2

Carnegie-Mellon University
Software Engineering Institute

Kernel User's Manual Version 1.0

Judy Bamberger
Tim Coddington
Robert Firth
Daniel Klein
David Stinchcomb
Roger Van Scoy

February 1989

February 1989

Kernel User's Manual
Version 1.0

Judy Bamberger
Tim Coddington
Robert Firth
Daniel Klein
David Stinchcomb
Roger Van Scoy

Distributed Ada Real-Time Kernel Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

User's Manual

CMU/SEI-89-UG-1
ESD-89-TR-15

This manual was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this manual should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This manual has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Table of Contents

List of Figures

List of Tables

1. Scope

1.1. Identification

This manual describes the models underlying the Kernel and its concept of operations, presents the primitives available to the application program, and provides a number of abstractions that may readily be built on top of Kernel primitives.

The Kernel is a body of code that implements real-time facilities which can be invoked by applications written in Ada for execution on a distributed target. The requirements for the Kernel, both behavior and performance, are provided in the *Kernel Facilities Definition*, and its design will be provided in the forthcoming *Kernel Architecture Model*. The Kernel was built at the Software Engineering Institute by the Distributed Ada Real-Time Kernel (DARK) Project.

This *Kernel User's Manual* provides the information needed by a programmer to understand and use Version 1.0 of the Kernel to support a distributed application. The host system is the MicroVAX II (under MicroVMS 4.2). The Kernel is implemented in Ada (see Section 1.4.2), using the TeleSoft Telegen2 Compiler, Release 3.22 (see Section 1.4.2), and MC68020 assembler (the OASYS XA68000 V4.12 Cross-Assembler; documentation bundled with TeleSoft documentation; see Section 1.4.2). The target system is a network of distributed MC68020 processors and peripherals.

Target dependencies (on the Ada compiler, on the 68020, and on the DARK hardware testbed configuration) are described in detail in Appendix H and in documentation that will be provided with the code.

1.2. Motivation

Ada is now being mandated for a large number of DoD development projects as the sole programming language to be used for developing software. Many of these projects are trying to build distributed real-time systems. Many project managers and contractors are anxious to support this effort, to reap the advantages of Ada, and to use the newer techniques of software engineering that Ada can support. This transition, however, has not always been smooth; some serious problems have been encountered.

1.2.1. Ada Runtime Environment

One of the most persistent and worrying problems is the suitability of the Ada runtime system, most notably the tasking features, and especially on distributed systems. There are issues concerning functionality (amply documented in [ARTEWG 86b]), customization, tool support (especially target debuggers and performance monitors); issues of inter-process communication and code distribution; and, perhaps most intractable, issues of execution-time efficiency.

One way of approaching this problem is to press for better, "more mature" Ada implementations: more optimization; user-tailorable runtime systems (as in [ARTEWG 86a]); special-purpose hardware. This is a valid route, but one that will take time, money, and experience, and many of the solutions will be compiler dependent, machine dependent, or application dependent. Many developers are still unsure even how to use the new language features of Ada, and at least one cycle of application use, performance measurement, and methodology review will be needed before users can be sure which parts of the Ada language and runtime are indeed critical.

The Kernel described by this document implements another route to a possible solution (defined at length in [KFD 88]) which is being pursued at the Software Engineering Institute (SEI). It should be a quicker and cheaper route, and hence a feasible short-term alternative.

1.2.2. Application and System Code

In conventional programming, application code (which is what has to be written to meet the user requirements) is distinguished from system code (which is obtained with the target machine, and which is intended to support applications generally). With Ada and embedded systems, these distinctions are not so clear cut. First, it has been traditional, when developing real-time systems in other programming languages, for the application programmer to write specific code down to a far lower level, including special device drivers, special message or signaling systems, and even a custom executive. There is far less general-purpose system code. Secondly, the Ada language complicates the distinction between application and system code. In older languages, almost all system functions were invoked through a simple and well-understood interface — the system call — expressed as a normal subroutine call. In Ada, however, many traditionally system-level functions are explicit in the language itself, or implied by language constructs; for example, tasking, task communication, interrupt acquisition, and error handling. In fact, the work is really done by the old familiar system code, now disguised as the *Ada runtime.*

1.2.3. Abstractions and Their Breakdown

If the user is satisfied with the Ada level of abstraction — with its view of what tasks are, what time is, and so on — then the Ada view is a simplification: the application code in fact performs system calls, but the compiler inserts them automatically as part of the implementation of language constructs.

Unfortunately, many users are dissatisfied with the Ada abstraction, and seek either finer control or access to lower-level concepts, such as semaphores, send/wait or suspend/resume primitives, and bounded delays. Under the above circumstances, the extra language features, and the hidden system calls they generate, are an active hindrance to the application programmer, and an obstruction to the work of implementation.

