereas the PAL concurrent elements are intraprocess.

Viability by way of Performance

Several different applications were configured and submitted to run on this real-time kernel. By varying system parameters such as the number of slave processors, activity execution durations, and the frequency range of external events, it was possible to show that the kernel behaves in a predictable manner and yields run-time performance suitable for a wide variety of applications. Well suited to the prototyping-based approach of multiactivity system development, the performance data can also be used to establish configuration and minimum resource requirements for specific applications.

1.4 Thesis Outline

Chapter 2 serves as an introduction to multiactivity systems and to PAL. Issues addressed include multi-processor systems and inter-processor data passing, the implied architecture of multiactivity systems, and the language elements of PAL.

Chapters 3 and 4 describe the design of the run-time kernel developed to support the implementation of multiactivity systems. Chapter 3 gives an overview of the kernel and describes some design alternatives and critical issues that were considered. Chapter 4 provides a detailed explanation of the resulting design.

Chapter 5 discusses the real-time performance of the kernel and describes how the results were measured and interpreted. Chapter 6 describes some multiactivity-based application examples and shows how real-world systems can be mapped onto the multiactivity view. Finally, chapter 7 presents conclusions and suggests areas for further research.

Chapter 2

Multiactivity Systems and Process Activity Language (*PAL*)

Over the years, computer technology has evolved from small sequential batch machines, where single jobs were executed to completion one after the other, to larger more complex systems, where multiple users could execute multiple programs at the same time. Similarly, design methodologies have also evolved, matured, and become formalized and incorporated into modern CASE tools. Since many existing design methodologies are well suited to a range of applications, and produce quite acceptable results, the *Multiactivity* paradigm should offer substantial benefits to be worthy of consideration. What, then, is Multiactivity and when can it be used effectively?

The Multiactivity Approach

The primary premise of the multiactivity paradigm is that an application may be separated into distinct *execution* and *coordination* components. This separation of execution and control is applicable to a large range of event driven systems, where the term "event driven" is used to describe an application that can be viewed as being comprised of a set of responses to a set of discrete external events (Figure 2-1).

While "event driven" is usually synonymous with real-time systems, it is possible to have a system where the responses to events are clearly defined, but have no time constraints. An example of such a system is an interactive user interface, where responses must be generated for keystroke and mouse movement "events", but the time taken to respond to the user is not critical to the correctness of the operation.

Figure 2-1: Event Driven System

The one common element of many system design methodologies is a structured approach through functional decomposition [19]: large complex systems are broken down into levels of progressively smaller modules, with modules of one level combining to make up a higher level module. In multiactivity systems, the highest level building block of an application is called a process, where a process usually forms the entire response to an event, and hence holds a "response" view of the world. In certain applications, a response may be comprised of multiple cooperating processes triggered by the same event.

The Multitasking Approach

In these systems, the highest level building block of large applications is typically the task, where the overall application is viewed as a *Multitasking* system. A task may, in turn, have several levels of child tasks, and so on down to the lowest level building block called a function (using C language nomenclature).

In such multitasking systems, tasks can be of various sizes and complexities, performing a variety of jobs, and any task may communicate with and control one or more other tasks. In other words, the coordination may be left largely to the tasks themselves, with the operating system facilitating the intertask communications and interactions (as well as task scheduling and resource handling). Furthermore, in many applications, tasks take on a "resource" view of the world, in which a task "surrounds" a resource and offers services for interacting with that resource.

It is the aim of this thesis to show that the performance, small size, and portability of a run-time kernel for multiactivity systems makes this approach well suited to a variety of applications. These applications may range from low end microcontroller-based embedded systems, to high end multi-microprocessor industrial control and Flexible Manufacturing Systems (FMS).

2.1 Multiactivity Systems

At the first level of modular decomposition, a multiactivity system can be viewed as the set of responses that must be generated to properly satisfy the requirements of the environment that generates the events (Figure 2-1). A response can be further broken down into one or more *Processes*, which can, in turn, be partitioned into one or more *Sub-processes*, and so on down to the lowest modular level comprised of *Activities* (Figure 2-2).

Figure 2-2: Multi-Level Decomposition

A process, then, is ultimately defined as a set of activities that must be executed in a given order (precedence relation) so that a correct response to an event is produced (or partial response, if a response is comprised of multiple cooperating processes). The precedence relations of a process's activities are dictated by the data dependencies between the activities of the process, and possibly by dependencies on other processes.

A process may have multiple parallel threads of control, allowing multiple activities to be executed concurrently. There is no direct or implied mapping of threads to subprocesses, as subprocesses may themselves contain multiple concurrent threads.

The activity is the schedulable entity of multiactivity systems. In contrast to the task of multitasking systems, an activity is an "atomic" unit of execution, in that once started, it runs to completion without the need to wait for further information from other activities, or for availability of resources, etc.; all the scheduling considerations have already been taken care of by the precedence relations of the parent process. The granularity (functional size) of activities in an application is governed by such factors as communications overheads, resource requirements, and system reliability and performance requirements. The handling of prioritized access to shared resources is not addressed by the run-time kernel.

Since multiactivity promotes, and indeed requires, the separation of execution from the coordination aspects of systems, there are certain application areas that are better suited to and reap greater benefits from this approach than others:

- applications where a large portion of the system is concerned with coordination and control
- applications that contain complex control requirements that would benefit from the inherent handling of concurrency
- applications that require frequent changes in the coordination portion
- high reliability applications.

Reliability is inherently greater in a (multiactivity) system whose coordination is maintained by a computer other than the one executing the schedulable entities. In such a system, in the event of an executor (slave processor) failure, only the central coordinator needs to know about the failure. Since a failure of the central coordinator in such a system represents a single point catastrophic fault, in applications requiring very high reliability, the coordinator computer can be augmented with a variety of backup mechanisms (a shadow CPU running in hot standby, and/or a third arbitrator processor).

2.1.1 Multiactivity System Design

It is the separation of execution and coordination that allows multiactivity to speed the design cycle and enhance the reliability of systems. Conventional design of a system consists of a requirements phase and a detailed design phase. In the first, an applications engineer analyzes the requirements of a particular application and generates a high level solution called a functional (or requirement) specification, in which the desired behaviour of the system-to-be is described. In the second phase, a computer specialist "translates" the functional specification of the applications specialist into a design specification which can then be implemented on a particular target system.

In the design cycle for a multiactivity system, on the other hand, the (computer literate) applications specialist defines the requirement specification using Process Activity Language (PAL), a formal, structured, (and *executable*) high level programming language. Here, the requirement specification fully describes the necessary behaviour of the system and its structure through modular decomposition down to the activity level. This specification, however, is independent of any specific target platform, and as such is not constrained by any specific implementation.

The use of PAL to fully describe all the coordination aspects (and hence the behaviour) of the system-to-be, allows the requirement specification to be verified for any temporal errors that may result in deadlock or starvation [10].

In the development cycle of a multiactivity system, the design specification phase is much less distinct than the requirement specification stage: it simply consists of submitting the requirement specification to the PAL simulator [17], where the constraints imposed by an implementation on a specific target configuration may be applied. Through an iterative interactive procedure, the final design specification may be determined by trading off the various design parameters, such as: the required number and type of slave processors, activity execution durations, the activity to slave allocations, the required system I/O resources and their mapping to particular slaves, etc.

The main benefit of the multiactivity approach in the development cycle of a system, then, is the use of a consistent methodology from the conception phase to implementation phase. As well, the early verification of the requirement and design prototypes serves to increase reliability and reduce development time.

2.2 Centrally Coordinated Architecture

Computer systems span a wide range of architectures, from single processor machines, to multiple microprocessors sharing data in a common memory, to mainframe computers networked over an entire continent [8]

By virtue of the separation of coordination and execution, the most natural mapping of a multiactivity system is onto a multi-processor system, where a *Central Coordinator* computer contains and handles the coordination aspects of the application processes, and one or more slave processors serve as activity executors. With the centralized handling of all the precedence relations of all processes, the only entities executed by the set of slave processors are the activities defined for the system.

Figure 2-3: Centrally Coordinated Architecture

As shown in Figure 2-3, a typical high performance system will have the processors tightly coupled through shared memory. Where lower performance is tolerable and/or the processors are physically remote, the multiple processors may be loosely coupled by way of communications links. In applications with lower reliability and performance requirements, the execution of activities may be moved onto the central coordinator itself. Note also, that while multiactivity is inherently well suited to applications requiring large amounts of coordination, there is no reason why one or more of the slave processors cannot be a heavy computational element.

The centrally coordinated architecture may be a multi-layered one, in which an executor may itself be a coordinator of yet lower level executors, and so on. This is, in fact, the most typical implementation of a Flexible Manufacturing System (FMS): the overall FMS is controlled by a central coordinator, which dispatches activities to the various work cells (such as those used in flexible machining and assembly). To implement the overall work cell command, its controller will, in turn, dispatch activities to the work cell elements (such as a robot and/or numerically controlled machining device), and so on.

The primary drawbacks of a centrally coordinated architecture are: 1) potential control bottleneck, 2) potential data bottleneck, and 3) the single point failure of the coordinator computer. Control bottlenecks can be reduced by either increasing the granularity of activities and/or increasing the performance of the central coordinator computer. Data bottleneck can be alleviated by the proper definition of activities so as to reduce the communications requirement. If the communications requirement is still too great, the system may be reconfigured by adding data paths at the points of need (between the slaves that exchange the large amounts of data) and by implementing the data movement as the movement of pointers to the data in shared memory rather than the data itself. Note, however, that all control must still pass through one or more coordinator computers.

With control systems being the primary applications domain of multiactivity, these systems will be largely concerned with the acquisition of information from the environment via sensors, and with the control of the environment via actuators. Although the central coordinator computer could contain sensors and actuators, the job of handling them does not mesh well with the philosophy of a unit dedicated to system coordination. For this reason the sensors and actuators (as shown in Figure 2-3) will

typically be relegated to slave processors. System outputs, which are the culmination of responses to events, are generated by activities, which, by definition, execute on slaves. In systems where the number and frequency of input events generated by the slave processors becomes overwhelming to the central coordinator, a front-end processor may be added to reduce the overhead of data acquisition.

Slave Interaction Mechanism

While the fine details of the interaction between the central coordinator and its slaves is highly dependent on the coupling between the processors, the following list describes the common elements that must be present, independent the coupling method. To invoke the execution of an activity on a given slave processor (Figure 2-3), the central coordinator must at a minimum:

- transfer an activity identifier, which can be used by the slave to determine which activity to start executing. The activity identifier must be independent of the implementation of the activity code to allow the decoupling of changes in the slave software from those in the central coordinator.
- transfer the input data that the activity is to operate on, in the proper order and to the proper locations expected by the slave.
- signal the slave that all the activity startup conditions have been met, and that an activity may now be started (by causing an interrupt in the slave in a shared bus architecture such as the one depicted in the previous figure, or by sending a message in a loosely coupled arrangement).

In a common bus architecture, such as the one shown in Figure 2-3, each slave processor's shared memory would simply be mapped as a different physical area in the memory map of the central coordinator.

The corollary operations upon the completion of an activity are as follows:

- the slave signals the central coordinator of the activity completion (by causing an interrupt in a common bus architecture, or sending a message in a more loosely coupled topology).
- the central coordinator retrieves the output data generated by the activity from the memory map area designated as shared with that slave. Clearly, the output

data must be in the order expected by the central coordinator.

To expedite interprocessor handshaking in tightly coupled systems, some manufacturers of single board computers [7] have added support circuitry to automatically generate an interrupt when a particular memory location is written (in this case, when the activity identifier is deposited in the slave).

2.3 Overview of the PAL Language

Process Activity Language (PAL) is a structured executable high level language, having many common elements that are found in all programming languages, such as constructs for the expression of control flow. In multiactivity systems, the response to an external event consists of one or more processes. Using PAL, the internal structure of a process is specified by a set of expressions that describe the precedence relationships of the various sub-expressions and ultimately activities (executable elements) that comprise the process. The activities themselves are not specified using PAL, but rather by using conventional programming languages.

The language elements of PAL can be said to fall into three general categories: *Constructs*, *Primitives*, and *Activity*. Elements of the first category define the precedence relations for the execution of elements from the second and third categories, and allow conditional execution. Elements from the second category provide mechanisms for application processes to manipulate their private data, to exchange messages with other processes, and to synchronize with system events. The single element comprising the third category represents the schedulable unit in a multiactivity system, and is the mechanism by which application processes perform 'useful' work.

Constructs

As shown in Figure 2-4, the four primary Construct language elements are the *SEQ* (Sequence), *CON* (Concurrency), *CAS* (Case), and *RPT* (Repeat):

- the *Sequence* construct specifies that the elements contained within it are to be executed one after the other; each of them started only after the completion of the previous one. The Sequence is complete only when its last

Figure 2-4: Construct Language Elements

element completes execution.

- the **Concurrence** construct specifies that all the elements contained within it can be executed in parallel, meaning that all of its concurrent threads of control are to be started simultaneously. The Concurrence is complete only when all its elements complete execution.
- the **Case** construct is completely analogous to that of other programming languages, where only one of several possible branches is taken based on the value of the condition variable PC_i (PC for Process Condition). The Case construct is complete when the expression of the chosen path completes execution.
- the **Repeat** construct is analogous to a Repeat While statement that is found in standard programming languages, in that the expression contained by the RPT is executed as long as the Process Condition PC_i is true. The Repeat construct is complete when its contained expression completes execution and the process condition is false.

Three secondary control constructs (variations on the primary) are also defined in the PAL language: **SIM** (Simultaneous), **ALT** (Alternation), and **SEL** (Select). The first two

are variations on the Concurrency construct. In a SIM, start of execution of all threads is delayed until the first element of each thread is ready to execute. An ALT construct is considered complete when the first of its parallel threads completes. The SEL is similar to Ada's Select statement [2], where each path is examined for eligibility to execute, and of the eligible ones, one is chosen non-deterministically.

Primitives

PAL also provides eight Primitive language elements to facilitate inter-process communication and synchronization. Four of the primitives are always non-blocking, two are always blocking, and the remaining two are possibly blocking. The primitive language elements are grouped as follows:

- *TXM* (Transmit Message) and *WTM* (Wait for Message) are used for message based communications between processes. Message reception is blocking if no message is currently available on the specified channel, and message transmission is blocking if the buffer is full on the requested channel. In the definition of PAL, a message channel provides only one way communication between one single thread of a source process and one single thread of a target process.
- Processes may read or write to global billboard variables via the *RDS* (Read System Condition) and *STS* (Set System Condition) primitives, respectively.
- Synchronization to environmental events is provided by way of the *WTE* (Wait for Event) primitive, and access to environment data variables is available via the *RDE* (Read Environment variable) primitive. The Wait for Event primitive is always blocking.
- A general delay mechanism is provided by the blocking *DLY* (Delay) primitive, which also serves as a blocking NOP instruction when used with a delay parameter of zero.¹
- Operations on process data variables are provided by the *STP* (Set Process

¹ Interestingly, to avoid lockup in an endless loop in the kernel, a "dummy" blocking primitive is needed in a Repeat construct, if the expression bounded by the RPT contains no other blocking elements.

condition or variable) primitive, which allows process and global data variables to be combined in arithmetic and logical expressions with the result assigned to a target process data variable. The result of the operation may then be used for program flow control in a RPT or CAS construct.

Activities

An Activity is the basic building block of a process's expressions and constitutes the schedulable entity in PAL. An activity is a blocking language element. Once initiated, it executes to completion and requires no further interaction with the parent process or with other activities. In a multi-processor implementation, an activity executes on a slave computer, and for speed and reliability, may be executable by more than one slave. In a heterogeneous multi-processor, activities with special processing or resource needs are executable only by the particular slaves that fulfill those requirements.

In systems with large granularity activities (long execution times), and resources that are shared by the slave processors, an activity suspend mechanism may be needed to help boost system performance by giving higher priority activities privilege over the lower priority ones.

In conclusion, a multiactivity system is one comprised of the processes of an application, where the PAL language is used to define any single process. A process, in turn, is a set of activities and primitives whose order of execution is governed by the

Figure 2-5: Sample Process of an Application

precedence relations describing that process. A sample process is shown in Figure 2-5.

Chapter 3

Overview of the Run-Time Kernel

In order to show the viability of a multiactivity based system, the primary design objective was to implement a **highly portable** run-time kernel that could be the **only** software layer to run on top of a hardware platform i.e. the kernel was to be ROM'able and have no dependencies on any underlying operating system (OS) services. Portability was achieved by writing the kernel in a conventional language that is widely supported and provides good low level access to hardware (C was used), and for ROMability it was created as a single executable image. These portability criteria completely decouple the kernel from any particular CPU and its instruction set, and from any run-time executives that are available for embedded systems (be they for systems based on a single chip microcontroller or powerful multi-board computers).

Since multiactivity systems are control oriented by definition, the most important requirement of an OS for such an embedded machine is, necessarily, the process management portion, which the kernel implements. While the memory management component of an OS is a secondary consideration because of its effect on a run-time system, the third portion of an OS - the file system - clearly did not need to be addressed: since the central coordinator in a multiactivity system acts as an embedded controller, a file system is not needed in that computer. If file operations are needed elsewhere in the system, one of the slave processors can provide that functionality.

Heap memory management (not to be confused with a virtual paging system) is a non-trivial part of an operating system that supports many higher layers of system-level software. The lack of memory management in the kernel requires that all system data structures be allocated statically at compile time, and hence that all contingency handling of overload conditions also be addressed at compile time. The implications of an entirely static design are addressed below.

Dynamic Process Creation

Dynamic process creation allows multiple instances of a process to be invoked as responses to events, when events are arriving at a rate greater than the response's execution time. This serves to improve system response time during a temporary overload condition by indirectly allocating more processor time to the response(s) serving that condition. The lack of run-time memory management capability, however, requires that the memory needed to define all the processes of an application and their respective data areas must be completely defined at "application compile time".¹

Although the implemented kernel does not support multiple instances of processes, it does provide queuing of events as a means of handling temporary overloads. The difference between dynamic and static process creation, then, is that in the static model, the worst case system overload must be known or estimated, and provisions made at compile time, rather than at run time.

Overload Handling

In any system, with or without dynamic memory management, there is a point at which temporary run-time demands exceed the system's resources or performance capabilities. In most application areas, it is important for the system to provide some means of handling the overload condition in such a manner that it continues to run and respond to its environment, but at a reduced performance level i.e. the system must degrade gracefully. In a system with dynamic memory management, the resource limit is reached when all the system's memory has been allocated. In a system with no such memory management, the resource limit is reached when the statically allocated memory areas become filled. Clearly, either type of system must be designed such that on the average, the available performance and resources can successfully service the demands of the application.

¹ for lack of a better term, the word compile is used here to denote the merging of an application's configuration information with that of the kernel itself into a single executable image.

In multiactivity systems, the PAL language itself provides a partial mechanism for the specification of a system's overload handling by allowing events to be classified as one of Volatile, Latched, or Queued. If a volatile event's response (process) is not available for execution at the time of the event, then the event is simply ignored or lost. When an event is specified as queued, if its response is not available for execution at the time of the event, then the event occurrence is remembered for up to N occurrences (where N is the event's specifiable queue length). A latched event is an event with a queue length of one.

Event overload handling is supported in the run-time kernel, as is the handling of interprocess communications overload: when a communications channel becomes full, a send message primitive (TXM) becomes blocking, even though in the language definition it is non-blocking.

Issues in Concurrency

Easily the most difficult logistical aspect of concurrent systems, is that of the synchronization of control and data among the concurrent entities, be they parallel threads of a single application process, or multiple cooperating processes (or tasks in a conventional system). In multiactivity systems, intraprocess concurrency and interprocess concurrency are distinct areas and are addressed in the following discussion.