For example, the programmer may need a strong delay primitive — one that guarantees resumption as soon as possible after the expiration of the delay. But Ada already has a "delay" statement, with different semantics. When implementing a different delay primitive, the user risks damaging the Ada runtime behavior, since Ada assumes it has sole control of the Ada tasks and does not expect an extra routine to perform suspensions and resumptions. To implement the new delay robustly, the user has to interface with the internals of the Ada runtime, which may be very hard to do and will surely be hard to maintain. Moreover, the Ada delay statement composes naturally into timed entry calls and timed select statements. If the user wishes to do these things with the new delay statement, a substantial part of the Ada semantics must be rebuilt, and a substantial part of the runtime must be modified.

All this, of course, is a distraction from the real work — the work of implementing the application. One of the main motivators of the Kernel is the observation that many contractors using Ada are spending most of their time worrying about the Ada system level and far too little time solving the application problems, some of which are not easy.

In sum, it can be harder to build applications using Ada language features than it would be to implement the required functionality without them. But it is also undesirable for every application to reinvent specific incarnations of real-time functional abstractions.

1.2.4. Distributed Applications

A further and equally difficult problem is the issue of executing applications on a distributed target configuration. Good software development methods teach decomposition of large applications into functional units communicating through well-defined interfaces. The physical allocation of such units to individual processors in the target environment can be done in many ways, without impairing their functionality. Good design therefore requires that the specification of these functional units and interfaces be independent, as far as possible, of their physical distribution.

In a real-time system, this implies that the mechanisms by which units interact — to synchronize, communicate with, schedule one another, or alert one another — should be uniform, regardless of whether the units are sited on the same processor or at some distance across a distributed network. If the implementation language is Ada, this leads to a requirement *for distributed Ada*.

Unfortunately, nearly all current commercially-available Ada implementations do not support this requirement. They implement the real-time mechanisms of the language only on individual or isolated processors, and provide no help with communication between processors, and hence between units on different machines. This situation leads to systems where Ada tasks communicate by different mechanisms, with different style, semantics and implementations, merely because the Ada tasks are local in one case, and remote in the other. Overall, there is a substantial loss of application clarity, maintainability, reconfigurability, and conceptual economy.

1.2.5. Real-Time Requirements

This brings us to the crux of the Kernel's rationale. Users — people who have to write application code — do not want language features: they want language functionality. In Ada, much of the real-time functionality is captured in the form of special features. This may well be (the) correct solution in the long term ([Firth 87]), since by making real-time operations explicit in the language, the compiler is permitted to apply its intelligence to their optimization and verification. But in the short term, it is palpably not working: the users either cannot use, or do not know how to use, the given features to achieve the required functionality; the implementors of the language do not know how to satisfy the variety of needs of real-time applications; the vendors are unable to customize extensively validated implementations; and commercial support for distributed targets is rare, even as the need for such support is becoming endemic among application developers.

Accordingly, it is opportune to revert to the former method of providing functionality: by specific system software implemented as a set of library routines and invoked explicitly by the user. The Kernel has taken this approach.

1.2.6. Purpose and Intended Audience

The main purpose of the Distributed Ada Real-Time Kernel (DARK) Project is to demonstrate that it is possible to develop application code entirely in Ada that will have acceptable quality and real-time performance. This purpose is achieved by providing a prototype artifact — a Kernel — that implements the necessary functionality required by real-time applications, but in a manner that avoids or mitigates the efficiency and maturity problems found in current Ada runtime implementations.

This prototype embodies a tool-kit approach to real-time systems, one that allows the user to build application-specific, real-time abstractions. This prototype is not intended to solve all the problems of embedded, real-time systems, nor is it the only solution to these problems. However, it is intended to be a solution where efficiency and speed are the primary motivation and, where warranted, functionality has sometimes been limited accordingly.

The Kernel provides one solution to the problem of using Ada in distributed, real-time, embedded applications — one that can readily be accomplished in the near term. The Kernel is truly "in the spirit of Ada" — that is, it uses the Ada language features (e.g., packages, subprograms) to provide needed adjunct capabilities. This alternative returns explicit control of scheduling to the application implementor and provides a uniform communication mechanism for supporting distributed systems.

Other difficult areas, such as fault tolerance and multi-level security, are not directly addressed in the Kernel definition. The primitives have been studied in light of these and other equally demanding issues, and are simple and flexible enough to accommodate future development in these areas.