Intraprocess concurrency: within any single process, the problem is in the handling of queued requests, such as those arising from the blocking language elements. Here, an entry in a queue may have an associated set of data, but that data is indirectly referenced by the entry, rather than being deposited with the entry itself (the latter being very costly for large sets of data). When the queued entry finally comes up for service, the data that it references may no longer represent the state of the data as it was at the time that the (queued) request was issued; other concurrent entities may have operated on that data in the meantime.

Since parallel threads of a process must be completely independent of each other for true concurrent execution, there can be no sharing of data among those threads. If a blocking language element (ACT for example) in one thread references the same data

variable as another blocking language element in another parallel thread, then this condition is an error (and is flagged by the Extended PAL Editor tool).

Interprocess concurrency: when an application in a multiactivity system is made up of multiple processes that communicate with each other, it must be ensured that for a given channel, every send message primitive has a corresponding receive. Both the send and receive primitive must be testable for deterministic occurrence, or else the system may be susceptible to deadlock. This function is performed by the PAL Verifier tool.

Several other potential problem areas arise in an application configured with multiple communicating processes: - how should messages destined for an inactive process be handled? - how should the case of multiple instances of one process communicating with a single instance of another be handled? Clearly, these are error conditions, and are, in fact, flagged by the Extended PAL Editor tool, which imposes the following conditions on processes that cooperate by exchanging messages:

- only processes which together form a response to an event, may communicate by messaging i.e. cooperating processes must be started by the same event
- when an additional instance of one process of a cooperating set of processes is started, additional instances of all other cooperating processes in the set must also be started
- if messages are sent or received on parallel threads of any single process, then those messages must be sent on separate channels.

Structure of the Kernel

The kernel can be viewed as a collection of four *Executive Processes* interacting with each other by way of non-blocking service requests (Figure 3-1). The four executive processes cooperate in the execution of the *Application Processes*. The logical partitioning of the kernel's single executable image into four executive processes allows the individual processes to implement distinct and logically separable system-level functions. The partition is also based on a possible multiprocessor implementation of the kernel itself, where each of the executive processes would run on its own processor.

1. WTE, DLY, Next_Done_WTE, Next_Done_DLY, RDE
2. Signal_Proc, Data_To_Proc
3. ACT, Next_Done_Act
4. Signal_Proc, Data_To_Proc, Data_From_Proc
5. TXM, WTM, Next_Done_TXM, Next_Done_WTM
6. Signal_Proc, Data_To_Proc, Data_From_Proc

Figure 3-1: The Kernel's Executive Processes

Each executive process is said to have multiple entry points: one for each service request that it handles, one for each interrupt that it services, and a foreground portion that is scheduled by the outermost control loop of the kernel. A service entry point handles the reception of a request from another executive process for a particular service. The request is usually deposited in an internal queue or data structure of the receiver, whose foreground portion operates on the request at some later time. Since the kernel is a single executable image, however, all service request interactions between the executive processes are implemented as function calls.

For convenience of design and for the smallest overhead, the scheduling of the foreground portions of the executive processes was implemented as a free-running non-preemptive round-robin: each executive process (foreground portion) runs to completion before the next can start. The execution of each executive process is qualified by the need for its services i.e. if there are no application processes currently active, then there is no need to invoke the Processes Manager, or if there are no

messages currently queued, then there is no need to invoke the Message Handler, etc. Viewed at a high level, the executive processes can be described as follows:

Environment Interface

- services events when they occur and collects their data (if there is any)
- initiates event responses by signalling the Processes Manager to start one or more processes
- queues event occurrences and their data (up to some finite limit), when the corresponding response process(es) are not ready for execution
- provides services for the handling of the *Wait_For_Event* (WTE), *Read_Environment_Variable* (RDE) and *Delay* (DLY) language primitives

Processes Manager

- "executes" all application processes by interpreting their precedence relations
- maintains the current state of application processes as (application-level) service requests are issued and completions are signalled
- provides services for the movement of data into and out of application processes
- maintains a global billboard area by the handling of all *Read_System_Variable* (RDS) and *Set_System_Variable* (STS) language primitives

Activity Scheduler

- provides the interface between the central coordinator computer and the surrounding slaves by dispatching activities for execution
- maintains the relationship of slave processor ID, activity ID and the parent application process
- provides services for the handling of the activity (ACT) language primitive

Message Handler

- provides services for the *Send_Message* (TXM) and *Wait_for_Message* (WTM) language primitives by handling all message communications among the application processes, including queueing of received and transmitted messages.

Application Process States

Application processes can be viewed as having distinct states, with startup and resume transitions controlled by external events (Figure 3-2). The states and state transitions can be described as follows:

- All event triggered processes are initialized to the *Inactive* state. When an event occurs, the process(es) comprising the response to that event become *Ready* to run. Processes that run as infinite loops are initialized to the *Ready* state as part of the system power-on initialization.
- When a blocking language element (ACT, WTM, DLY, WTE, etc.) is encountered in a given thread of an application process, that thread becomes blocked, and a service request for the blocking element is issued. The overall process continues execution until all its threads eventually become blocked or completed, at which point the process becomes *Blocked*.
- When service of a blocking language element is completed, a *Blocked* process becomes *Active* when any one of its threads is unblocked, even though other threads of the same process (if any) may be currently blocked.

Figure 3-2: Process States

- an *Active* process which completes execution becomes *Inactive*, as does a process which is *Ready* and completes execution (i.e. a process with no blocking language elements executes to completion once started).

The main difference, then, between single-thread tasks (some OSs allow multi-thread tasks), and multi-thread processes is that a process becomes blocked only when all its threads are blocked. Activities are considered to be atomic executable elements, and hence have only two states: *Active* and *Ready*.

Data and its Movement

To do any useful work, a system must typically operate on input data to produce some output data. In event driven systems, of which PAL is a subset, data that is associated with events is moved into the system where it is processed, and the response of the system is then typically output by one of the slave processors.

Within the PAL run-time kernel, data must also be moved among the executive processes:

- data that is generated by an event must be moved into the appropriate data area (variables) of the Environment Interface.
- data maintained by the Environment Interface may be needed by an application process when it starts or at some other point in its execution by way of a *Read_Environment_Variable* (RDE) language primitive.
- if an activity running on a slave needs to operate on data that belongs to the parent application process, that data must be moved from the process's private data area in the central coordinator computer to the slave processor before the activity can be invoked (similarly, output data generated by an activity that is needed by the parent process).
- data that comprises the messages exchanged by application processes must be moved to/from the private data areas of one application process to another.

As in any program or multitasking OS, each application process of a PAL based system has a private data area containing an assortment of data variables that must be maintained. In this implementation of the run-time kernel, a PAL application (the set of all application processes) is not compiled and linked into machine instructions directly

executable by a CPU. Rather, the application is described by sets of tables and data structures that are interpreted at run-time by the kernel's executive processes. For this reason, the location of application process data variables are not directly known at run-time, and data movement must, therefore, incur one level of indirection.

This indirect movement of data is controlled by "IO_Lists" - data structures that contain pointers to the variables themselves (actually indices into a data area). The IO_Lists are grouped together into an IO_List_Area for each application process (Figure 3-3). The data variables themselves reside in a separate Data_Area. Since an IO_List may be referenced by data movement operations at several points in an application process and contains only variable identifiers, the direction of data movement is determined by the context in which the IO_List is used.

Although adding the overhead of one level of indirection to the movement of data, IO_Lists allowed a structured approach to be used in the handling of the PAL elements: all data referenced by a language element is via a single, fixed length field in the record that describes that element. Language elements that involve the movement of data include activities and messaging primitives.

IO_LIST_AREA:

Figure 3-3: Structure of an I/O List

In this prototype implementation of the run-time kernel, data variables can be one of the four basic types supported by the C language: *char* (1 byte), *int* (2 bytes), *long* (4 byte long integer), and *float* (4 bytes floating point). Every application process maintains a private *Data_Area*, which is simply the collection of all the variables declared at compile time.

In the discussion of the subsections that follow, the graphical notation first introduced by Buhr [2] is used to depict the system level interaction of the executive processes. The need to use that notation brings to light a shortcoming of the PAL notation: while clearly illustrating the inner control structure of an application process, Process Activity Language is unable to directly show the interaction between processes.

3.1 Environment Interface

A system's real world interface is comprised of two elements: sensors for reading environment information into the system, and actuators for transferring system information back out to the environment. In a general system, sensors and actuators may be connected to any of the processors in the configuration, but in keeping with the philosophy of PAL, the central coordinator merely controls the operation of all other subsystems, leaving the real world I/O interfacing to be performed by slave units.

Since PAL is designed for use in event-driven systems, all environment data is also acquired in an event-driven manner. Because the sensors in the system are attached to slaves only, the data from a particular sensor needs to be transferred to the Environment Interface in the central coordinator. This is accomplished by the slave causing or passing an event in the Central Coordinator, whose event service will then receive the data sensed by the slave. This means that the PAL system requires all slaves (that have sensors) to acquire environment data in an event driven manner. This is a reasonable requirement, since the slaves are expected to be busy executing activities, thereby relegating the job of interfacing with the environment to the background (driven by a local slave timer if the interface hardware requires polling).

Although typically there is data associated with the occurrence of an event, that is not necessarily so. It is possible to have purely signalling events, as is it also possible to

require the acquisition of data without a related event. In this latter case, the central coordinator computer would have to be configured with sensors.

When large amounts of data need to be continually acquired from the environment, the overhead of event servicing or sensor polling may become unacceptably high, in which case a dedicated front end acquisition slave could be added to relieve the load. Such a front end processor would likely be tightly coupled to the central coordinator and deposit the acquired data in a shared memory arrangement. Clearly, the central coordinator would have to specify the locations of its Environment Variables to match the locations in which their data is deposited by the front end slave (or move them from the shared area).

The Environment Interface is the PAL executive process responsible for the maintenance of real world information in the central coordinator, and for initiating responses to external events. Since in general the response of the PAL system to an external event is the invocation of one or more processes to deal with that event, it is the Environment Interface that maintains the correspondence between an event and its response. Although a system's interface to its environment must necessarily include inputs (events) and outputs (responses), the PAL kernel is structured in such a way that the Environment Interface only handles the inputs; system outputs are handled by the activity executors (slave processors). In high reliability/performance applications, it may be desirable to configure the system with actuators that are controllable by multiple activity executors: either in a primary/backup configuration, or one designed to increase the slave availability. In either case, the most likely point of failure will still be in the mechanical parts of the actuator.

The overall flow of data through the system is as follows:

- an environmental input is read by a slave
- the slave causes an event and passes the data to the Environment Interface
- the Environment Interface triggers the start of a response in the Processes Manager
- the activities of the response (process) are submitted to the Activity Scheduler
- the activities are invoked for execution on a slave processor
- part of the activity's function is to generate an output via an actuator.

3.1.1 Interaction with the Processes Manager

In addition to interfacing with the real world, the Environment Interface also provides services to the Processes Manager by handling the blocking language primitives *Wait_for_Event (WTE)* and *Delay (DLY)*. Although the delegation of *DLY* handling to the Environment Interface was somewhat arbitrary (since any executive process could have kept track of time), it seems reasonable to view system timekeeping as an interaction with the environment, and therefore allow the Environment Interface to handle it. Since no provision is made for the keeping of absolute date and time in this initial implementation, the handling of time is reduced to the generation of the delays requested by the application processes in units of ticks (where a tick is simply the fraction of a second supported by the hardware platform).

Figure 3-4: ENV_IF Process Level View

As shown in Figure 3-4, the Environment Interface interacts with the Processes Manager through a set of six service requests, and maintains five private data structures. Two of the service requests (*Signal_Process* and *Data_To_Process*) are initiated by the Environment Interface to indicate a required change of state for an application process, and to indicate a request for data movement from the Environment Interface to the *Data_Area* of an application process. The remaining four service requests are initiated by the Processes Manager for the handling of the *WTE* and *DLY* language elements, and to retrieve information on completed requests that have been signalled via *Signal_Process* as fulfilled (*Next_Done_WTE*, and *Next_Done_DLY*).

Events and waiting processes are handled with the three private data structures *Event_Table*, *Wait_Table* and *Delay_Table*. Two additional data structures are used in the movement and storage of Environment Interface data: each *IO_List* in the *IO_List_Area* is used to control the movement and assignment of data associated with an event to the particular Environment Interface variables in the *Data_Area* that are to receive that data.

A detailed discussion of the data structures maintained by the Environment Interface, and the algorithms that operate on them is found in the next chapter.

3.2 Processes Manager

In a system driven by Process Activity Language (PAL), the Processes Manager is the executive process responsible for the execution of all application processes as they are described by their PAL expressions. The expressions defining the precedence relations of a process are translated into a data structure known as a Descriptor Table at system "compile" time. The algorithms of the Processes Manager interact with that data structure at run time to interpret the *Constructs* and *Primitives* that comprise the process's expressions, and to invoke the *Activities* (executable elements) that perform the work of that process. In essence, the Processes Manager is a graph traversal program.

The Processes Manager can be viewed as the central PAL executive process, in that all other executive processes are treated as resource handlers by the Processes Manager,

since they interact only with it and not with each other. The Processes Manager maintains the current state of every application process, as service requests are issued to and completions arrive from the other cooperating executive processes. Data movement into and out of the *Data_Area* of application processes (as needed by handling of messages, activities, and events) is controlled by the *Data_To_Proc* and *Data_From_Proc* services provided to the other executive processes by the Processes Manager. All the blocking language elements of PAL are submitted by the Processes Manager to the other cooperating executive processes.

In a most general system implementation, an application process would typically have an attribute indicating how many copies of itself are allowable in the system at any one time. To keep track of multiple copies of the same running process, therefore, at least the data areas pertaining to the process would have to be replicated. However, because a process's control structure contains dynamic data fields, the implemented design of this PAL run-time kernel will also require the replication of the control structure (Descriptor Table) itself, thereby requiring the effective duplication of the entire process.

Recall that because of the lack of underlying memory management, and hence lack of dynamic process creation, the implemented run-time kernel handles system overload conditions by queueing the occurrences and the data of events whose responses were not ready for execution at the time of the event.

3.2.1 Describing an Application Process

As viewed in PAL, an application process (one of many comprising the system) is described as a set of language elements whose order of execution is governed by the control constructs relating them in a PAL expression. For the purpose of discussion, consider the application process comprised only of activities and their relationships, as shown in Figure 3-5.

This is a simple process comprising of nested sequence (*SEQ*) and concurrency (*CON*) constructs, where the concurrency elements are represented by the forks in execution paths. The run-time kernel uses a Descriptor Table which is similar to the pseudocode representation shown below the graph.

Figure 3-5: Describing an Application Process

In a more realistic PAL application, a process would likely also contain repeat (*RPT*) and case (*CAS*) control constructs, as well as some of the other executable language elements called Primitives (see Chapter 2). Therefore, in addition to controlling the execution of a process by interpreting its Descriptor Table at run-time, the kernel must also deal with a process's data and its movement. To support the handling of the process variables, a Data_Area is maintained. To support inter-process messaging and activity input and output, an IO_List_Area is maintained, which describes the grouping of the process's data variables for the I/O operations. The IO_List handling used in the movement of data in an application process is identical to the handling of the IO_Lists in the Environment Interface.

3.2.2 Global Billboard Area

In addition to the tables describing each process individually, the Processes Manager maintains a data area used for the exchange of billboard (or global) information that is common to all application processes in the system. This global area is accessible to a process via the 'Set System Condition' (*STS*) and 'Read System Condition' (*RDS*) primitives. The global data area is called *Sys_Data*, and is simply the collection of all the global variables (conditions) that were declared by all application processes in their 'STS' and 'RDS' primitives.

3.2.3 Handling of Parallelism

A multi-microprocessor system is an inherently parallel machine. In a PAL driven system, the overall application is comprised of multiple application processes, which may in turn, be comprised of multiple parallel threads of execution. The handling of the concurrent threads is controlled by the PAL language expressions as maintained by the Processes Manager in the central coordinator computer.

Since any single thread of an application process can become blocked when a blocking language element is encountered, an efficient method had to be devised to determine which of the threads of a process (if any) are available for execution, and at what point in the thread, execution needs to be resumed. Of the several thread tracing methods that were considered, the algorithm with the lowest overhead was found to be one in which a point of blockage (Descriptor Table index) is submitted along with the blocking request. When the service request has been completed and signalled, the point of blockage is retrieved by the Processes Manager from the resource server (via a *Next_Done_xxx*) and the thread tracing can simply continue from that point of blockage.

3.2.4 Language Simplifications

In this initial development of a PAL run-time kernel, all Primitives language elements have been included, but implementation was restricted to the handling of the four basic Construct element types: Sequence (*SEQ*), Concurrent (*CON*), Repeat (*RPT*), and Case (*CAS*). The Select (*SEL*), Simultaneous (*SIM*), and Alternation (*ALT*) Constructs, are

considered to be variations on the *CON* Construct, and were omitted in order to reduce the scope of the effort required to create an achievable system.

The handling of the Subprocess (*SUB*) concept was studied, but found to be difficult to implement at the run-time. Clearly, a top-down structured design methodology for the decomposition of large application processes into more manageable modules is highly desirable for the applications developer, and at the specification level, there is certainly no reason to restrict the usage of the Subprocess. The Extended PAL Editor [5], treats a Subprocess as an 'include' type function, which is pulled in-line to create the overall process Descriptor Table.

The reason for breaking a process down into more modular Subprocesses at run-time, would be to allow 'code' (Descriptor Table) sharing for PAL expressions occurring in more than one process or in several places in the same process. The implemented design, however, uses data structures with dynamic fields as the language elements, resulting in inherently unsharable 'code'. Specifically, the *SEQ* and *CON* elements require a mechanism for indicating their current state of completeness, which cannot be shared. Although some form of a context switch mechanism could have been used to swap the state of the *SEQ* and *CON* elements in a shared subprocess (to that of the currently executing parent process), it was felt that such an implementation would not contribute significantly to the body of research.

The other difficulty in the sharing of subprocesses is that there is no means of dynamically controlling the source and destinations of data movements: data references made in the language elements, be they directly to a process variable or indirectly via an *IO_List*, are hardcoded into the particular element.

A partial solution to making a Subprocess shareable given the implemented design, would be to ensure that *IO_Lists* and variables referenced by the language elements of the subprocess reside at the same locations in the *IO_List_Area* and *Data_Area* of all processes that share the '*SUB*'. In this case only the Descriptor Tables of the processes would have to contain replicas of the subprocess structure, however, this kind of fine tuning over variable locations would only be achievable with some form of linker. This solution would also likely result in larger *IO_List_Areas* and *Data_Areas* than are strictly necessary for the referencing of process variables, since the data locations would also be dictated by the 'sharing' requirement.

3.2.5 Interaction with the Rest of the System

As discussed earlier, the Processes Manager uses the other executive processes as servers for the handling of requests resulting from the blocking language elements. When a request has been fulfilled by a cooperating server (executive process), the completion must be signalled to the Processes Manager so that the execution state of the application process that had issued that request can be updated.

Further, as events occur, activities are started and completed, and messages are exchanged, data must be moved into or out of the private Data_Areas of application processes. Since the Processes Manager has exclusive control over this private data of application processes, it has to provide data movement service to the other executive processes. As well as issuing service requests to the cooperating executive processes, the Processes Manager, then, also provides three services itself (Figure 3-6):

- *DATA_TO_PROC*, for the transfer of data into the private area of an application process
- *DATA_FROM_PROC*, for the transfer of data from the private area of an application process
- *SIGNAL_PROC*, for the signalling of a resource completion.

The Processes Table data structure is private to the Processes Manager and comprises separate sub-structures for each application process configured into the application. A detailed discussion of the data structures maintained by the Processes Manager, and the algorithms that operate on them is given in the next chapter.

Figure 3-6: PROC_MAN Process Level View

3.3 Activity Scheduler

In a multiactivity system, the Activity Scheduler is the executive process that acts as the interface between the central coordinator and the surrounding activity executors (slave processors in the case of a multiprocessor system). Since it is not necessary that all slave processors be identical (indeed are likely to be specialized for a particular application), the Activity Scheduler must know which activities are executable by which slaves.

The responsibilities of the Activity Scheduler, then, can be described as follows:

- to receive activity execution requests from the Processes Manager
- to maintain the request (Ready-To-Run) queue according to the scheduling algorithm
- to buffer activity completions and report them to the Processes Manager
- to assign activities to the appropriate executors (slave processors) based on the slave's availability and capability to execute the activities
- to move an activity's input data from its parent process's private data area into the slave processor, and to move the activity's output data back into the parent process's data area
- to perform the low level handshaking with the slave processors to ensure proper data and control transfer.

In general, job scheduling in a multiactivity system can be said to be either activity based or response based. The former uses parameters associated with the activities themselves to determine which one to activate, while the latter uses parameters associated with the overall response to make that decision. A versatile (and complex) Activity Scheduler could implement a variety of job scheduling algorithms based on activity parameters such as Arrival Time, Static Priority, and Execution Length, or response parameters such as Due Date and Slack Time, and allow dynamic switching between them for run-time tuning of system performance. However, since the run-time kernel was designed to get a first order estimate of its feasibility and performance, it was deemed sufficient to implement a simple First-Come-First-Served activity scheduling algorithm to prove the point.

Slave I/O Data Movement

As was shown in Chapter 2, the centralized control required in a multiactivity system is best served by a vertical organization, with an assigned Master and one or more slaves connected in a single (control) bus topology (as opposed to other interconnections such as Star, Ring, and Fully Connected). It is when an application requires the movement of large amounts data into and out of the coordinator, that the centralized architecture may produce an unacceptable bottleneck.

Without changing the connection topology, the bottleneck can be reduced through several mechanisms: 1) increasing the granularity of activities into units that do more work, thereby reducing the control overhead, 2) decreasing inter-activity data dependencies, thereby reducing the amount of data to be moved into and out of the central coordinator, and 3) manipulating pointers to large data blocks, rather than the data itself, in applications that are designed for operating on large amounts of data.

Clearly, to handle the latter case, the Activity Scheduler would have to implement a job scheduling scheme that takes into account the dependencies of activities on the locations of the large data blocks. This was not implemented in this prototype run-time kernel.

3.3.1 Interaction with the Processes Manager

The *Activity* element of the PAL language is always blocking; that is, the current thread of execution of the application process becomes blocked as the request for the execution of the activity is submitted to the Activity Scheduler by the Processes Manager. The primary service that the Activity Scheduler provides, then, is the reception of activity execution requests. As shown in Figure 3-7, the other service provided by this executive process is the retrieval of activity completion information (*Next_Done_ACT*).

The Activity Scheduler itself, also issues 3 requests for service to the Processes Manager, and maintains 3 private data structures. The service requests are for the movement of data to/from an activity's parent application process, and for the signalling to that process of the completion of an activity. The data structures maintained by the Activity Scheduler are the Ready-To-Run-Queue, the Slave Processor table, and the Activity Allocation table.

Figure 3-7: ACT_SCHED Process Level View

Figure 3-7 shows that the signalling of a slave completion is by way of an interrupt. This background interrupt function serves to update the Slave Table with the completion of the particular activity, move the activity's output data to the parent process's private Data Area, and service the Ready-To-Run-Queue based on the slave that just became available. The foreground portion of the Activity Scheduler handles any requests that were not servicable at the background level i.e. the Ready-To-Run-Queue is serviced to the extent possible based on the current slave availabilities.

A detailed discussion of the data structures maintained by the Activity Scheduler, and the algorithms that operate on them is given in the next chapter.

3.4 Message Handler

As part of a multiactivity run-time kernel, the Message Handler is the executive process responsible for the maintenance of messages exchanged among the various application processes. In a concurrent environment that allows multiple threads of execution in a single process, certain restrictions must be observed for the system to behave deterministically: specifically, messages can only be sent or received sequentially on a given communications channel. If a process is designed such that at any time more than one concurrent thread contains a communications primitive, then a different channel will have to be used by each of those threads.

This is a reasonable restriction, since message exchange on any given channel must be deterministic. The *PAL Verifier*, for example, checks for the possibility of deadlock, by ensuring that on every communications channel there is message receive (*WTM*), for every message send (*TXM*) primitive. To simplify the task of design verification, then, PAL allows any single thread of one process to communicate with any single thread of a cooperating process over any one channel. In addition, only processes that are started by the same trigger event may cooperate by way of message passing, otherwise the temporal correctness of the system cannot be verified.

Three design options were considered for the handling of inter-process communications: 1) a common message pool for all channels, with individual channel queues containing pointers into the common pool, 2) individual channel data areas maintained as circular buffers, and 3) individual channel data areas maintained as linear buffers, where expired data in the buffer is purged by block moving the remaining data to the buffer start.

The first option was rejected primarily on the grounds that the "garbage collection" associated with the maintenance of a common message pool would be too costly in processing time. Of the remaining two options, the question was one of a tradeoff: the (per-byte) cost of pointer checking in a ring buffer, versus the (per-message) cost of purging the buffer of the expired message when it is extracted. Although one way message traffic maps nicely onto a ring buffer structure, the computational cost of the bounds checking is much higher than the single block move operation required to purge a linear buffer of an extracted message. The latter approach, therefore, was implemented in the run-time kernel.

To permit inter-process messages to be used for pure control synchronization, the Message Handler also provides a sync-message capability, in which the body of the message is null.

3.4.1 Resolution Of Requests

In the definition of the PAL language, the *Transmit_Message* (TXM) primitive is non-blocking. The implication of this in a run-time system, is that there must **always** be buffer space available to receive the body of a message being sent. Since this cannot be guaranteed under absolutely all possible circumstances, a practical implementation must allow for some form of exception handling.

This is handled in the run-time kernel by making the TXM primitive blocking in the case of insufficient data buffer space on the requested channel. In such a case, the message transmission request is queued until such time that sufficient buffer space becomes free, through the removal of a message by the receiving process. In order to free up buffer space on a communications channel, therefore, queued receive requests are serviced before queued transmit requests.

Since TXM requests may or may not be blocking, message data may be moved immediately or delayed: 1) immediate; when there is enough capacity, the message body is deposited in the channel's data buffer at the time of the transmit request, 2) delayed; if the TXM request must be queued, then the message body is only deposited into the channel's buffer at the time that the queue entry is serviced. Similarly, requests for *Wait_For_Message* (WTM) may also be blocking or non-blocking, depending on the availability of the message data at the time of request.

3.4.2 Interaction with the Processes Manager

As shown in Figure 3-8, the Message Handler interacts with the Processes Manager by way of seven service requests: it receives four from the Processes Manager for the handling of the TXM and WTM primitives, and it issues two for data movement and one for signalling state change due to service completion.

Figure 3-8: MSG_HAND Process Level View

The interaction may be described as follows:

- When a *Txm()* request is received, the message data is deposited in the channel buffer, if there is enough capacity (the message data itself is requested from the Processes Manager by way of the *Data_From_Process()* service). If the channel is full or if there are already some pending requests, the request is entered into the transmit request queue (TX_Q).
- The handling of *Wtm()* requests is similar to that of the *Txm()*.
- when the foreground portion of the Message Handler runs, it attempts to resolve queued receive and transmit requests. When a queued request is

successfully serviced, the parent application process is signalled by way of the *Signal_Process()* service of the Processes Manager

- when an application process becomes unblocked by the servicing of a queued message request, the point of blockage is then retrieved by the Processes Manager from the Message Handler by way of a *Next_Done_TXM/WTM()* request.

A detailed description of the data structures maintained by the Message Handler and the algorithms that operate on them is given in the next chapter.

Chapter 4

Design and Implementation

In setting out to build a low level run-time kernel, some specific design goals, assumptions, and limitations were first identified.

Design Goals

The design goals for the implemented kernel were twofold: 1) it had to be small, portable and efficient, and should be able to serve as the first software layer running on a target hardware platform - the second layer being the PAL based application, 2) its real time performance had to be testable on a bare-bones target machine.

For portability and efficiency, the kernel was written in C with no system dependent calls, making porting to a particular target processor, simply a matter of compiling and linking the kernel source code (and locating for ROM based systems).

For ease of testability, it was best that the development machine should serve as the test bed as well, providing an environment similar to that of a target embedded system. To that end, the unprotected, single-user single-tasking DOS platform was seen as a good choice: in this environment, programs have full access to all system hardware functions. At the same time, some of the best C language prototyping tools are available for DOS.

If DOS was to be the development platform, then, it was clear that the interaction of the central coordinator computer (the one running the kernel) with its surroundings would have to be simulated: the slave processors serving as activity executors would have to be simulated, as would the environmental "events" that the kernel must respond to. Furthermore, when measuring the system's performance, the overhead cost of the simulation portion would have to be either known and accounted for, or be negligibly small. The kernel itself, however, had to be independent of its simulated environment.

Design Assumptions

The primary design assumption was about the interconnection of the multiprocessors of the underlying architecture. Specifically, it was assumed that the system was tightly coupled by way of shared memory, where each slave processor had a separate buffer area in the memory map of the central coordinator, as was shown in Figure 2.3 (the physical location of that memory, however, may be on the slave itself). To communicate with a slave, therefore, the central coordinator simply reads from or writes to that particular area of its memory map (see also Slave Interaction Mechanism of section 2.2).

In the implemented kernel, the memory shared with the slave processors is simulated by a single buffer that is "shared" by all slaves i.e. all slaves map to the same block in the memory map of the central coordinator. Since the kernel actually implements the data movement into and out of the slave memories, its run-time performance is that of a real life implementation. The shared memory buffer was sized to be large enough to accomodate the largest data block supported by the system i.e. the largest process IO_List.

Design Limitations and Constraints

With no memory management built into the kernel, and none available from any underlying software layer (since the kernel is to be the first layer over the hardware), all data structures had to be implemented as static. In a mature development environment, the incorporation of application configuration information into an executable image of the run-time kernel would be a straightforward matter. In such a production environment, every application would be bound with the kernel into a distinct executable image.

Since this environment does not exist, it was clear that an application independent executable image would serve more easily in the testing of various configurations. This meant that the data structures compiled and linked into the run-time kernel had to be sized to accomodate the largest possible application that was going to be tested. One additional constraint was imposed on the sizing of the kernel's executable image: all code and data had to fit into 64KBytes. This was done in order to get an estimate of the allowable complexity of an application that could be implemented on an 8-bit single chip microcontroller, which is typically limited by its 16-bit address bus.

The following list, then, describes the size limitations that were imposed on the implemented, application independent, kernel. These size limits were chosen arbitrarily, confirmed to produce an executable image <64KB, and implemented as constants definitions in an include file (PAL_CNFG.H):

- **Environment Interface:** 15 events, 5 processes per event, 10 elements per IO_List, 100 environment variables
- **Processes Manager:** 20 processes, 30 IO_Lists per process, 12 elements per IO_List, 50 private data variables, 30 PAL language statements to describe the process, 100 global data variables for the whole system
- **Activity Scheduler:** 16 slave processors, 100 activities in the whole system
- **Message Handler:** 32 channels, 60 data-item buffer size per channel, 32 entry queue sizes for the send and receive message queues.

The one other area of difficulty in developing the kernel without the benefit of an integrated PAL environment, was in the creation of the expressions used by *STP* and *STS* primitives. Recall that the purpose of these language elements is to assign a new value to an output variable based on the current values of one or more input variables, and the expression that relates those inputs. The output variable is then typically tested for some form of conditional execution by the *CAS* or *RPT* constructs).

One way of generating the expressions in a production environment, would be through the use of a code generator, which would create a C source code module containing all expressions in the application. That module would then be compiled and linked into the executable image of the kernel and application. Without the benefit of the integrated environment, however, the expressions had to be created by hand. Since the expressions are highly application specific, the generic expressions that could be built into an application independent kernel would have to be general enough and simple enough to be usable with the "dummy" applications used to test the kernel. To that end, the implemented expressions perform simple decrement operations on integer type variables.

4.1 Design Overview

As was shown in Figure 3-1, the run-time kernel is implemented as four Executive Processes that interact by way of message passing. Each of these processes is said to have multiple entry points: foreground, service request (one for each message that it receives), and, for some, interrupt (or background). Being the first software layer to run on the bare hardware platform, the foreground entry points of the executive processes are scheduled in a simple non preemptive round robin fashion. As shown in the C code fragment, below, each process is allowed to run to (foreground) completion before the next is started:

```

System_Init();
do
{
    Environment_Interface();
    Processes_Manager();
    Activity_Scheduler();
    Message_Handler();
}
while(TRUE);

```

To minimize overhead, the first action performed within the foreground of each executive process is to determine the need for its own services, and if none are required then control is relinquished:

- **Environment Interface:** quit if there are no events currently queued
- **Processes Manager:** quit if no application processes currently active
- **Activity Scheduler:** quit if all slaves are busy or if Ready-To-Run queue is empty
- **Message Handler:** quit if both receive and transmit queues are empty.

Since the entire kernel is created as a single executable image, the service request entry points of the executive processes are implemented as simple function (subroutine) calls, where the service request call of one module (executive process) is merely resident in another source code module, and is resolved when the program is linked.

As depicted in Figure 4-1, each executive process was designed as a set of code modules, one module for each type of entry point, with each module containing multiple routines. The only exception to this structuring is the Processes Manager, whose foreground processing was further partitioned into separate Graph Traversal, Construct, and Primitive handling modules (not surprising, since it is the Processes Manager that is the 'central' executive process). The two other modules completing the overall run-time kernel, are the system startup initialization module, and the module containing the routines for simulating environmental event and slave processors.

Figure 4-1: System Code Modules

System Initialization

To properly configure an application, system startup initialization in the DOS development environment is controlled by the reading of two application configuration files, and by interpreting several command line parameters. The application's event and process definitions are found in the files, while the number of slaves, and number of

communications channels are passed on the command line. Event and slave simulation parameters are also passed on the command line.

The application configuration files are implemented as simple sequential text format, to allow the information to be entered with any text editor. They are discussed in greater detail in Chapter 5. The command line parameters are interpreted as follows:

```

/C:n      the configuration to load & run
/S:n      the number of (homogeneous) slave processors
/E:mm:nn  the min. & max. time between events (used as
           limits on a random number generator)
/A:n      duration of an Activity, in clock ticks
/M:n      the number of communications chans
/T:n      run duration in minutes (for batch testing)

```

In a ROM based embedded application, the system startup would consist of moving the application's configuration information from ROM into the RAM resident run-time data structures. In either case (file or ROM based configuration initialization), data initialization is performed for all environment, global and private application process data variables, all queues are set to empty, and all application processes are marked as inactive, except for those processes that run as continuous loops (declared with no associated trigger event). As will be seen in Section 4.3, the Descriptor Table of each process must also be initialized when the application process is invoked, because of the dynamic data elements contained in that data structure.

The remaining subsections of this chapter, then, describe the design of the executive processes through explanations of their private data structures, and the algorithms that operate on them. The data structures are described as they were implemented. For brevity and clarity, the algorithms are given in pseudocode, which serves to highlight the conceptual aspects and while simultaneously hiding the language dependent details.

4.2 Environment Interface

Designed as three code modules, one for each of the Foreground, Background, and Service entry points, the Environment Interface is the executive process responsible for:

- the maintenance of environment data in the Central Coordinator
- the correspondence of events to responses, and the activation of a response's process(es) upon the occurrence of its event
- the queuing of event occurrences and their data, to handle the condition where an event's response was not ready for execution when the event occurred
- the handling of the *wait* (WTE) and *delay* (DLY) language primitives as message based services provided to the Processes Manager.

Serving as the system interface to the environment's events, most of the work done by this executive process is in the **background**, by way of an interrupt routine that is activated in response to an event. The **foreground** entry point performs the checking of queued events, as required for the invocation of processes that may have completed since the last occurrence of their associated event, and for which that event has a backlog of queued occurrences. The multiple **service** entry points are in the form of callable subroutines that respond to the Processes Manager's requests for service for the WTE and DLY primitives. Requests by the Processes Manager for information on completed WTE and DLY primitives are also handled by the service entry points, with the completions being signalled from the background by the Environment Interface as they complete. This interaction with the Processes Manager was shown in Figure 3-5. Table 4-1 shows the grouping of code functions (subroutines) into the three modules comprising the Environment Interface.

The Environment Interface could be moved entirely to the background if the termination of an application process could be signalled as an event, in which case it (the process) would be immediately 'restarted' if its associated event had occurred and been queued. Since it is the Processes Manager that 'executes' application processes, this immediate restart would only be an advantage if the executive processes were concurrent themselves, otherwise the 'execution' of a process has to wait until the Processes Manager becomes active. In the implemented design, where the executive processes run sequentially and are non-preemptive, an application process is signalled to become *Ready*

```

FOREGROUND:  Env_If()
                Get_Quued_Data()

BACKGROUND: Service_Event()           ;event interrupt
                Get_Event_Data()
                Invoke_Responses()
                Queue_Response()
                Service_WTE_Requests()
                Service_DLY_Requests()   ;timer interrupt

SERVICES:   Wte()
                Dly()
                Next_Done_Wte()
                Next_Done_Dly()
                Read_Env_Var()

```

Table 4-1: Modules of the Environment Interface

by the Environment Interface when its trigger event occurs, but is not actually *Activated* until the next pass of the Processes Manager. There is a small, finite probability that the event will occur just before Processes Manager tests the corresponding process for eligibility to run, in which case startup latency will be near zero.

At the occurrence of an event interrupt, then, the Environment Interface activates eligible processes, and unblocks processes that are currently blocked by the WTE primitive, waiting for the event in question. The change in process state is signalled to the Processes Manager by way of the Signal_Proc service, used with the appropriate parameter to indicate either process startup or WTE completion. At the occurrence of a timer tick interrupt, all unexpired DLY requests are updated, and delay completions are signalled to the Processes Manager in a similar manner.

When a system contains asynchronous program elements, such as the background interrupt entry point of the Environment Interface, the integrity of data areas that are shared by the background and foreground must be protected in the foreground in all

critical areas as uninterruptible code. In the Environment Interface, the critical areas revolve around the updating of queues in the foreground:

- the 'unqueueing' of events as responses become available, and the queues are moved up,
- the 'unqueueing' of completed WTE and DLY requests in their respective service queues.

As was shown in Figure 3-5, the Environment Interface maintains five private data structures. The **Data_Area** and **IO_List_Area** are used to store and control (respectively) the movement of data associated with the various environmental events. The **Wait_Table** and **Delay_Table** serve as the WTE and DLY service request queues, respectively. The **Event_Table**, however, is the primary data structure handled by the Environment Interface, and the most complex.

4.2.1 Event Handling

In response to the occurrence of an event, the background level processing (interrupt routine) of the Environment Interface must perform the following functions:

- if an event generates data, then that data must be transferred from the memory area shared with the slave that caused the event into the **Data_Area** of the Environment Interface,
- invoke all the *Ready* processes that use this event as a startup trigger. When a process is successfully invoked, the event data may also be passed to it if so configured in the application,
- if the event is non-volatile and one or more processes were not successfully invoked, then queue the occurrence and data of the event,
- check the queue of processes waiting on this event (WTE) and signal them that their request has just been completed.