The goal of the Kernel is to provide a viable paradigm of near-term support to a wide number of real-time embedded applications currently being required to use Ada for implementation. This Kernel is based on the belief that applications builders, not compiler vendors or language designers, best know the system-level behavior required for their programs; and that standardization of such behavior should be provided via a library package interface under the control of the application implementor, *not* via modifications to the Ada language. The strategy embodied in this Kernel provides that kind of support.

1.3. Document Overview

This document serves as both a traditional user's manual and as a reference manual. As such, there is information in it that is appropriate to different audiences at different times of analysis and development. The following section summarizes the organization and usage of each section of the *Kernel User's Manual*. Appendix A is bound as a separate volume, and Appendices F and I will be provided in the next version of this document.

The *Kernel User's Manual* is organized as follows:

Chapter 1 - Scope

Defines the scope of this document, describes the motivation for designing and implementing the Kernel, and introduces the overall manual organization.

Chapter 2 - Kernel Models and Restrictions

Describes the models on which the Kernel is based and delimits the scope of Kernel functionality. This information is first presented in the context of the International Standards Organization/Open System Interface (ISO/OSI) model, followed by an overview of each of the capabilities exported by the Kernel. The error model used by the Kernel is presented, as well as an enumeration of the limitations and restrictions on general Kernel functionality. This chapter should be read by all users evaluating, tailoring, or writing applications using the Kernel. The network example introduced in this chapter is used throughout the remainder of this manual.

Chapter 3 - Concept of Operations

Explains the Kernel process model in more depth. This chapter continues the description of a Kernel process—what form it takes, how to name it, how the Kernel manipulates the Kernel process environment (at a conceptual level), and the life cycle of a Kernel process. This chapter also outlines the steps required to prepare the Kernel for use (full details are given in documentation that will be provided with the code), and provides a simplified step-by-step "how to" guide for those building an application using the Kernel (e.g., what packages must be available, defining the network configuration). This chapter should be read by all users evaluating or writing applications using the Kernel.

Chapter 4 - Kernel Primitives

Provides an overview of each of the functional areas of the Kernel and describes each Kernel primitive. Beginning with visible data type packages and continuing through each area of Kernel functionality, this chapter introduces all packages comprising the functional area, all names exported and referenced by the functional area, sample invocations of Kernel primitives, and conditions that would cause the Kernel primitive to block. This chapter should be read by all users evaluating or writing applications using the Kernel; it is, effectively, a reference manual for the Kernel packages.

Chapter 5 - Kernel Data Structures

Provides an overview of each of the key data structures exported by the Kernel or used by it internally. With the goal of providing insight into the working of and the resource usage by the Kernel, this chapter describes each of the key external and internal data structures, its structure, initialization and storage allocation requirements, and any constraints on usage that apply. This chapter should be read by all users evaluating or writing applications using the Kernel.

Chapter 6 - Application Evaluation

Describes the Kernel facilities to obtain real-time performance metrics to manage degradation within the distributed network. This information will be provided in the next version of this document.

Chapter 7 - Notes

Provides a list of acronyms and a glossary of terms.

Appendix A - Kernel Packages

Provides the Kernel specification and primitives. (This appendix is bound as a separate volume.)

Appendix B - Kernel Exceptions

Lists all exceptions that may be raised by execution of the Kernel.

Appendix C - Tailoring and Preparing the Kernel

Enumerates the steps required to tailor the Kernel for a specific target network or application. This appendix discusses tailoring those parameters that must be consistent across the entire network on which the Kernel is to execute and those parameters that may be unique from processor to processor. Parameters that may be tailored include those interfacing to the real-time clock, granularity of time as perceived by the application, data structure storage, and more. Each tailoring parameter is identified, and, where appropriate, analysis to determine rational settings for each is provided. The default settings for all generic parameters for the 68020 target are provided in Appendix H. Appendix C should be read by all users responsible for tailoring the Kernel for a particular target network or application.

Appendix D - Scheduling Algorithms
Provides the Kernel's scheduling algorithms. This appendix provides a description of the algorithms used by the Kernel Scheduler.

Appendix E - Building Abstractions
Provides example abstractions that can be built using the Kernel primitives. These examples include: building typed message passing, safe critical regions, cyclic and periodically scheduled processes, time-critical transactions, monitors, mutually self-scheduling processes, and a message router. The examples provided in this appendix can be used as templates for application builders who need to construct application-specific code that can be based on the paradigms herein.

Appendix F - Application Example
Provides an example Ada application that is implemented using the Kernel. This information will be provided in the next version of this document.

Appendix G - Relation to Standard Design Models
Provides an outline of the relation of the Kernel to standard design models.