To illustrate the implementation of the Environment Interface's event handling, the pseudocode algorithm of the interrupt service routine is given as follows:

```

Service_Event(Event_Id)          /* interrupts are off */
{
    Get_Event_Data(Event_Id) if this event has data
    Invoke_Responses(Event_Id);
    Service_WTE_Requests(Event_Id);
    enable interrupts
}

```

Routine Get_Event Data() uses the data source and destination fields of the primary event record (Figure 4-2) to control the movement of event data (if there is any) into the Central Coordinator. The data destination is controlled by an IO_List, and the source of the data is assumed to come from the memory area shared with the slave that caused the event. The algorithms of routines Invoke_Responses() and Service_WTE_Requests() can be described as follows:

```

Invoke_Responses(Event_Id)
{
    repeat for all processes triggered by this event
        signal the process to start up
        if startup successful then
            if process expects startup event data then
                transfer event data to process
            endif
        else
            if event is non-volatile then
                if event queue is not full then
                    Queue_Response()
                endif
            endif
        endif
    endrepeat
    if any processes were queued then
        queue the event's data
    endif
}

```

```

Service_WTE_Requests(Event_Id)
{
    quit if no processes waiting on any events
    repeat for number of pending entries
        find next pending entry
        if entry uses Event_Id then
            signal the process of the service completion
            move event data to process
            update wait table with completed entry
        endif
    endrepeat
}

```

The primary data structure used by the Environment Interface in the processing of events, is the Event_Table, which contains the correspondence of processes to events, as well as the relationship of an event and its associated data. As shown in Figure 4-2, each event in the system has a primary record, which defines the event in terms of its queueing and required data handling. Each event also has one or more secondary records, one for each of the processes associated with this event. Note that a process cannot be triggered by more than one event, but an event can trigger more than one process.

Figure 4-2: Event Table

The C language declaration of an event record is simply:

```
struct Event_Record
{
    char    Q_Len, Q_Cnt, E_V_List, *Data_Ptr,
           Proc_Cnt, Q_Data_Buf[..];
    struct Process_SubRecord
    {
        char  Proc_Id, Q_Cnt, Q_Idx, P_V_List, Wait;
    }Proc[..];
};
```

The discrete table elements `Q_Evts` and `Evt_Cnt` represent the current number of queued events, and the total number of events configured into the system, respectively. It is the counter `Q_Evts` that is tested to determine the need for the foreground level service of the Environment Interface. If there are queued events, then they are serviced and `Q_Evts` decremented (`Q_Evts` is incremented in the background, when an event occurrence is queued).

The fields of the primary record of an event are described in the following point form summary:

- `Q_Len` (Queue_Length): the allowable queueing depth of this event. A value of 0, 1, or n, is used to indicate that the event is *Volatile*, *Latched* or *Queued*. A *Volatile* event is one whose occurrence is ignored if there are no corresponding processes that are *Ready* for activation at the time of the event. A *Queued* event can remember the last `Q_Len` occurrences that were not serviced at the time of the event. A *Latched* event simply remembers that there was at least one occurrence that was not serviced.
- `Q_Cnt` (Queue_Count): the number of times that this event occurred and had to be queued because one or more of its associated processes were not *Ready* (all *Ready* processes are activated immediately).
- `E_V_List` (Event_Variables_List): the identifier of the I/O descriptor list that contains the data variables associated with this event. When an event occurs, all data associated with it is moved to the `Data_Area` of the Environment Interface to update the current state of all Environment Variables. If there is

- no data associated with this event, this field will contain a NULL entry.
- **Data_Ptr** (Data_Pointer): pointer to the memory area shared with the device (slave or sensor processor) causing this event. This field represents the source operand and is used in conjunction with the destination **E_V_List** field, when event data is to be moved into the Environment Interface **Data_Area** by routine **Get_Event_Data()**.
- **Proc_Cnt** (Processes_Count): the number of processes that are associated with, and triggered by this event. An event has **Proc_Cnt** secondary records. If there are no processes associated with an event (**Proc_Cnt** = 0), then the event is only used for the acquisition of environment information, and/or for the servicing of processes that use the WTE primitive.
- **Q_Data_Buf** (Queued_Data_Buffer): alternate area used for holding the Environment Variables associated with the event, when the event occurrence and its data is to be queued. When constrained by the 64KB limit of 8-bit microcontrollers, the buffer is sized (statically) to be large enough to hold data for up to 5 event occurrences, using the longest allowable **IO_List**.

The secondary records, one for every process that is triggered by this event, can be described as follows:

- **Proc_Id** (Process_Id): identifier of the process to be activated as a result of the occurrence of this event. Note that since a process can only be started by one event, there is only a single occurrence of any given **Proc_Id** in the entire Event Table.
- **Q_Cnt** (Queue_Count): the number of queued occurrences of the event for which this particular process was not ready for execution. A separate **Q_Cnt** is kept for each process associated with the event to allow for handling of multiple eligible processes per event, where not all processes will be at the same execution state at the time of the event.
- **Q_Idx** (Queue_Index): array of up to 5 entries, used to indicate which queued occurrence of the event's data should be used when starting a process whose event occurrence(s) was queued.
- **P_V_List** (Process_Variables_List): the identifier of the process I/O descriptor list used to control which of the process variables are to receive the data that is associated with the occurrence of the event. A NULL entry is used to indicate that the process does not wish to receive any of the event

data.

Recall that the implemented design of the run-time kernel does not support multiple copies of a process. In a system that did, however, only a single Q_Cnt field would be kept for all copies of the process associated with that particular event. To indicate that the event was serviced by a response, the one Q_Cnt field would then be decremented whenever the primary process *or any one of its copies* were invoked.

4.2.2 Handling of Queued Events

With the background entry point handling all event occurrences, the foreground entry point of the Environment Interface is used to check for and activate any processes that may not have been ready at the time of their trigger event, but which may have since become eligible. Applicable only to Latched or Queued events, the algorithm of the Environment Interface foreground routine is as follows:

```
Env_If()
{
    if there are no queued events at all then return
    repeat for all events configured into the system
        skip to next event that is queued
        repeat for all processes associated with this event
            if process's event has been queued then
                signal process to activate it
                if activation was successful then
                    disable interrupts
                    if process needs startup data then
                        Get_Queued_Data();
                    endif
                    update Event table
                    enable interrupts
                endif
            endif
        endrepeat
    endrepeat
}
```

where, routine `Get_Queued_Data()` must perform the following functions:

- establish the point in the `Queued_Data_Buffer` from which to get the data
- move the queued data to the process by requesting the `Data_To_Proc()` service from the Processes Manager
- update the queueing fields of this process's record in the Event Table
- if there are no other references to this queued event data by other (queued) processes, then discard it and move up the `Queued_Data_Buffer`
- adjust all other Event Table references to queued data that is older than the data just removed.

4.2.3 Environment Interface Services

The five services offered by the Environment Interface to the Processes Manager, for the handling of the WTE, DLY, and RDE language primitives are: `WTE()`, `DLY()`, `Next_Done_WTE()`, `Next_Done_DLY()`, and `Read_Env_Var()`. The first two are requested by the Processes Manager when a new WTE or DLY primitive is encountered and needs servicing, and the second two services are requested (also by the Processes Manager) when a process is being updated and the completed requests are being searched. The last service is non-blocking and is used by an application process to copy the contents of an Environment Variable to one of its private data variables. Figure 3-4 illustrated this interaction between the two executive processes.

The data structures maintained in support of the queueing of WTE and DLY requests are the **Wait_Table**, and the **Delay_Table**, respectively. Since these tables are updated in the background by event and timer tick interrupts, respectively, the (foreground) service entry points must contain uninterruptible code sections to protect the integrity of the data structures. Recall that in PAL, an application process may have several threads of execution and may, therefore, be waiting on more than one event at the same time.

Handling of the WTE Primitive

The *Wait_For_Event* (WTE) primitive is used by application processes to synchronize internal execution points to system events. The handling of the WTE primitive is the responsibility of the Environment Interface, which provides the required services to the

Processes Manager. To help understand the `Wait_Table` data structure and its handling, then, it is useful to review the interaction of the Environment Interface with the Processes Manager in the processing of the WTE language element. The following point form summary describes that interaction:

- When the Processes Manager encounters a WTE language element in the graph structure of an application process, a request for WTE service is issued to the Environment Interface.
- The Environment Interface responds by queueing the request as a new entry in the `Wait_Table`.
- At the next occurrence of an event, the Environment Interface checks the table for pending requests that are waiting for the event being serviced. Matching records are marked as `DONE`, and the processes that had issued the requests are signalled of their WTE completions. In addition, if a process is configured to receive the event data, then that data is moved into the process's private data area.
- On the next pass of the Processes Manager, unblocked (*Active*) processes are checked for new WTE completions. When one is found, a `Next_Done_WTE()` request is issued to the Environment Interface to retrieve information about that completion.
- The Environment Interface responds by removing the referenced entry from the `Wait_Table`, and moving up the requests queue. Note that the addition and removal of records from this table is a critical code (uninterruptible) section.

Figure 4-3 shows the implementation of the `Wait_Table` data structure. The table elements are described as follows:

- **Ent_Cnt** (`Entries_Count`): the current number of entries in the table, whether they are still pending or have already been serviced. Used to speed up search algorithms.
- **Pend_Cnt** (`Pending_Count`): the current number pending requests. If no requests are pending, then the table needs no updating at the present occurrence of the event.
- **Proc_Id** (`Process_Id`): identifier of the process that issued this request.
- **Evt_Id** (`Event_Id`): identifier of event is being waited for.
- **Var_List**: `IO_List` used by the process to specify which of its private data

Figure 4-3: Wait Table Data Structure

variables are to receive the data that is associated with the occurrence of the event.

- **Elm_Id (Element_Id):** identifier of the point in the requesting process's graph structure, from which the WTE primitive was issued. That location is the point from which execution will resume when the request is completed. This field is returned to the Processes Manager as the Environment Interface's response to the Next_Done_WTE() service request.
- **Stat (Status):** flag used to indicate the current state of the request as either pending or completed.

In this initial implementation of the run-time kernel, handling of the WTE primitive has been designed to use this single table of fixed length records, where the efficiency of record searching is likely to be adequate. In a much larger system, a more efficient implementation may be needed to reduce the search time, such as one with a separate sub-table for each event.

The size of the Wait_Table data structure is determined at system 'compile' time, by counting the number of possibly parallel occurrences of the WTE primitive in each process that contains it. The length of the table, then, is the sum of all possibly parallel WTE primitives in the system. The order of WTE request entries in the Wait_Table is unimportant since all pending entries are serviced at every occurrence of an event.

Handling of the DLY Primitive

The Delay (DLY) PAL language primitive is used by application process to implement a general wait mechanism. The handling of the DLY primitive is the responsibility of the Environment Interface, which responds to timer tick interrupts and provides the required services to the Processes Manager. The interaction between the Environment Interface and the Processes Manager in the handling of the DLY language element is very similar to that used in the processing of the WTE primitive. It can be described as follows:

- When the Processes Manager encounters a DLY language element in the graph structure of an application process, a request for DLY service is issued to the Environment Interface.
- The Environment Interface responds by queueing the request as a new entry in the Delay_Table.
- At the next occurrence of a timer tick, the Environment Interface checks the table for non-expired timeouts, and decrements their time remaining. If the decrement results in expiry, then the corresponding process is signalled of the completion of its DLY request.
- On the next pass of the Processes Manager, unblocked processes are checked for new DLY completions. When one is found, a Next_Done_DLY() request is issued to the Environment Interface to retrieve information about that completion.
- The Environment Interface responds by removing the referenced entry from the Delay_Table, and moving up the requests queue. Note that the addition and removal of records from this table is a critical code (uninterruptible) section.

Figure 4-4 shows the implementation of the Delay_Table data structure. The table elements can be described as follows:

- **Ent_Cnt** (Entries_Count): the current number of entries in the table, whether they are still pending or have already been serviced. Used to speed up search algorithms.
- **Pend_Cnt** (Pending_Count): the current number pending requests. If no requests are pending, then the table needs no updating with this occurrence of a clock tick.

Figure 4-4: Delay Table Data Structure

- **Proc_Id** (Process_Id): identifier of the process that issued this request.
- **Time_Out**: the remaining time of this delay request (in clock ticks)
- **Elm_Id** (Element_Id): identifier of the point in the requesting process's graph structure, from which the DLY primitive was issued. That location is the point from which execution will resume when the request is completed. This field is returned to the Processes Manager as the Environment Interface's response to the Next_Done_DLY() service request.

The size of the Delay_Table is determined at system 'compile' time by counting the number of possibly parallel occurrences of the DLY language element in each process that contains it. The length of the table, then, is the sum of all possibly parallel DLY primitives in the system. The order of DLY request entries in the Delay_Table is unimportant, since all pending entries are serviced at every occurrence of a clock tick.

4.2.3 Data Storage and Movement

Most events requiring a response, also generate data on which the response operates. The data associated with an event, therefore, must be moved in an orderly fashion into the internal storage area of the system. As was shown in Chapter 3, an IO_List data structure is used to control the assignment of data associated with the occurrence of an event, to specific Environment Interface data variables. The data variables themselves are, in turn, grouped into the Environment Interface's internal storage area known simply as the Data_Area.

4.3 Processes Manager

To take advantage of the inherent parallelism of a multi-microprocessor machine, it is the Processes Manager that controls and maintains all threads of all application processes running in a system. Clearly, parallel threads of execution controlled by a sequential machine (as are effectively all microcontrollers and microprocessors) can only be properly maintained if every active parallel thread of every application process is updated with every pass of the Processes Manager.

Designed as four modules, the Processes Manager is the 'central' executive process, in that all other executive process interact only with it and not with each other (see Figure 3-1: The Kernel's Executive Processes). As shown in Table 4-2, the routines comprising the Processes Manager are grouped into four functional modules, with the first three (Graph_Trav, Constructs, and Primitives) making up all the processing performed by the **foreground** entry point. The multiple **service** entry points are in the form of callable subroutines that respond to requests from the other executive processes for data movement and process signalling services.

The Processes Manager, then, is responsible for:

- 'Executing' all application processes by interpreting the precedence relations of their Descriptor Tables (traversing their graph structures).
- Maintaining the current execution state of each application process in the system (see Figure 3-2: Process States).
- Maintaining the private data of each application process, by providing data movement services to the other executive processes. Data is generated by events, messages and activities, and is also required as input by messages and activities.
- Maintaining the global billboard data area by handling the *Read_System_Variable* (RDS) and *Set_System_Variable* (STS) language primitives, which are used by application processes to gain access to the global area.

The global data area is called *System_Data*, and simply contains all the variables that were declared by all application processes in their STS and RDS primitives.

GRAPH_TRAV:	Proc_Man() Start_Proc() Update_Proc() Trace_Thread() Do_Elmnt()
CONSTRUCTS:	Seq() , Con() , Cas() , Rpt()
PRIMITIVES:	Stp() , Sts() , Rds() , Exec_Expr()
SERVICES:	Data_To_Proc() Data_From_Proc() Signal_Proc()

Table 4-2: Modules of the Processes Manager

To speed up run-time processing, the Processes Manager uses a combination of straight tables and linked lists. Since a system compiler does not yet exist, and the tables were constructed by hand for the purpose of testing, most linking in the lists is in the form of offsets from known addresses rather than the addresses themselves.

The Processes Table data structure, shown in Figure 4-5, is used in the handling of all application processes in the system. The overall table, can be expanded at several levels to reveal a multilayered sub-table, record and sub-record structure. Each application process in the system has its own process sub-table, which is comprised of a primary process record, and IO_List_Area, a Data_Area, and a Descriptor Table. The IO_List and Data Areas are implemented much the same as in the Environment Interface, and are used to control data movement and store data, respectively. The primary process record contains fields that maintain the current process execution state, and current state of service requests and completions. The Descriptor Table contains the data structure that describes the process's graph structure, and is discussed at length later in this chapter.

The discrete table elements **Procs_Act** and **Proc_Cnt** represent the current number of application processes in the *Active* state, and the total number of processes configured

Figure 4-5: Processes Table

into the system, respectively. It is the **Procs_Act** field that is tested to determine the need for the foreground level service of the Processes Manager.

The elements of the Processes Table, then, are described as follows:

- **State:** the current execution state of the process. Can be any one of *Inactive*, *Ready*, *Active*, or *Blocked*. The process states and their transitions were discussed in Chapter 3.
- **Event:** flag used at system startup time to identify a process as having no trigger event. Such a process runs as a continuous (infinite) loop and is only started once at power on initialization.
- **Compl_Rqsts** (Completed_Requests): containing a field for each of the blocking PAL language elements (Figure 4-6), this sub-record is used in maintaining the completion status of all outstanding service requests made by the process. When a completed service request is signalled to the Processes

Figure 4-6: Completed Service Requests

Manager by one of the other cooperating executive processes, the appropriate field of this sub-record is updated (incremented). When the process is next 'executed', the point(s) of blockage are determined using this sub-record.

- **IO_List_Area:** contains all the lists of variables that are used to control the movement of data into and out of the process's private Data_Area. Used in the processing of inputs and outputs of activities, messages and events. The actual process variables (data) reside in the Data_Area, and they are merely referenced here by a single level of indirection.
- **Data_Area:** contains all the process private data variables. In this prototype implementation, only the primary data types of the C language are supported: *char* (1 byte), *int* (2 bytes), *long* (4 bytes), or *float* (4 byte real).

The C language declaration of a Process Sub-Table is given as follows:

```
struct One_Process
{
    unsigned char    State, Event;
    Struct Completed_Requests
    {
        unsigned char  Wte, Dly, Txm, Wtm, Act;
    } Compl_Rqsts;
    Var_List    IO_List[..];
    Data_Var    Proc_Var[..];
    Lang_Elmt    Elmnt[..];
}
```

where, all variables declared as *unsigned char* are used as either flags or counters. The *Var_List* data type is a previously described IO_List, and *Data_Var* data type is a process private data variable. The *Lang_Elmt* data type is a language element as used in

the process's Descriptor Table, and is discussed in detail in Section 4.3.2, The Descriptor Table.

With every pass of the Processes Manager, every application process that is currently *Active* or *Ready* is 'executed' to the point where it either becomes *Blocked*, or runs to completion and is marked as *Inactive*. An *Inactive* process is invoked as an event's response and becomes *Ready*. The Processes Manager, which maintains the execution state of all processes, must therefore handle application processes that are ready to run in either of the *Ready* or the *Active* states.

Asynchronous system elements, such as environmental events or slave completions, may cause transitions in the execution state of application processes and may occur at any time. As shown in the following algorithm, to properly handle process state transitions, the Processes Manager protects its foreground processing with uninterruptible code sections and by working with a copy of the current service completions:

```

Proc_Man()
{
    quit if there are no Procs_Act
    repeat for every process in system
        skip process if its state is INACTIVE or BLOCKED
        disable interrupts
        if process state is READY then
            enable interrupts
            new process state = START_PROC(Proc_Id)
        else
            save Compl_Rqsts and enable interrupts
            new state = UPDATE_PROC(Proc_Id, Compl_Rqsts)
        endif
        if new state is DONE then
            disable interrupts and mark process as INACTIVE
            decrement count of active processes
            enable interrupts
        endif
    endrepeat
}

```

All initialization of a starting application process is performed by routine `Start_Proc()`, which must consist of the following:

- initialize the dynamic fields of the *Concurrence* and *Sequence* language elements in the process's Descriptor Table,
- clear all service request completions,
- initiate 'execution' of the process by starting traversal at the beginning of the graph structure i.e. setting the first element to be executed to the first element in the Descriptor Table.

The 'execution' of a process which is found in the *Active* state, is restarted at the point(s) of blockage by routine `Update_Proc()`. The method used to traverse the graph and to determine where to restart is discussed in the next subsection.

4.3.1 Graph Traversal

In a PAL based multiactivity system, where application processes are defined by graph structures comprising multiple concurrent threads of control, the algorithm used to traverse the graphs is clearly the heart of the system. Graph traversal must be, above all, efficient, where an efficient implementation will likely also result in a simple design. The problem, then, was twofold: 1) how to keep track of the current execution point in multiple concurrent threads of a process's graph structure, and 2) how to best traverse the nested layers of control constructs (the PAL language elements defining the graph).

Several approaches were considered including: 1) a stack based algorithm, whose inherent Last-In-First-Out nature was found to be ill-suited to the random order of service request completions, and 2) a forward tracing algorithm whose overhead increased as the thread approached completion (because of the retracing of many already completed elements to find the current execution point in a thread).

To avoid the above problems, one element of a graph traversal solution was clear: the point of blockage of a thread had to be saved and then later retrieved when the thread became unblocked. In such a scheme, the graph point where the blockage occurred is included along with the request for the service which produced the blockage. When the service request becomes completed, the blockage point in the graph is returned with the

completion (in reality, the Processes Manager must request this information).

The second part of the graph traversal solution had to address the handling of nested *Construct* elements. Although nesting is known to be best served by a stack based algorithm (as implemented by most procedural programming languages), it had already been established as unsuitable for this application. As illustrated in Figure 4-7, two linked list options were considered for the implementation of the nesting of *Constructs*. In the figure, Con,x and Seq,x are used to signify a *Sequence* consisting of x elements, or a *Concurrence* consisting of x parallel threads.

In both linked list options, every blocking language element has a link back to its 'parent' construct (dashed line, shown only for the first construct). This allows the parent construct to be located immediately as the thread becomes unblocked and is traced backward. The variation, then, is in the implementation of the forward link usage.

Option 1 uses a forward link from a construct to locate the first executable language element of that construct. In such a structure, a construct's first element may lie several records away since all the elements for one nesting level are entered in the list before any of the elements of the next nesting level, and so on for all levels of nesting.

The forward link in option 2 is used to locate the next language element of the same nesting level, but only when skipping over language elements nested by a *Construct* at that level. This results in a layout of the same structured representation as that entered by the user. This second option had two further advantages: it was much easier to 'link' by hand, but more importantly, it was more efficient in its use of forward links. The graph traversal algorithm, then, was implemented using option 2, and can be generally described as follows:

- a process is started (or resumed) by tracing a thread in the forward direction until the thread becomes blocked;
- the graph is then traversed backwards using the links to 'parent' construct elements, until an eligible thread is found which can be once more traced in the forward direction;
- this procedure is followed until all threads are either blocked or completed.

```

Con,2
.  A1
.  Seq,3
.  .  A2
.  .  Con,2
.  .  .  Seq,3
.  .  .  .  A3
.  .  .  .  A4
.  .  .  .  A5
.  .  .  Seq,2
.  .  .  .  A6
.  .  .  .  A7
.  .  A8

```

Option 1

Option 2

Figure 4-7: Nesting of Constructs

When a thread is being traced in the forward direction and a completed *Construct* element is encountered, the elements nested under that *Construct* are skipped and the next element of the currently executing expression is located by way of the (completed) *Construct*'s forward link.

Thread Tracing

When a process is determined to be in the *Active* state, 'execution' of the process resumes at the point(s) of blockage. This approach is implemented in routine `Update_Proc()`, where all threads blocked by one primitive are traced all the way to process completion or blockage of the thread. The same tracing is then applied successively to threads blocked by each of the remaining blocking language primitives:

```
Update_Proc(Proc_Id)
{
    check service request completions
    if any 'WTE' then State = TRACE_THREADS(Proc_Id, WTE);
    if any 'DLY' then State = TRACE_THREADS(Proc_Id, DLY);
    if any 'TXM' then State = TRACE_THREADS(Proc_Id, TXM);
    if any 'WTM' then State = TRACE_THREADS(Proc_Id, WTM);
    if any 'ACT' then State = TRACE_THREADS(Proc_Id, ACT);
    return(State);
}
```

The actual thread tracing is done by routine `Trace_Threads()`, which uses the backward links of the language elements to identify the 'parent' construct of the starting point. The completion state of that construct is then updated, and if it becomes complete, then the completion state of its 'parent' construct gets updated. and so on. In other words, the thread is traced back as long as constructs become completed i.e. all elements of a *Sequence* have been completed, all parallel threads of a *Concurrence* have been completed, etc.


```

Trace_Thread(Proc_Id, Primitive)
{
    loop for all completions of this Primitive
    case Primitive
        WTE: Elmnt = Next_Done_WTE(Proc_Id);
        DLY: Elmnt = Next_Done_DLY(Proc_Id);
        TXM: Elmnt = Next_Done_TXM(Proc_Id);
        WTM: Elmnt = Next_Done_WTM(Proc_Id);
        ACT: Elmnt = Next_Done_ACT(Proc_Id);
    endcase
    do
        if Back_Lnk of Elmnt is NULL then return(DONE);
        else get 'parent' construct of Elmnt
        case type of construct
            SEQ: update execution state
                if not done then execute next SEQ elemnt
            CON: update execution state
            CAS: always done
            RPT: execute RPT and check for done
        endcase
        while (construct is DONE);
    endloop
    return(stat);
}

```

Backward tracing continues until a construct is encountered, whose execution state is incomplete, at which point thread tracing resumes in the forward direction and continues until a blocking language element is encountered, or until the process executes to completion. Recall that the elements nested under a completed *Construct*, are skipped (by way the forward links) when a thread is traced in the forward direction. Note that the point of blockage itself was retrieved from one of the other cooperating executive processes by way of a *Next_Done...()* service request. Also note that forward tracing of a thread always restarts at a *Construct* language element. The handling of the specific language elements is discussed in section 4.3.3.

4.3.2 The Descriptor Table

The graph structure controlling the execution of an application process is handled at run time by a process's Descriptor Table. Each language element entry in the table is a fixed length record, with the first field always used to identify the element type and the last field always used as the back link to its 'parent' construct. As was discussed in Chapter 2, a language element can be one of four possible *Constructs*, one of eight possible *Primitives*, or an *Activity*. Although the different element types have different field counts and sizes, their implementation as fixed length records allows the Processes Manager algorithms to treat all language elements as a 'Union of Structures' (using C language terminology). The record structures of the various language elements are discussed below.

Constructs

As shown in Figure 4-8, the structure of all constructs is similar in that they all share the *Next_Lnk* and *Back_Lnk* linked list fields, which are interpreted the same way for all types of constructs. The interpretation of the remaining two fields is dependent of the type of construct. The fields of a *Construct* language element record, then, are interpreted as follows:

Sequence	SEQ	Elm_Cnt	Done_Cnt	Next_Lnk	Back_Lnk
Concurrency	CON	Elm_Cnt	Done_Cnt	Next_Lnk	Back_Lnk
Case	CAS	Elm_Cnt	Proc_Cond	Next_Lnk	Back_Lnk
Repeat	RPT		Proc_Cond	Next_Lnk	Back_Lnk

Figure 4-8: Construct Language Elements

- **Elm_Cnt** (Element_Count): the interpretation of this field depends on the type of construct being handled. For a *Sequence* construct, it represents the number of language elements bounded by this construct, not including elements belonging to nested constructs. For a *Concurrence* construct, it represents the number of parallel threads belonging to the construct. For a *Case* construct, it represents the number of branches in the case.
- **Done_Cnt** (Done_Count): keeps track of the number of completed elements of a *Sequence*, or the number of completed threads in a *Concurrence*.
- **Proc_Cond** (Process_Condition): identifier of the process's private variable that is to be tested for the conditional execution. In a *Case* construct it is tested to determine which branch to take (if any), and in a *Repeat* construct it is used to test for end of repeat condition.
- **Next_Lnk**: forward link used to find the next table element that is at the same level of nesting. It is used for skipping over completed constructs at lower nesting levels. This field is used in conjunction with the **Elm_Cnt** field.
- **Back_Lnk**: backward link used to find the construct one nesting level above (i.e. the 'parent' construct). The **Back_Lnk** is used in the tracing of process threads back to their source. A **Back_Lnk** value of NULL means that this is the entry point of the process.

Primitives

As can be seen in Figure 4-9, except for elements that represent similar operations (such as STP/STS, RDE/RDS, and TXM/WTM primitive pairs), the records implementing the PAL language's *Primitives*, share little more than the **Back_Lnk** and the element type identifier. The last four primitives shown in Figure 4-9 are all blocking, the first four are not. The fields of a *Primitive* language element record, then, are interpreted as follows:

- **Proc_Cond** (Process_Condition): identifier of the process's private variable that is the destination operand of the primitive's function.
- **Sys_Cond** (System_Condition): identifier of the global billboard variable that is either the source (in RDS) or the destination (in STS) operand of the primitive's function.

Set Process Variable	STP	Proc_Cond	Expr_Id	
Set System Variable	STS	Sys_Cond	Expr_Id	
Read System Variable	RDS	Proc_Cond	Sys_Cond	
Read Environment Var.	RDE	Proc_Cond	Env_Cond	
Transmit Message	TXM	Chan	Var_List	Back_Lnk
Wait For Message	WTM	Chan	Var_List	Back_Lnk
Wait For Event	WTE	Evt_Id	Var_List	Back_Lnk
Delay	DLY	Amount		Back_Lnk

Figure 4-9: Primitive Language Elements

- **Env_Cond** (Environment_Condition): identifier of the environment variable source operand used by the *Read_Environment_Variable* (RDE), primitive.
- **Expr_Id** (Expression_Id): identifier of the Expression used by the STS and STP primitives in the assignment of a new value to the destination operand. The expression itself is coded and linked into one of the the Processes Manager code modules at system compile time.
- **Chan**: the Channel identifier to be used for the message transaction.
- **Evt_Id** (Event_Id): identifier of the event to wait for (in the WTE primitive).
- **Var_List** (Variables_IO_List): identifier of the process's IO_List that is to be used in the transaction to move data into or out of the process's private Data Area.
- **Amount**: the number of time units to delay with the DLY primitive.

- **Back_Lnk:** used only by the blocking primitives, the backward link is used to find the construct one nesting level above (i.e. the 'parent' construct). The Back_Lnk is used in the tracing of process threads back to their source. A *Primitive* will never have a NULL Back_Lnk, since the first language element in a Descriptor Table will always be a *Construct*.

Activity

Representing the schedulable unit of work in multiactivity systems, the *Activity* language element (Figure 4-10) only specifies which activity is being requested (Id field), and what comprises the activity's input and output (In_List and Out_List identifiers). The Back_Lnk field is used as in all other language elements.

Activity	ACT	Id	In_List	Out_List	Back_Lnk
----------	-----	----	---------	----------	----------

Figure 4-10: Activity Language Element

4.3.3 Handling of the Language Elements

As previously discussed, a PAL language element falls into one of three categories: *Construct*, *Primitive*, or *Activity*. Constructs define the execution precedence relations of the other two element types. Non-blocking primitives are 'executed' directly by the Processes Manager while blocking primitives are handled as service requests and are passed on to the Environment Interface or the Message Handler. The only exception is the *Read_Environment_Variable* non-blocking primitive which is issued as a service request to the Environment Interface. Activities are always blocking, and are passed on to the Activity Scheduler for slave processor assignment and activation.

When a process is first started, its thread(s) of control are traced in the forward direction, and if any multiple parallel threads are found, then each thread is traced until a blocking element is encountered or until it completes. Only once **all** threads have been started and become blocked, is the process considered to be in the *Blocked* state. It is only on the next pass of the Processes Manager that the process is again 'executed' (if it has since been unblocked and is found to be in the *Active* state), by picking up the ends of the threads, tracing them backward and 'executing' them forward until the process is again *Blocked* or completed.

This subsection, then, describes the handling of the *Construct* and the non-blocking *Primitive* language elements, as they are 'executed' by the Processes Manager. The discussion is organized by element type.

Constructs

Sequence. The elements of the expression bounded by this construct are 'executed' one after another until all are completed. The element at which to start 'execution' is determined by the number of already completed elements in this sequence (Done_Cnt field):

```
Seq()
{
    repeat
        determine next element to execute, using
            Next_Lnk if necessary
        state = DO_ELMNT(Proc_Id, element);
        if state is DONE then increment Done_Cnt
    until SEQ is done or state is BLOCKED
    return(state of SEQ construct);
}
```

Routine Do_Elmnt() is a simple dispatcher that redirects the handling of the language element to the function for that element type.

Concurrence. The first time that this construct is encountered, every thread is executed until a blocking element is encountered in that thread or until that thread is completed. If the first language element in a thread is a construct, then the *Next_Lnk* field of that construct is used to find the next thread of the concurrence. Note that a concurrence element is only 'executed' once (the first time it is encountered); when it is updated with the completion of one or more of its threads, only its *Done_Cnt* is updated and checked for completion:

```

Con()
{
    repeat for all threads in this construct
        determine thread starting point, using
            Next_Lnk if necessary
        state = DO_ELMNT(Proc_Id, element);
        if state is DONE then increment Done_Cnt
    endrepeat
    return(current state of construct);
}

```

Case. Execution of this construct begins with an comparison of the construct's condition variable and the number of possible branches in the case. If the test variable matches none of the possible cases, the construct is considered to be complete and is skipped. This is equivalent to the selection of a path with a do nothing operation. If the value of the condition variable falls within the range of the *Case* possibilities, then that case is taken (one and only one case will ever be satisfied at any one time). The *Case* construct itself is non-blocking:

```

Cas()
{
    evaluate process condition referenced in construct
    if no match with any case then return(DONE);
    determine Branch starting point, using
        Next_Lnk if necessary
    return (DO_ELMNT(Proc_Id, Branch));
}

```

Repeat. Execution of this construct begins with an evaluation of the construct's condition variable. If the condition variable tests false then the expression bounded by the construct is skipped, and the construct is considered to be completed. If the condition variable tests true, then the bounded expression is to be executed, and all nested SEQ and CON constructs must be initialized (with every pass of the loop):

```
Rpt ()
{
    if condition variable is FALSE then return(DONE);
    initialize all nested SEQ and CON elements, using
        Next_Lnk to find them
    return (DO_ELMNT(Proc_Id, first element of the RPT));
}
```

It is interesting to note that without a blocking language element, a *Repeat* construct becomes an infinite loop, causing the system to become deadlocked. For this reason the compiler must insert a blocking NOP (*Delay* with a value of 0), simply to create a blocking element from which the start of the construct can be found on the next pass of the Processes Manager. This means that a bounding *Sequence* construct must be modified to include the new *Delay* element. The further implication is that a *Sequence* construct must be created inside a *Repeat*, whose expression consists of a single non-blocking element (to bound the original element and the compiler generated blocking NOP). Note that in most applications, the condition variable is modified by a STP primitive contained in the expression bounded by the RPT itself.

Primitives

The three non-blocking primitives that are 'executed' by the Processes Manager are: the *Read_System_Variable* (RDS), *Set_System_Variable* (STS), and *Set_Process_Variable* (STP) language elements. The RDS primitive performs a simple copy of a global variable into a process's private data variable.

The primitives STS and STP use expressions to calculate or otherwise determine the value of the global or process variable (Condition) being set. The expressions themselves are limited to simple arithmetic and logical operations, generated as routines

by the system compiler (or by hand as in the implemented system, where no compiler existed). If the desired expression is considerably more complex, then it may be implemented as an activity.

Every expression of every process, then, is compiled, and all the resulting routines are linked into the kernel's executable image, thus eliminating the need for run-time interpretation. A routine naming convention is used such that a simple run-time jump table is used to find the expression pertaining to the particular STP or STS primitive:

```
Exec_Expr()
{
    case Proc_Id
        PROC_1: case Expr_Id
            EXPR_1: PROC_1_EXPR_1()
            ...
            ...
            endcase
        ...
        ...
    endcase
}
```

4.3.4 Processes Manager Services

Three services are provided by the Processes Manager to the other cooperating executive processes: *Signal_Proc()* for signalling service request completions or process startup requests, and *Data_To_Proc()/Data_From_Proc()* for moving data into/out of an application process's private Data Area, respectively.

The data movement services use the *IO_List* data structures to determine which of the private process variables will serve as either the source or the destination of the operation. When moving data into a process, the source is referenced by a pointer: event data and activity output data, for example, are referred to by a source pointer which identifies the memory area shared with the slave processor in question. When moving data out of a process, the destination is a pointer: activity input data, for example, must

be written to the memory area shared with the slave that is the activity executor (recall that activity input data is moved only at the time of executor assignment). The same approach is used in the movement of data that comprises messages, except that the source or destination pointers refer to the Message Handler's channel buffers.

The process signalling service of the Processes Manager implements the asynchronous portion of the process state transition handling. As service requests are handled and completed by the other cooperating executive processes, the completions are transmitted to the Processes Manager by the *Signal_Proc()* service. The invocation of a process as the response to an event is also signalled via this service. The following algorithm, then, describes the state transition handling performed by the *Signal_Proc()* service:

```

Signal_Proc(signal)
{
    disable interrupts
    if process state is INACTIVE then
        if signal is START then
            set process state to READY
            increment count of active processes
        endif
        enable interrupts and return(process state);
    endif
    case signal
        WTE: increment WTE field of the Compl_Rqsts record
        DLY: increment DLY field of the Compl_Rqsts record
        TXM: increment TXM field of the Compl_Rqsts record
        WTM: increment WTM field of the Compl_Rqsts record
        ACT: increment ACT field of the Compl_Rqsts record
        START: enable interrupts and return(process state);
    endcase
    set process state to ACTIVE
    enable interrupts and return(process state);
}

```

4.4 Activity Scheduler

Serving as the interface between the Central Coordinator and the surrounding slave processors, the Activity Scheduler is the executive process responsible for:

- receiving activity execution requests from the Processes Manager
- maintaining the request queue in a First-Come-First-Served order
- assigning activities to executors, based on the slave's availability and capability to execute the activity
- managing the activity input and output data
- performing the low level handshaking with the slave processors to ensure proper transfer of data and control
- buffering activity completions and reporting them to the Processes Manager.

Designed as three code modules, one for each of the **Foreground**, **Background** and **Service** entry points, most of the work done by the Activity Scheduler is in the handling of the Ready-To-Run queue. This RTR_Q is serviced both in the foreground and in the background, the latter in response to an interrupt signalling the completion of an activity on a particular slave. The two service entry points are in the form of callable subroutines that comprise the responses to requests from the Processes Manager for activity execution and for information on activities that were signalled (by the Activity Scheduler) as complete. Table 4-3, shows the grouping of the function subroutines into the three code modules.

FOREGROUND:	Act_Sched()	
	Service_Queue_Foreground()	
	Activate_Slave()	
BACKGROUND:	Service_Slave_Completion()	;slave interrupt
	Get_Activity_Output_Data()	
	Service_Queue_Background()	
SERVICES:	Act()	
	Next_Done_Act()	

Table 4-3: Modules of the Activity Scheduler

4.4.1 Handling of the Activity Requests

Clearly, to maximize concurrency in any multiprocessor system (such as multiactivity), a new activity must be assigned to a slave as soon as possible: at the time of the request if there is an eligible slave, or as soon as an eligible slave signals completion of its current activity. Since a slave completion is signalled as an interrupt and the executive processes run in non-preemptive round robin, one way to ensure a speedy slave restart is to assign a new activity to the slave as part of its completion interrupt service. If there is no activity that the slave is eligible to execute immediately, then it simply idles and its utilization decreases. Recall that in a heterogenous multi-processor system, slaves may be specialized, hence, not all slaves would be capable of executing a requested activity.

As in the Environment Interface, the sole purpose of the foreground portion of the Activity Scheduler, then, is to handle queued requests. Activity requests are queued if there is no eligible slave to execute the activity at the time of the request or if there already are some queued requests. Again, similar to the Environment Interface, if there are no pending queued requests (or any free slaves), then the services of the foreground portion of the Activity Scheduler are not needed and control is relinquished. An activity, then, may be assigned to a slave processor at one of three possible times:

- At the time that the activity request is received by *Act()* service's entry point. If there is an eligible slave and there are no other queued requests then it may be assigned to the first eligible slave.
- At the time of a slave completion if there is an activity that the slave is eligible to execute (performed in the background slave completion interrupt).
- During the execution of the foreground portion of the Activity Scheduler, as the request queue is serviced.

Note that before a new activity can be assigned, the output data (if there was any) of the activity just completed must be moved to the parent process's private data area. This is done by issuing a *Data_To_Proc()* request to the Processes Manager. The process must also be signalled of its activity completion, by way of the Processes Manager *Signal_Proc()* service. When an activity is assigned to a slave processor, the input data (if there is any) is fetched by way of the Processes Manager's *Data_From_Proc()* service.

Figure 4-11: the Ready-To-Run-Queue

The data structure maintained by the Activity Scheduler for the handling of the queued requests is called the Ready-To-Run queue (RTR_Q), and is shown in Figure 4-11. The fields of the structure are described as follows:

- **Ent_Cnt** (Entries_Count): the current number of entries in the RTR_Q, comprised of those still pending and those already serviced but not yet reported to the Processes Manager. Used to speed up the processing of the queue by keeping track of its tail.
- **Pend_Cnt** (Pending_Count): the current number of outstanding activity execution requests that have yet to be serviced. If no requests are pending, then the foreground services of the Activity Scheduler are not needed.
- **Proc_Id** (Process_Id): identifier of the process issuing this activity request.
- **Act_Id** (Activity_Id): identifier of the activity being requested.¹
- **In_List**: identifier of the IO_List (of process Proc_Id) to be used when getting the input data for the activity (Act_Id).

¹ Since parallel threads of a process may issue requests for the same activity, the **Proc_Id** and **Act_Id** fields may not uniquely identify a request in the queue. This will only be a problem under the following conditions: 1) the activity is executable by more than one slave, 2) the requests for the same activity get assigned and execute on different slaves simultaneously, and 3) the order of the completions is different than the order in which the requests were received by the Activity Scheduler. To guard against this situation, the system compiler/verifier would have to rename the conflicting activities with new Activity_Id's, and adjust the Activity Allocation Table with the new system generated entries.

- **Out_List**: identifier of the IO_List (of process Proc_Id) to be used when retrieving the output data of the activity (Act_Id).
- **Elm_Id** (Element_Id): identifier of the point in the requesting process's graph structure, from which the ACT request was issued. That location is the point from which execution will resume when the request is completed. This field is returned to the Processes Manager as the Activity Scheduler's response to the *Next_Done_Act()* service request.
- **Status**: the current state of completion of this activity request. Can be one of *Pending*, *Active*, or *Done*. A *Done* state is needed since activity completions are buffered for reporting to the Processes Manager.

Since all structures are created statically (at system compile time), and since all parallel threads of a process are executed 'concurrently', the length of the queue must be long enough to accommodate every parallel thread of every process in the system that contains an *Activity* language element. In reality there may be processes that would never run simultaneously, but currently there is no mechanism for indicating a process-level precedence relationship.

Figure 4-12: Activity Allocation Table

The eligibility of slave processors to execute specific activities is maintained by the Activity Allocation Table data structure (Figure 4-12). Here, each activity is described by the number of slaves (**Slave_Cnt**) that are capable of its execution, and their identification (**Slave_Id**). The **Duration** field was used in the simulation of *Activities*,

by simply indicating the expected execution time of the particular activity (in clock ticks). As before, since the data structure is statically declared (at system compile time), it is sized according to the total number of distinct activities in the system.

Although not implemented in this prototype run-time kernel, the ease of handling of a slave processor failure (in a centrally coordinated architecture such as Multiactivity) can be seen from the simplicity of this data structure: when a slave processor fails, all activity entries that reference the failed processor are adjusted by removing that *Slave_Id* from the list of slaves. Clearly, if there were one or more activities that were the exclusive domain of the failed slave, then a system-level error handler must be invoked.

To round out the handling of activities and slave processors, one additional data structure is needed, which must maintain the dynamic correspondence of executor (slave processor) to activity request. In addition to this function, the Slave Table (Figure 4-13) keeps track of the current *Busy* state of an executor, provides an indication of how that processor is interfaced (in this case, shared memory is assumed), and lists the activities that the slave is capable of executing. Although this last function is a duplication of the information in the Activity Allocation Table, the ordering of activities by slave is used to significantly speed up the assignment of a new activity to a slave that just signalled the completion of its previous activity i.e. at the completion interrupt level.

Figure 4-13: Slave Table

The Slave Table data structure is sized to accomodate the maximum number of slaves configured into a particular application. For testing purposes, the simulation of an activity execution on a slave processor was implemented as a simple duration which is decremented (in the background, by timer tick interrupts) for the slave while the activity is 'executed'. All fields, except **Data_Ptr** and the list of executable activities, are dynamic and can be described as follows:

- **Slave_Cnt** (Slave_Count): the number of slaves configured into this application
- **Slaves_Avail** (Slaves_Available): number of executors that are not *Busy*. Used to speed up queue service by stopping when all slaves have become busy.
- **Proc_Id** (Process_Id): identifier of the process that requested the activity being executed by this slave
- **Act_Id** (Activity_Id): identifier of the activity currently being executed by this slave
- **Data_Ptr** (Data_Pointer): pointer to the shared memory area dedicated to the interface of this slave processor (assuming a shared memory architecture)
- **Status**: current execution state of this Slave: one of *Busy*, *Not_Busy*, or *Failed*.

In servicing a slave completion, the **Proc_Id** and **Act_Id** fields are used together as a backward link into the Ready_To_Run queue to find and update the entry that corresponds to the activity that has just been completed.

Handling of the Request Queue

As was mentioned earlier, an activity may be assigned to a particular executor at one of three possible times: 1) when the request is received, 2) when the slave signals completion, and 3) when the RTR_Q is serviced by the foreground of the Activity Scheduler. The algorithm below is used in the foreground servicing of the queue. It represents the most complex case, where all possible assignments of activities to slaves are made.


```

Service_Que_Foreground()
{
    while Slaves_Avail and Pending requests and not all
        requests have been attempted yet
    {
        disable interrupts                ;;;;;;
        find next PENDING entry in Queue  ;;;;;;
        mark it as ACTIVE                  ;;;;;;
        enable interrupts
        get Act_Id, Proc_Id and In_List fields of entry
        if there is a Slave that can execute Act_Id then
            get Data_Ptr of that slave
            DATA_FROM_PROC(Proc_Id, In_List, Data_Ptr);
            ACTIVATE_SLAVE(Slave_Id, Act_Id);
            decrement Slaves_Avail and Pending_Count
            mark the slave as BUSY in the SLAVE_TBL
            transfer Proc_Id and Act_Id to Slave record
        else
            mark Queue entry as still PENDING
        endif
    }
    endwhile
}

```

The other two cases are simpler variations of the same algorithm: 1) for the case of assignment upon request, the queue is first tested for any *Pending* requests before assignment proceeds or aborts (if there are), and 2) for the case of assignment upon slave completion, the RTR_Q is searched only for activities that are executable by that slave.

Note that checking of an individual queue entry for serviceability is protected from the background interrupts by first marking the entry as Active, which is in turn, done as a critical code section (as marked in the code fragment above).

4.4.2 Activity Scheduler Services

Two services are provided by the Activity Scheduler to the Processes Manager: *Act()* for submitting service requests of Activity language elements, and *Next_Done_Act()* for retrieving blockage point information about requests that were previously signalled as complete. The way in which these services are used is best described by reviewing the interaction of the Activity Scheduler with the Processes Manager:

- When the Processes Manager encounters an ACT language element in the graph structure of an application process, a request for *Act()* service is issued to the Activity Scheduler.
- The Activity Scheduler responds by trying to immediately assign the requested activity to an eligible slave, or by queueing the request if there are already some queued. If the assignment is successful, then the activity's input data is fetched from its parent process by issuing a *Data_From_Proc()* request to the Processes Manager.
- At the occurrence of a slave completion interrupt, the Activity Scheduler moves the activity's output data (if there is any) to the private data area of the process that had issued the activity request, and signals that process of the completion of its request. The data movement and signalling are by way of the Processes Manager's *Data_To_Proc()* and *Signal_Proc()* services.
- On the next pass of the Processes Manager, unblocked processes are checked for new ACT completions. When one is found, a *Next_Done_Act()* request is issued to the Activity Scheduler, which responds by returning the point of blockage and by clearing the completed request from its queue.
- On the next pass of the Activity Scheduler (foreground entry point), as many queued requests as possible are assigned to executors.

4.5 Message Handler

Serving as the communications interface between application processes that cooperate by way of message passing, the Message Handler is the executive process responsible for:

2 of/de 2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

- responding to message transmit (TXM) and message receive (WTM) requests from the Processes Manager
- maintaining the request queues in a first-come-first-served order
- managing the movement of data and synchronization-only messages into and out of the channel buffers
- keeping track of completions and reporting them to the Processes Manager.

Designed as two code modules, one for each of the **Foreground**, and **Service** entry points, all of the work done by the Message Handler is at the foreground level, requiring no protection from asynchronous interrupt-driven events (since the executive processes run in a non-preemptive cyclic manner). Table 4-4 shows the grouping of the function subroutines into the two code modules.

In the definition of the PAL language, the interprocess communications primitives *Send_Message* (TXM) and *Wait_For_Message* (WTM) are potentially blocking. Since available buffer space cannot be guaranteed for absolutely all message transmit requests, some will be blocked until the space becomes available. A similar situation occurs in the processing of message receive requests, where a request that can be fulfilled immediately (because the message had already been transmitted and is available in the channel) is non-blocking, otherwise the request is blocking.

```

FOREGROUND:  Msg_hand()
                Service_Rx_Waits()
                Service_Tx_Waits()
                Put_Msg()
                Get_Msg()
                Purge_Buf()

SERVICES:    Txm()
                Wtm()
                Next_Done_Txm()
                Next_Done_Wtm()

```

Table 4-4: Modules of the Message Handler

Recall that in multiactivity systems, determinism in inter-process messaging is ensured by defining message communications on any one channel as one way only between any one single thread of the sending process and any one single thread of the receiving process. As well, the messaging primitives are defined as operators on channels, with no explicit reference made to the cooperating process and indeed no explicit information is kept that relates the processes of corresponding send and receive requests. The one assumption made in the design of the Message Handler, then, is that it is safe to process all transmit requests without checking the execution state of the target receiving process. Since cooperating processes must be closely associated (triggered together and comprising a single response to an event), this is a reasonable assumption. Recall also that the design of a multiactivity application is submitted to a *PAL Verifier* [10] that checks the temporal correctness of the design to guard against deadlock and starvation.

4.5.1 Handling of the Messaging Requests

The general approach used by the Message Handler in servicing message transmit and receive requests is very similar to that used by the other executive processes that service requests issued by the Processes Manager. As in those executive processes, the foreground portion of the Message Handler is primarily used to service any requests that are currently queued (because the resources to complete them were not available at the time of the request).

The interaction between the Message Handler and the Processes Manager, then, can be described as follows:

- When Processes Manager encounters a TXM/WTM language element in the graph structure of an application process, a request for *Txm()/Wtm()* service is issued to the Message Handler.
- If the channel buffer has room for the transmission or contains the data for the reception, then Message Handler proceeds with the request and indicates completion to the Processes Manager. Recall that the movement of message data to/from an application process's private data area is by way of the *Data_To/From_Proc()* service of the Processes Manager.
- If the request cannot be completed, then it is entered in the appropriate queue and the Processes Manager is notified of the blockage.

- On the next pass of the foreground portion of the Message Handler, all blocked requests are re-examined for channel space or data availability and if eligible, the requests are serviced. Completed requests are marked as *Done* and the parent process is signalled of the completion (via the Processes Manager *Signal_Proc()* service).
- On the next pass of the Processes Manager, unblocked processes are checked for new TXM/WTM completions. When one is found, a *Next_Done_Txm()* or *Next_Done_Wtm()* request is issued to the Message Handler, which responds by returning the point of blockage and by clearing the completed request from its queue.

The three private data structures maintained by the Message Handler in the processing of the TXM and WTM primitives, then, are the Channel Table and the transmit and receive request queues, Tx_Wait_Q and Rx_Wait_Q, respectively.

As shown in Figure 4-14, the Channel_Table maintains each of the communications channels configured into the system by way of a descriptor record and the channel's data buffer. The descriptor record contains three fields that describe the current amount of available space in the channel's data buffer (*Avail_Spc*), and the current number of pending transmit and receive requests that had to be queued at the time of the request (*Tx_Pend* and *Rx_Pend*, respectively).

Since messages are of variable size, and transmit and receive requests on any one channel are expected to have a one-to-one correspondence, the *number* of messages present in a channel is not maintained. Instead, the *Avail_Spc* field is used to determine

Figure 4-14: Channel Table

the availability of buffer space at transmit time, and the availability of a message at receive time. The penalty for this method of housekeeping is that pure synchronization messages must also be maintained by the same Avail_Spc field, which is reduced by one count to represent the transmission of such a pure synchronization message.

A channel's data buffer is sized (statically at compile time) to house a maximum of 5 messages at the maximum IO_List size of 12 data variables. In this prototype/test system, the buffer size is partly governed by the design goal of a single chip 8-bit microcontroller implementation (64KB total address space). In a less constrained implementation, the buffer would be sized according to the expected number of messages transmitted on that channel (or the number received, since the two are equal).

The two other data structures maintained by the Message Handler are the transmit and receive queues, for message requests that could not be fulfilled at the time of the request. As shown in Figure 4-15, the two structures are identical, and are indeed quite similar to the queue structures maintained by some of the other kernel executive processes. The table elements are described as follows:

Figure 4-15: Tx/Rx Request Queue

- **Ent_Cnt** (Entries_Cnt): the current number of entries in the table, whether they are still pending or already serviced. Used to speed up the processing of the queue by keeping track of its tail.

- **Pend_Cnt** (Pending_Count): the current number of outstanding message requests that are yet to be serviced. If there are no pending requests, then the foreground services of Message Handler are not needed.
- **Proc_Id** (Process_Id): identifier of the process that issued this message request.
- **Chan_Id** (Channel_Id): identifier of the channel to be used in the message transaction.
- **Var_List** (Variables_List): identifier of the IO_List (of process Proc_Id) that is used to describe the variables that make up the message (for both incoming and outgoing messages). A Var_List with a value of NULL is used to indicate that the message is pure signalling.
- **Msg_Siz** (Message_Size): an indication of the length of the message. Used at the time that the request is issued to test for transmit buffer space availability.
- **Elm_Id** (Element_Id): identifier of the point in the requesting process's graph structure, from which the TXM/WTM request was issued. That location is the point from which execution will resume when the request is completed. This field is returned to the Processes Manager as the Message Handler's response to the *Next_Done_Txm/Wtm()* service request.
- **Status**: an indication of the current completion state of this request. Can be one of *Pending* or *Done*.

One optimization in the handling of transmit requests that was not implemented is a 'write through' of transmit messages: if a process is waiting (WTM) on an empty channel, then the first message that is transmitted on that channel can be used to immediately fill the WTM request. Although the message data must still be moved into and out of the channel's data buffer (since data movement is controlled by an IO_List, which incurs one level of indirection), the 'write through' has the effect of unblocking the waiting process sooner.

Recall that in an effort to free up channel buffer space for the transmission of messages, the implemented Message Handler resolves *Wait_For_Message* requests before those for *Transmit_Message*. Without 'write through' optimization, therefore, a waiting (WTM) process may have to wait for one additional pass of the executive for its request to be fulfilled. Note, however, that since 'write through' optimization only applies to a WTM on an empty channel, the overall gain in performance may be small.

4.5.2 Message Handler Services

Four services are provided by the Message Handler to the Processes Manager: *Txm()*/*Wtm()* for submitting service requests for the PAL message passing primitives, and *Next_Done_Txm/Wtm()* for retrieving information about requests that were previously signalled as complete. To illustrate the handling of the *Wtm()* service entry point, its algorithm is listed in the following code fragment:

```

Wtm()
{
    if there are already queued requests or if the
        channel buffer is empty then
        queue the request in the Rx_Wait_Q
        increment Rx_Pend in requested channel's record
        return(BLOCKED);
    else
        GET_MSG(Proc_Id, List_Id, Msg_Siz, Chan_Id);
        update Avail_Spc in the requested channel's record
        return(DONE);
    endif
}

```

In the above algorithm, routine *Get_Msg()* is used to extract the message itself from the channel's data buffer. Recall that the data is moved into the receiving process's private data area by way of the *Data_To_Proc()* service of the Processes Manager, requested from *Get_Msg()*. As was discussed in Chapter 3, since a channel's data is maintained as a linear (rather than circular) buffer, after a message is extracted from the top of the buffer, the remaining contents of the buffer are moved up, or the previous message space is purged (in this case by routine *Purge_Buf()*, called by *Get_Msg()*).

The algorithm of the *Txm()* service entry point is exactly analogous to that of the *Wtm()* shown above, and the message body is deposited in the channel's data buffer by the equivalent *Put_Msg()* routine. Here, the Processes Manager's *Data_From_Proc()* service is used to get the message data from the application process's private data area.

Chapter 5

Performance Evaluation

The real world viability of PAL-based multiactivity systems relies on two principal factors: 1) the existence of a mature set of development tools¹, and 2) kernel run-time performance matching that of other comparable modern systems. Dealing with the latter issue, this chapter discusses the measured performance of the run-time kernel.

As was discussed in the introduction to Chapter 4, the DOS environment was chosen for the development and testing of the run-time kernel because it provided the nearest approximation to a bare-bones system: one where the kernel would be the only layer running over the hardware. In DOS, once an application is loaded and running, the only Operating System (OS) overhead is that required for real-time timekeeping (and even that is not strictly necessary in this implementation of the kernel). Occurring at a rate of 18.54Hz, and requiring approximately 0.1ms to service, the DOS overhead for timer tick handling, therefore, amounts to less than 0.2% and can be neglected.

5.1 Simulation of the Runtime Environment

Multiactivity systems are event driven and are interfaced to their environment through input events and activity executor slaves that generate the system outputs (the events may themselves be generated by slave processors that serve to gather environment information). Since the kernel executes in the DOS-based Central Coordinator, and slave processors do not physically exist, system environmental *events* and execution of activities on *slave* processors had to be simulated.

¹ The feasibility of PAL-based development tools is explored in the theses by Chubukian [5], Harvey [10] and San Martin [17].

With both event and slave simulations are functions of time, and with DOS having no independent hardware timers available for application programs, there was little choice but to chain onto the system timer tick interrupt and simulate the events and slaves from an Interrupt Service Routine (ISR). Event *occurrences* were simulated by using a bounded random number generator to create the Event_Id and the time to wait until the next *occurrence* of an event. Since the kernel performance was being measured, rather than that of the overall system, the execution of activities on slave processors was simulated by simply decrementing the remaining time (number of ticks) of the current activity running on a particular slave (see also Figure 4-13: Slave Table).

Clearly, slave execution simulated in this manner can neither perform useful concurrent work nor generate meaningful activity output data, nor make use of the activity input data. Recall, however, that the *movement* of activity input and output data to and from the application processes' private Data Areas is actually done at run-time as it would be on a real multiprocessor. The measured performance of the kernel, therefore, is not affected by the simulation of slave execution. Further, since no useful work is done by the simulated activities, it was deemed to be of no advantage to test the system with a heterogeneous set of slaves (except for the purpose of debugging the job allocation algorithm of the Activity Scheduler). The applications submitted for the performance tests, therefore, were configured with a set of homogeneous slave processors: all slaves capable of executing all activities. The aspects of overall system performance on a heterogeneous set of slaves was explored by R. SanMartin in his thesis on the *SIMPAL* simulator [17].

Although the simulation per-se required only the generation of two random numbers and several integer decrement operations, the responses to the simulated events and slave completions are also handled at the interrupt level. Event service consists of invoking and possibly queuing the event's response(s) and the handling the WTE requests for that event. Slave completion service consists of the removal of the activity's output data and the handling of the RTR queue in an attempt to assign a new activity to that slave. As well, the background handling of timer tick interrupts is used by the Environment Interface in the servicing of requests of the DLY primitive.

The principal problem with the timer-tick based simulation, then, is the poor temporal distribution of kernel execution - all of it concentrated about the start of the tick interval. Not only was the interrupt level processing heavy, but upon return from the

interrupt, the foreground level processing had to contend with updating the system state after the flurry of changes brought about by the events and slave completions that just occurred in the background interrupt. The measured performance, therefore, represents a very conservative estimate of the kernel's behaviour.

5.2 Sample Applications

As was discussed in Chapters 3 and 4, to facilitate testing, the kernel was implemented as much as possible in an application independent way. For the test phase, then, several applications were configured and *compiled* by hand (the PAL editor was not yet in existence). Reflecting the multiactivity view of systems and the data structures used by the run-time kernel, each application was defined in the following terms:

- **Events:** each event in the application is defined by its IO List used to gather its data, and the identifiers of the processes that are triggered by the event (as well as the processes' IO Lists used to receive the event data)
- **Processes:** each process in the application is defined by its IO Lists (used in the movement of data), the initial values of its private Data variables, and the pseudo code defining its graph structure
- **Activities:** all activities in the system defined by a single activity duration (specified in ticks)
- **Slave Processors:** the total number of activity executors configured into the application.

Since slave executors were simulated as a homogeneous set and all activities were executable by all slaves, the length of an activity was inconsequential to the kernel's performance. For ease of testing, therefore, all activities were assumed to be of the same duration and were submitted as a command line parameter. The system events and processes were defined in simple text files, to allow the configuration information to be entered with any text editor. Note that the number of communications channels used by an application is not a constraining factor, since all required channels must exist for proper operation. Recall that the (assumed) shared-memory architecture was implemented with a single dummy buffer, whose address was used in all the source and destination locations that required movement of data from a shared area.

A total of four different processes were *compiled* (Figures 5-1a to 5-1d), two containing mostly serial execution and two containing mostly parallel. Applications were then configured by using various combinations of the four processes.

Figure 5-1a: Serial Process 1

Figure 5-1b: Serial Process 2

Figure 5-1c: Parallel Process 1

Figure 5-1d: Parallel Process 2

Care had to be taken in the hand *compilation* of the processes' Descriptor Tables in the following areas:

- the correctness of the forward and backward links defining the hierarchy of nested constructs
- the correlation of messaging channels among communicating processes
- the coordination of IO_Lists used in data movement of parallel threads, such that variables referenced in one thread are not referenced in the other.

As an example of the graph structure portion of the information describing a process, the following text shows the Descriptor Table pseudocode for the serial process #1 of Figure 5-1a. The space delimited fields of each language element correspond directly to those defined in Section 4.3.2 (the Activity structure is shown for reference):

```

SEQ 3    0 0           ;1
  CON 2    9 1         ;2
    SEQ 4    8 2       ;3
      ACT 1 2 3 3      ;4
      ACT 2 3 4 3      ;5
      RDS 5 3 3        ;6
      RDE 2 3 3        ;7
      ACT 3 2 1 2      ;8
    SEQ 3    13 1       ;9
      TXM 0 1 9         ;10
      WTM 1 1 9         ;11
      ACT 10 2 3 9      ;12
    CAS 3 5 0 1        ;13
      SEQ 3    18 13    ;14
        ACT 4 3 2 14    ;15
        TXM 0 1 14      ;16
        ACT 5 4 3 14    ;17
      ACT 2 3 4 13      ;18
      ACT 1 2 3 13      ;19

```

Activity	ACT	Id	In_List	Out_List	Back_Lnk
----------	-----	----	---------	----------	----------

A total of six applications were configured and tested, four small and two medium sized. The small configurations consisted of 4 processes and 46 activities (not all distinct), and were designed to be mostly serial or mostly parallel i.e. using only serial processes, or only parallel processes, respectively. The medium sized configurations consisted of 8 processes and 92 activities and were designed to have a combination of serial and parallel execution by using both serial and parallel processes. All applications were configured with two copies of each process, each copy appearing as separate process Id, so as to double the number of processes without having to design more applications.

Half of the applications were configured with interprocess communications, and half were configured without. For those that did contain communications, cooperating processes were configured to be triggered by the same event, and message primitives were carefully compiled by hand to ensure that messaging over the set of channels was temporally correct and consistent with the rules of multiactivity systems (concurrent threads must use separate channels). Applications were configured both with and without inter-process communications in order to compare and verify their expected behaviour: cooperating processes will be delayed by their temporal dependency on each other. The six configured applications, then, can be listed as follows:

- 4 processes, mostly serial execution, no inter-process communications
- 4 processes, mostly serial execution, with inter-process communications
- 4 processes, mostly parallel execution, no inter-process communications
- 4 processes, mostly parallel execution, with inter-process communications
- 8 processes, combined serial and parallel, with inter-process communications
- 8 processes, combined serial and parallel, no inter-process communications.

5.3 Measuring the Performance

As Heisenberg discovered, the very act of measuring a system perturbs its operation so as to alter its behaviour. This could be especially true when a program tries to measure its own real-time performance, unless it can be ascertained that the measurement aspects constitute a negligible (<1%) amount of overhead. As discussed below, this is the case for the run-time kernel.

The strategy used to minimize the measurement was very straightforward: keep the run-time accumulation of performance data as simple as possible, and leave all the more complex statistical derivations for the end of the run. As it turns out, all run-time data gathering for performance measurement was accomplished by long-integer increment operations (except for one data item that needed a long-integer addition). Although some statistics were based on foreground level operations most were functions of time and, hence, were based on the background operations (DOS timer tick). The data gathered by the system can be summarized as follows:

- **Elapsed_Ticks**: the duration of the run, measured in timer ticks
- **Passes_of_Executive**: the total number of times around the outermost loop of the kernel
- **Proc_Occur**: the number of times that a process was invoked during the run (measured for each process in the application)
- **Proc_Ticks**: the total length of time (in ticks) that a process was *Active* or *Blocked* during the run
- **Rtr_Q_Hist**: histogram of the Activity Scheduler's Ready-To-Run Queue in terms of requests still pending after it has been serviced in the foreground
- **Slave_Usage**: the total length of time (in ticks) that a slave was busy (measured for each slave in the application)
- **Total_Events**: the sum of all occurrences of all events during the run
- **Evt_Occur**: the total number of occurrences of this particular event
- **Evt_Qued**: the number of times that this particular event had to be queued
- **Evt_Missed**: the number of times that this event was missed altogether because of queue overflow
- **Evt_Q_Sum**: the sum of this event's queue length, as accumulated every timer tick, such that the average queue length is $\text{Evt_Q_Sum} / \text{Elapsed_Ticks}$.

It is the last item in the above list that represents a long-integer addition operation - all of the remaining data items are gathered at run-time as long-integer increment operations. Considering the worst case measurement load when all 8 processes are *Active*, all 9 slaves (maximum configuration) are *Busy*, and all 4 events are queued, the total number of data collection operations is calculated to be 23 as follows:

Foreground. Approximately two increments per pass: one for *Passes_of_Executive*, one for *Rtr_Q_Hist*, and every so often one for a process starting up (*Proc_Occur*).

Background. Approximately 21 increments per tick: one for *Elapsed_Ticks*, eight for *Proc_Ticks*, nine for *Slave_Usage*, and three for all event statistics.²

If the worst case measurement overhead is 23 long-integer increment operations, then a conservative average overhead can be calculated by assuming that 75% of all processes are active all the time, and that 50% of all slaves are active all the time. The average measurement overhead, therefore, is 16 long-integer increment operations (same values as before but with 75% of *Proc_Ticks*, and 50% of *Slave_Usage*). Since the average execution path of a single pass of the run-time kernel (combined foreground and background) is greater than the equivalent of 1600 long-integer increment operations, the average performance measurement overhead is clearly <1%.

So far, the configured applications submitted for testing have been described only in terms of their processes and their use of communications. Two other parameters were varied in the series of tests that were used to collect the performance data:

- the number of slave processors available to the application
- the arrival rate of events, in ticks between event occurrences, specified as a minimum and a maximum used to bound a generated random number.

Although the duration of activities was specifiable as well, this parameter was little explored as it was found that an activity duration greater than 1 tick produced negligible system loading. Finally, a length of run of 15 minutes was chosen as a good compromise between a long duration that is statistically meaningful but requires too long to test, and a short duration that may not represent a good sampling procedure. The statistical reliability of the 15 minute run was confirmed by an even distribution of event occurrences i.e. each of the events configured into the system occurred approximately the same number of times.

² For any single test run, the period between occurrences of a given event was generated as a random number bounded by an upper and lower limit. From run to run, those limits were varied and statistics gathered. Since in this case, the application response time was predetermined by the longest execution path in a process's graph structure, the minimum period between events was constrained by that response time. Assuming activity execution time of a single tick, a minimum period of 8 ticks between events was chosen to generate events at a rate just faster than the best response time. This per-event rate corresponds to an aggregate rate of event arrivals of one event every 2 ticks for a system with 4 event sources.

A total of 144 configurations were tested (each for a 15 minute run), comprising all combinations of slaves, event frequencies, and application processes:

- 6 configured application (see Section 5.2)
- 3 different event arrival rates: 8-12 ticks, 10-14 ticks, and 12-16 ticks
- 8 slave configurations: minimum of 2 to a maximum of 9.

Performance Criteria

Although there is no formal set of characteristics that defines the performance of real time executives, there are some criteria that are generally accepted as relevant. As well, in establishing the viability of a PAL-based multiactivity (multiprocessor) system, a measure of the bottleneck brought about by the centrally coordinated architecture is a critical parameter. The following list describes the issues that were investigated in the performance of the run-time kernel:

- The **overhead** incurred in the execution of an application process. This was determined by comparing the longest execution path in a process's graph structure (measured in terms of 1 tick activities) with the measured average execution time of that process (a function of the number of occurrences of the process and the measured time that it was active during the test run).
- Event **response** delay. This measure of the time taken by a system to initiate a response to an event is frequently used in the performance analysis of real-time executives. Other event handling measurements in the run-time kernel included the proportion of event occurrences that were queued and missed. Clearly, when the number of missed events becomes large, the response delay parameter becomes meaningless.
- In a centralized architecture, such as the one underlying the run-time kernel, the effectiveness of the system relies on the fact that the Central Coordinator itself is always available to handle its duties and never becomes a **bottleneck**. The critical question, then, is at what point *does* the Central Coordinator become a bottleneck?

5.4 Performance Results

When all the statistical performance data had been compiled and analyzed, several facts became apparent. First, as previously mentioned, it was readily observed that using an activity duration greater than one tick produced negligible system loading, creating a system that was little influenced by its configuration - a minimum of 1 tick was used thereafter (an activity length of zero renders most measurements meaningless, since the temporal behaviour of the system is based on the tick interval). Second, for a designed application to make sense, the rate of arrival of events has to correspond to the average execution time of the application processes, which varies as the activity execution time. The relationship of event arrival rate to average process execution time must hold for any implemented system, simulated or real.

As shown in Figure 5-2, very low loading was also observed for the smaller configured applications (4 processes), allowing the system to excel in all measured performance criteria. For this reason, the remaining discussion of this chapter centres on the medium sized configurations that were more thoroughly tested (those with 8 processes).

Figure 5-2: 4 Process Configuration

In the results illustrated by Figures 5-3 to 5-6, all cases represent tests performed with the medium sized applications, using an activity duration of 1 tick. The first of each pair of diagrams represents the application configured with inter-process communications, and the second configured without. Where an application process contains communications primitives, it has the same graph structure as the equivalent process that does not perform any communications, but some of the *Activity* primitives have been changed to messaging. In all figures the Z axis has no numerical significance, and is merely used for visual enhancement.

System Overhead

If the Central Coordinator in a multiactivity system were (computationally) infinitely powerful, then the functions that it performs in traversing the graph structures of the application processes would take no time. Ideally, then, the time for a process to execute, would simply be the duration of the activity execution times in the longest path of that process. In reality, the functions performed by the Central Coordinator, such as graph traversal, job scheduling, and activity and message data movement, all take finite amounts of time. The sum of those times expended in 'executing' a given process from start to completion is the system overhead.

Determining a process's longest execution path is simple for graph structures containing only *Sequence* and *Concurrence* constructs. Since the execution of the *Case* and *Repeat* constructs depends on run-time data, determining the path length in processes containing these elements is more complex. In the configured application processes, however, in order to make the system testable and predictable, all data handled by the *Case* and *Repeat* constructs was hand tailored, and hence, the path lengths were known.

The actual process 'execution' time was determined by measuring the number of times that the process was invoked during the test run, and by measuring the amount of time (in ticks) that the process was *Active* or *Blocked* over the duration of the run. The overhead for any single process, then, was the ratio of measured to ideal execution times. The overheads shown in Figures 5-3a/5-3b represent averages taken over all processes in the configured application. From this performance data, several observations can be made.

Figure 5-3a: System Overhead, w/ Communications

Figure 5-3b: System Overhead, no Communications

In both configurations (with and without communications), the overheads incurred when the event arrival rate is 8-12 ticks is much higher than that of the others. This is because at the highest arrival rates within the range (8 and 9 ticks), the longest execution path of some of the processes is actually longer than the time between events. The 8-12 tick range, then, creates a (over)load condition which the system has some difficulty in servicing, thereby exhibiting the larger overhead.

A feature common to the performance curves of all six configurations is the asymptotic plateau that is reached as slave processors are added to the system. This is an indication of an application whose processing requirements have been met, and which simply cannot make use of any more concurrent resources. As activity executors are added beyond this saturation point, the system performance remains constant (since most concurrent activity requests are already being serviced), but the average utilization of the slave processors will decrease.

The main difference between the configurations that contain inter-process communications and those that don't is the overall level of incurred execution delay. It is higher for communicating processes because of the dependence within process pairs: the effective execution time of a process is longer when it has to wait for a message from its cooperating partner. Although the execution time is longer in these cases, the delay does not necessarily represent an increased overhead (recall that for processes with equivalent graph structures, the communication is added by removing what would otherwise be activities).

Two conclusions can be made about the system's behaviour (these conclusions will also be valid for the remaining performance criteria):

- the true system overhead must be measured at the plateau region, since a system configured with fewer resources is waiting for those resources rather than incurring overhead
- only the plateau of the lower event arrival rate is a meaningful measurement, since the other arrival rates represent systems that are relatively underpowered i.e. the rate of event arrivals overruns the response execution times.

Response Delay Time

From the time that an event causes an interrupt in the Environment Interface to the time that the response to that event is invoked and actually 'executing', the Central Coordinator computer must perform a variety of actions. Functions associated with servicing an event include the fetching and distribution of the event's data (to the Environment Interface data area and to the processes starting up), the signalling of the process(es) constituting the event's response, and the queueing of the event occurrence and its data if its response cannot be started immediately. Recall that an event may initially be captured by a slave, which then passes it on to the Central Coordinator. Since the delay incurred by a slave processor in its event servicing is not included in this discussion, the response delay figures may be somewhat optimistic.

Since the response delay time, as defined above, is typically less than the smallest time unit kept by the system, an indirect method had to be devised to assess this performance parameter. As shown earlier, four run-time statistics were gathered about each configured event:

- the number of occurrences of that event
- the number of times that the event had to be queued
- the number of times that the event was missed because its event queue was already full
- a running average of the event's queue length.

The first statistic was used to simply confirm that length of the test run was statistically meaningful by way of an even distribution of occurrences of the system's set of events. The second and third statistics were used to qualify the overall performance of the system - if the number of queued and missed events was very large, then all remaining measurements were meaningless, and the system was not functioning properly. The fourth statistic was used to (indirectly) determine the response delay time. By sampling an event's queue length at the occurrence of every timer tick and accumulating it into a running sum, the average queue length of that event was then calculated at the end of the test run by simply dividing the sum by the run duration (in ticks). The response delay times shown in Figures 5-4a/5-4b represent the average taken over all events. Several observations can be made from this performance data.

Response Startup Delay

Figure 5-4a: Event Response, w/ Communications

Response Startup Delay

Event Queue Latency

Figure 5-4b: Event Response, no Communications

A feature common to all configurations (both with and without communications) is that response time improves with the amount of resources for concurrent processing (slaves). As more of the activities requested by the processes are able to execute in parallel, the smaller the delay in the execution of the processes themselves. As processes run to completion with fewer delays, the greater the chance that they will be ready to be invoked as responses to events, and hence the shorter the response delay time. This plateau effect is similar to that seen in the overhead measurements and has the same significance here as well.

Also common to all configurations is the observation that at the highest event arrival rate, the response time does not go down to near-zero, even at the maximum number of slave processors (the response times for the lower event frequencies do). This is consistent with the previously described relationship of process execution time and event arrival rate, where at the very highest rate of event arrivals (8-9 ticks), the longest path of some of the application processes is longer than the time between events.

The main difference between the configurations that contain inter-process communications and those that don't, is the overall degradation of the response time in those that do. The reasoning here is similar to that used in the discussion of system overhead: the delays incurred by processes waiting for each other when using message passing result in longer execution times for event responses (cooperating processes). The longer the execution time, the lower the availability of those responses at the time of the event occurrences.

Central Coordinator Bottleneck

As there is no direct and clear means of determining the point at which the Central Coordinator can be said to be acting as a processing bottleneck in the system, an indirect method had to be used to assess the kernel's effectiveness in keeping up with the performance requirements of the configured application. Since the kernel has been implemented as a cyclic executive, the most natural way of measuring its load handling capability was to determine how many cycles, or passes it could execute per unit time.

The total number of passes executed during the test run was easily determined by simply incrementing a cycle counter with every iteration of the outermost control loop

Foreground Capacity

Figure 5-5a: C.C. Bottleneck, w/ Communications

Foreground Capacity

Figure 5-5b: C.C. Bottleneck, no Communications

of the foreground portion of the kernel. The number of passes per unit time, then, is the total number of passes divided by the total units of time elapsed over the duration of the run. This performance data is shown in Figures 5-5a/5-5b as Passes-Per-Tick (note the reversal of the highest and lowest event arrival rates from the scheme of the previous figures) Recall, however, that since the simulation of events and slaves was done in the background at every timer tick interrupt, the state of the overall system remained unchanged for much of the remaining tick duration. In other words, most of the work performed by the kernel's foreground processing was done on the first several passes after the occurrence of the tick mark, and the remaining passes represented mostly idle time (all blocking primitives are tick related, and therefore had to wait for next tick to be resolved). Nevertheless, the Passes-Per-Tick criterion gives a good indication of the kernel's ability to handle its load.

Several observations can be made from Figures 5-5a/5-5b. First, and most importantly, is the fact that the Passes-Per-Tick value never approaches zero, indicating that kernel is not approaching its processing limits, and hence, the Central Coordinator does not represent a processing bottleneck in this set of test applications (as implemented on a 12MHz 80286-based DOS machine).

Second is the general shape of the curves: first rising as slaves are added, then peaking at 3 or 4 slaves and then falling linearly (except the one configuration representing the heaviest system load: no inter-process communications and highest event arrival rate range). Since a lower number of cycles per unit time means a heavier processing load (both foreground and background), the only phenomenon that can explain the positive slope at the lower slave counts is the heavier queueing of events and their data that occurs when the system is not able to keep up. As slaves are added and processes run to completion in shorter times, fewer events need to be queued. Event queueing and all its related overhead, then, can be seen to be computationally expensive. The negative slope of the second part of the curve is an indication of an increased load resulting from the scheduling of a greater number of slaves. This load will further increase with the complexity of the scheduling algorithms and with the handling of a heterogenous set of slave processors.

As was seen in the other performance measurements, configurations incorporating inter-process communications behave differently than those without. Here, the overall load can be seen to be slightly greater for communicating processes, implying that

message handling is more costly than job scheduling. This is reasonable in light of the fact that the Message Handler is the only executive process that maintains significant data buffers that must be handled on a continuous basis (when communications are engaged). Although the Environment Interface also handles data buffers when queuing events, that processing is only occasional, when the system is momentarily overloaded. The other executive processes only handle queue structures, which represent a lower processing load.

Conclusions

The overall performance of the kernel can be summarized as follows:

- the kernel exhibits near-zero process execution overhead and near-zero response startup delay for sizes of applications of the same order as those that were tested (8 processes, 96 activities and an activity granularity greater than the average length of a pass through the kernel)
- given these results, it is reasonable to expect that a modern high-performance microprocessor could handle all but the most demanding applications
- the low overhead of the kernel means that performance results obtained for a particular application are a good indication of the level of resources required for that application.

5.5 Performance Byproducts

In multiprocessor systems, a measure of how effectively the system's computational resources are used is a term referred to as **slave processor utilization**. Although this figure is application dependent and varies with the homogeneity of the slaves and the scheduling algorithm, it does provide an indirect measure of the effectiveness of the system's job scheduling. The utilization of any single slave was calculated as the time that the slave was *Busy* divided by the total duration of the test run. In the prototyping-based PAL design environment, the slave processor utilization provides a primary indication of the number of activity executors that need to be configured into a particular application.

Slave Processor Usage

Avg. for all Slaves

Figure 5-6a: Slave Utilization, w/ Communication

Slave Processor Usage

Avg. of all Slaves

Figure 5-6b: Slave Utilization, no Communication

Slave Processor Utilization

It is intuitively clear that as activity executors (slave processors) are added to the configuration of a system, each slave will have to perform proportionately less work. Although this is true for the average utilization, it does not apply to the individual units unless the job scheduling algorithm is designed with the specific aim of achieving an even load distribution across the slave processors. An alternate way of achieving an even load distribution is to modify the allocation of activities that the individual slaves of a homogenous set can execute (this may have to be done empirically). Since the implemented kernel assigns activities to (the homogenous set of) slaves in a simple sequential order, the resulting load distribution more closely resembles a decaying exponential, with the 'first' slaves in line being the most heavily loaded. The slave utilizations curves of Figures 5-6a/5-6b represent averages taken over all the slaves in the system.

Of all the system performance parameters, slave utilization was the easiest to measure directly and accurately. By defining utilization to be the proportion of time that the slave was engaged in activity execution, a slave's *Busy* time was simply sampled and incremented at the occurrence of every tick. The utilization figure, then, was the ratio of the total *Busy* time to total run duration (both in ticks). Note that the state of a slave was sampled after all potential activities had been assigned to all slaves that completed at that time. This was done to ensure that the brief transitions between assignments were not part of the utilization data.

It can be observed that in four out of the six cases both slaves of a two unit configuration are 100% busy - a good indication that the system is overloaded and that the lack of concurrent resources is causing delays (this was also confirmed by the other performance data, where the system was similarly overloaded).

Note that the general shape of the utilization curves closely resembles that of the system overhead. This is consistent with the fact that as processing resources are added to the system, process execution times decrease as more concurrent requests can be fulfilled, and the average slave usage decreases as the total load is spread over more units. Note also that utilization is generally greater for a system configured without inter-process communications. This is caused by the fact that communicating processes were created by replacing some of their activity requests with the messaging primitives.

Chapter 6

Multiactivity Applications

When considering application areas that may be implemented as multiactivity systems, the primary issues of concern are those of performance and suitability of approach. As was shown in the performance analysis of the preceding chapter, the kernel running on a 286-based central coordinator produced near-zero overhead and near-zero response delays for relatively complex (properly configured) applications. The suitability of using multiactivity in a particular application depends on the degree of correlation between the characteristics of the application and those of multiactivity systems. Many applications can be viewed as multiactivity systems, yielding solutions in a variety of areas. In general, multiactivity systems can be described as having the following set of characteristics:

- central and/or hierarchical organization and coordination
- event driven interface
- separation of execution from system coordination
- inherent, explicit handling of multi-thread concurrency.

An event driven system is specified by defining the set of responses comprising the system, which in multiactivity systems are expressed directly with the language elements of PAL. To properly map event responses onto a multitasking system, on the other hand, a designer must group the activities into tasks, and must have knowledge of the task and message management services of the specific OS being used. To estimate the performance of the multitasking-based design, knowledge would also be required of the context switch and message handling overheads of the particular OS.

Although this thesis has focused on the application of multiactivity systems to real-time multiprocessor environments, other application areas, where time is not a correctness criterion, may also be well served by this design methodology. With a prominent feature of multiactivity systems being the specification of control flow, application areas well

suited to PAL-based development will be those dealing with or closely related to the flow of control. This chapter explores two broad areas that seem to be particularly well suited to multiactivity implementation: dual layer systems with both simple and complex activity executors, and hierarchically organized (multi-layer) systems.

6.1 Dual Layer Systems

Since the primary premise of multiactivity systems is the separation of coordination and execution, particularly well suited will be applications where activity executors can easily map onto physical entities and where the centralized coordination will not introduce an unacceptable overhead or bottleneck into the system. Multiprocessor implementations of embedded controllers satisfy several needs: increased processing power, true concurrency, the handling of multiple specialized processors, and greater reliability through modularity and reconfigurability.

The term 'controller' can encompass a wide variety of applications, covering a large range of performance and capability requirements, both real-time and not. Systems may range from single chip, very high speed, very high data throughput applications to large and slow chemical / industrial process control applications where temperature loops (for example) may take days to stabilize. The feature common to most of these *dedicated* applications is the specification of the flow of control, requiring system level coordination.

As an example of a dual layered system with simple activity executors and a high performance requirement, an early application study proposed that the multiactivity approach can be used in the design of VLSI microprogrammed devices [13]. In such 'Segmented Microprogram' machines, the activity executors became the microprocessor's functional blocks (ALU, registers, Bus Interface Unit, etc.) and the activities themselves corresponded to the distinct phases of machine instruction execution. Being an activity executor, each of the functional blocks represented a segment of the overall processor, and as such was driven by its own microprogram. As with multimicroprocessor systems, it was observed that the separation of control and execution resulted in a system that was more easily modified by virtue of the greater encapsulation of functions particular to the processor segments.

Representing a dual layered application but with intelligent activity executors, multiuser information systems have been explored from the multiactivity point of view [6]. This is a class of non-real-time dedicated systems in which the activity executors are general purpose computers (such as PCs) running application programs that interact with a central coordinator. In one particular example, a hospital Electrocardiogram (ECG) management system is considered, in which the purpose of the overall application is to efficiently process incoming patients given a finite number of ECG units, analysis stations (cardiologist) and data entry stations (admissions nurse). Although this LAN-based ECG management system is centrally coordinated for control flow, the common serial bus interconnection scheme is used to transfer data directly among the slaves. Specifically, the high volume of ECG data is directly moved from the (ECG) acquisition units to the analysis stations. The central coordinator automatically takes those data dependencies into account by the precedence relations expressed in the description of its application processes.

6.2 Hierarchical Multi-Layer Systems

Perhaps the most natural mapping of the multiactivity approach is to Flexible Manufacturing Systems or FMS [11], most of which have a hierarchical organization, are centrally coordinated, and require little data movement among the various intelligent components. The characteristic of these systems that most directly correlates with the multiactivity approach is the separation of activity execution from system coordination. PAL-based development in this application area is equivalent to an exercise in system integration, where the interaction between the programmable work cells is specified.

The hierarchical nature of FMSs is composed of multiple levels (Figure 6-1). At the highest level, the overall system is made up of programmable work cells (flexible machining cells, FMCs, or flexible assembly cells, FACs), which represent activity executors controlled by the system's Central Coordinator. At the next lower level, the elements of the work cells (robot arms, numerically controlled machines, vision systems, etc.) become the activity executors, providing such services as milling, assembly and transport of the product parts. The lowest level in the system hierarchy represents the intelligent sensors and actuators that implement the physical interface to the machinery.

Figure 6-1: Hierarchical Multi-Layer Systems

The inherent parallelism in a coordinated multi-machine manufacturing system is a direct result of its multi-level hierarchical nature and can be directly expressed in PAL. At the highest level, each of the programmable work cells comprising the FMS assembly line can be operating on a different component of any of the products being manufactured - the arrival of the component at the work cell would be the event triggering the response (process) to operate on it. At the next lower level, multiple machines of a work cell can be running in tandem, moving milling and/or assembling the part at the same time - these would be the activities of higher level processes, and at the same time represent the process level responses of this lower level. At the lowest levels, multiple actuators of a machine (or multiple axes of a single actuator) can be in concurrent motion - these represent the activities of the processes one level higher. In other words, the manufacturing process maps directly onto functional decomposition, which is directly supported by the PAL language.

Error handling and fault tolerance are important features of an FMS, where the mechanical nature of the devices provides limited accuracy and inevitably results in occasional failures. A variety of mechanical problems can be detected by sensors and simply returned as part of the output data of activities i.e. the PAL language and the

run-time kernel support the movement and handling of error information. Faults may be handled in one of three ways: kept local to the activity, returned as output data by the activity, or issued as an event to one (or more) higher level of coordinator. Fault tolerance in a mechanical environment, however, is primarily constrained by the capability of the machines to do each other's jobs. This is true at several levels in the hierarchy. The failure of a machine tool component in a work cell, for example, may or may not be replaceable with another tool of the same work cell. If the manufacturing capability of the overall work cell is available elsewhere, then that information can be readily updated in the system controller. However, some fault conditions will still require human intervention to clear the error.

The most attractive feature of developing an FMS using the PAL-based approach, is the visualization of the physical system afforded by the close coupling of the design steps to the shop floor machines and processes (the discussion below relates to the top most level in the control hierarchy):

- the machining and assembly steps of the product(s) to be manufactured are specified as PAL expressions. The expressions can be refined and expanded using functional decomposition at the process, subprocess and activity levels.
- the activities defining the manufacturing steps are grouped according to their particular resource requirements, resulting in activity to executor (work cell) allocations. This step provides an initial estimate of the number of work cells that will be required in the system.
- based on execution time estimates for the various activities defined above (manufacturing steps), the overall FMS operation can be simulated for all product mixes, and tested for bottlenecks, resource utilization and throughput. As well, simulation of fault conditions and their handling can be tested at this stage.

In conclusion, the effectiveness of PAL-based FMS design and development relies not only on the viability of multiactivity but also on the availability of highly standardized shop floor components. As those components become more standardized and more easily interconnected, the design of a complex FMS will come to resemble an exercise in system integration rather than global and low level design. In fact, the multiactivity approach can be used as a system integration methodology in all application areas (not just FMS).

Chapter 7

Conclusions

7.1 Thesis Summary

The design of real-time, concurrent systems is among the most complex and difficult of software engineering disciplines. These systems are typically developed using common procedural programming languages into application tasks that run over real-time multitasking executives. This kind of approach can be time consuming and expensive in both the initial implementation and long term maintenance stages. Multiactivity systems, designed using Process Activity Language (PAL), represent one method of reducing the life-cycle costs of concurrent systems. The PAL / Multiactivity approach is based on development through prototype refinement: by using an *executable* specification language, a system defined in PAL can be easily tested for conformance at the design phase. Multiactivity's separation of coordination from actual execution, allows these systems to be more easily modified and tuned as a result of the lower interdependency of the decoupled coordination and execution.

An event driven system is specified by defining the set of responses comprising the system, which are expressed directly with the language elements in PAL / Multiactivity systems. By contrast, to properly map event responses onto a multitasking system, a designer would have to group the activities into tasks, and would require knowledge of the task and message management services of the specific OS being used. To estimate the performance of the multitasking-based design, knowledge would also be required of the context switch and message handling overheads of the particular OS.

Through the design, implementation, and performance evaluation of a run-time kernel for PAL-based systems, this thesis has demonstrated the viability of the multiactivity approach in the real-time domain. As well, by implementing a kernel that is efficient, small (less than 49KB of combined code and data space), and independent of any

underlying software layers, it was shown that the multiactivity approach is highly portable and can be used on a wide variety of hardware platforms. The independence from underlying software layers was achieved by a fully static design, that handles run-time overload conditions through data queueing (rather than dynamic buffer allocation). This makes multiactivity systems particularly well suited to embedded systems, where their viability is enhanced by the kernel's ability to run on single-chip 8-bit microcontrollers. If applications require data capacities >64KB but must still be implemented with 8-bit devices, then several options exist, including vertical layering of activity executors, or horizontal layering by way of a multi-microcontroller Central Coordinator.

Designed as a set of four cooperating executive processes, the kernel embodies a graph traversal scheme that directly interprets the PAL language elements used to define the graph structure. In support of the centrally coordinated architecture of multiactivity systems, the kernel assumes a tightly coupled, shared-memory configuration for interaction with the surrounding slave processors. Since slave interaction was simulated at run-time, and hence the low level control handshaking not implemented, the system can be easily tailored to the specific requirements of a particular hardware configuration.

Testing of the kernel's real-time performance on a 12MHz 80286 DOS-based machine demonstrated that a properly configured application can be executed with near zero overhead, and produce a good *relative* response time. The response time is termed *relative* because it is compared to the rate of event arrival (the system's temporal behaviour was derived entirely from the DOS timer tick rate of 18.2Hz). In properly configured applications, those whose process execution times are faster than the average period between events, the system response delay time is of the order of 1/10 of a tick. As well, from the low processing load exhibited by the kernel, it can be concluded that the central coordinator does not form a computational bottleneck in the tested applications - i.e. those representing typical industrial control applications. The potential data bottleneck of the centralized architecture of multiactivity systems was discussed in Chapter 3, where several solutions were proposed, including an increase of activity granularity and the addition of extra data paths at the points of heaviest traffic.

Some examples of application areas that may be well suited to PAL / Multiactivity implementation were suggested, including general purpose controllers and Flexible Manufacturing Systems (FMS) in which several of the inherent characteristics of FMS

mapped almost directly to those of multiactivity. These included the multi-level hierarchical organization and the central coordination of the system, as well as the separation of coordination and execution on what is an inherent multi-processor system (FMS). It was also shown that the centralized control of (multiactivity) systems can yield greater reliability by virtue of the containment of the error handling i.e only the central coordinator needs to know of and make accommodations for the failure of any of its slave processors.

7.2 Further Research

The simple test cases that were described in Chapter 5 represent a baseline of the performance achievable by a properly configured concurrent system. Because the measured performance was that of the (single CPU) central coordinator, it should be scalable according to the power of the computer on which it is implemented. With the kernel complete and functional, the next logical development phase for multiactivity systems is an implementation on a true multiprocessor. This will require the detailed definition of the interaction mechanism between the slaves and the central coordinator (hardware), and the activities and their input/output data handling (software). An automated procedure by which the slave executable images can be generated and downloaded will also have to be developed.

A variety of system aspects not explored in this thesis have since been investigated by Mr. R. SanMartin in his thesis of a PAL Simulator tool [17]. Based on the kernel routines and data structures (both coded by July, 1989), the *SIMPAL* simulator added support for and investigated the consequences of a heterogenous set of slave processors, multiple activity scheduling algorithms, and fault tolerant handling of failed activity executors.

Aspects of the kernel that have not yet been investigated are the effects of being able to *Abort*, *Suspend* and *Resume* application processes and activities. While the interruption of an activity is in explicit contradiction with its definition as an atomic entity, there may be a set of conditions where only large granularity activities may be acceptable, in which case the *Abort/Suspend* capability may be necessary to release a shared resource. A related capability that would be useful in error detection is the concept of *Timeout* on

an activity, where a fault would be declared and an error handler invoked if an activity was not completed within the timeout period.

Another aspect of the kernel that would be worthwhile exploring, is that of handling the movement of large amounts of data between activities. This issue is one of particular concern in the suitability of centrally controlled systems to those application areas (with large amounts of data to move), where the central coordinator has the potential to become the system bottleneck. Also in the area of data and its movement, the handling of compound data types such as arrays and structures would be worthwhile exploring.

Finally, there is little doubt that acceptance of the PAL / Multiactivity design methodology rests largely on the availability and capability of a mature, reliable development environment. Such an integrated environment must be comprised of the complete set of PAL-based tools such that the output of the editor can be used by the simulator and verifier, and the final system specification can be compiled and linked into a combined run-time kernel-and-application executable image (presently, none of the tools interact). For optimization and system tuning, a useful tool would be one that can determine the most effective activity size for partitioning the application processes, taking into account a variety of parameters such as activity I/O, amount of parallelism, and processing power of the central coordinator CPU. To round out the development environment, an additional tool will be needed to show the overall system-level view of events responses and their correspondence (PAL is limited to depicting only the process levels and lower).

References

- [1] R.J.A. Buhr, *Machine Charts for Visual Prototyping in System Design*, Technical Report SCE-88-2, Department of Systems and Computer Engineering, Carleton University, August 1988.
- [2] R.J.A. Buhr, *System Design with Ada*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [3] D.R. Cheriton, *The Thoth System: Multi-Process Structuring and Portability*, Elsevier Science, 1982.
- [4] J.H. Christensen, *Graphic Programming Languages and Environments for Manufacturing Applications*, The Allen Bradley Company, Highland Heights, OH, 1988.
- [5] A. Chabukjian, "Extended PAL Editor: An Example of an Interactive Real-Time System Specification Tool", *M.Eng. Report, University of Ottawa*, April 1990.
- [6] P. Cousineau, *Design of Multilayer Information Systems*, M.A. Sc. Thesis, University of Ottawa, September 1983.
- [7] DY-4 Systems Inc., *DVME-134 Single Board Computer Operations Manual V1b*, Nepean, Ontario, March 1988.
- [8] E.T. Fahti and M. Krieger, "Executive for Task-Driven Multi Microcomputer Systems", *IEEE Micro*, October 1983, pp. 23-32.
- [9] W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software - Practice and Experience*, Vol. 11, No. 5, May 1981, pp 435-466.
- [10] R.A. Harvey, *Verification of Concurrent System Specifications using Temporal Logic*, M.A.Sc. Thesis, University of Ottawa, July 1989.

- [11] R.A. Harvey, M. Krieger, J.P. Lachance, *Process Activity Language for Planning and Coordination of FMS*, MAPL 88 Proceedings of the Symposium on Manufacturing Application Languages, Winnipeg, Manitoba, June 1988, pp. 127-135.
- [12] R.C. Holt, *An Overview of TUNIS: a UNIX Look-Alike written in Concurrent Euclid*, Technical Report CSRG-140, Computer Systems Research Group, University of Toronto, April 1982.
- [13] L. Lamport, "A Simple Approach to Specifying Concurrent Systems", *Communications of the ACM*, Vol. 32, No. 1, January 1989.
- [14] M. Krieger and P.R. Piche, "Segmented Microprogramming: A Technique for Designing Special Purpose VLSI Processors", *Proceedings of the IEEE International Conference on Computer Design*, Port Chester, NY, October 1984, pp. 393-396.
- [15] P.M. Mendel, *Structure of a Portable Operating System*, Technical Report CSRG-152, Computer Systems Research Group, University of Toronto, November 1983.
- [16] Multiprocessor Toolsmiths Inc., *Unison User's Guide, Software Release 1.1*, Kanata, Ontario, November 1987.
- [17] R.E. SanMartin, *The Multiactivity Paradigm in Rapid Prototyping of Embedded Systems*, M.A.Sc. Thesis, University of Ottawa, June 1990.
- [18] P.P. Tanner et al, "A Multitasking Swithboard Approach to User Interface Management", *ACM Siggraph*, Vol. 20, No. 4, August 1986.
- [19] E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of the Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

END

03·03·92

FIN