Appendix H - 68020 Specifics
Enumerates all 68020-dependent data structures and tailorings. This appendix outlines the limitations placed on the Kernel by the Motorola 68020 hardware and network communication protocol used by the DARK testbed. The communication protocol and the DARK testbed are described in detail in documentation that will be provided with the code. Included in this appendix is detailed information about the structure of the Interrupt Table, the resources consumed by each Kernel primitive, the size of Kernel data structures, the default values for Kernel tailoring parameters.

Appendix I - Index
An index to the *Kernel User's Manual.* This will be provided in the next version of this document.

1.4. Applicable Documents

The following DARK Project reports describe the DARK Project and the Kernel.

1.4.1. DARK Reports

[Bamberger 88a] Bamberger, J. and Van Scoy, R.
Distributed Ada Real-Time Kernel.
In *Proceedings NAECON '88.* May, 1988.

[Bamberger 88b] Bamberger, J. and Van Scoy, R.
Returning Control to the User (where it belongs).
Position paper presented at the 2nd International Workshop on Real Time Ada Issues, Devon, UK, June 1-3 1988.

[Bamberger 88c] Bamberger, J., Colket, C., Firth, R., Klein, D., Van Scoy, R.
Kernel Facilities Definition.

Technical Report CMU/SEI-88-TR-16, ADA198933, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, July, 1988.

[Bamberger 88d] Bamberger, J., Colket, C., Firth, R., Klein, D., Van Scoy, R.
Distributed Ada Real-time Kernel.
Technical Report CMU/SEI-88-TR-17, ADA199482, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, August, 1988.

1.4.2. Other Documents

In addition, the following documents are applicable to the Kernel:

1. *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A*; American National Standards Institute, Inc., New York, NY, 1983.
2. Motorola hardware manuals:
    a. *MVME225-1/MVME225-2 1Mb/2Mb Dynamic Memory Module User's Manual*; Motorola Inc., 1988; Motorola publication: MVME225/D2.

    b. *The VMEbus Specification*; Motorola Inc., October 1985, Motorola publication: HB212/D.

    c. *MVME133A Debug Monitor User's Manual*; Motorola Inc., June 1987, Motorola publication: MVME133ABUG.

    d. *MVME133A-20 VMEmodule 32-Bit Monoboard Microcomputer User's Manual*; Motorola Inc., April 1987, Motorola publication: MVME133A/D1.

    e. *MVME945 Chassis User's Manual*; Motorola Inc., February 1988, Motorola publication: MVME945/D1.

    f. *MC68020 32-Bit Microprocessor User's Manual*; Motorola Inc., 1985, Motorola publication: MC68020UM.

    g. *M68000 Family Resident Structured Assembler Reference Manual*; Motorola Inc., April 1988, Motorola publication: M68KMASM/D11.

    h. *MC68230 Parallel Interface/Timer (PI/T)*; Motorola Inc., 1983, Motorola publication: ADI-860.

    i. *MC68881 Floating-Point Coprocessor User's Manual*; Motorola Inc., 1987, Motorola publication: MC68881UM/AD REV 1.

    j. *MC68901 Multi-Function Peripheral Data Sheet*; Motorola Inc., 1984, Motorola publication: ADI-984.
3. *MM58274 Real-Time Clock*, Logic Data Book, Volume 1; National Semiconductor Corp, 1984, IM-RRDISOMIZY.
4. *MZ8305 Quad Parallel Port Module User's Manual*; Mizar Inc., 1985, Publication number: 7101-00024-0001.
5. *Z8030 Z-BUS SCC/Z8530 SCC Serial Communications Controller*; Zilog Inc., November 1987.
6. *TeleGen2 - The TeleSoft Second Generation Ada Development System for VAX/VMS to Embedded MC680X0 Targets User Guide*;.TeleSoft, 1988.
7. *OASYS User's Manual; Motorola 68000/10/20+68881 Cross-Assembler Development System*; OASYS, 1987.
8. Digital VAX/VMS manuals:

a. *VAX DEC/Test Manager User/Reference Manual*; Digital Equipment Corp, December 1985, Order number: AI-Z330B-TE.

b. *VAX Language-Sensitive Editor User's Guide*; Digital Equipment Corp, July 1985, Order number: AA-FY24A-TE.

c. *Developing Ada Programs on VAX/VMS*; Digital Equipment Corp, February 1985, Order number: AA-EF86A-TE.

d. *Guide to Using DCL and Command Procedures on VAX/VMS*; Digital Equipment Corp, September 1984, AA-Y501A-TE.

e. *MicroVMS User's Manual*; Digital Equipment Corp, April 1986, Order numbers: QLN55-GZ, Part 1 and Part 2, AI-FW62B-TN, Part 1, AI-FW63B-TN, Part 2.

f. *VAX DEC/MMS User;s Guide*; Digital Equipment Corp, August 1984, Order number: AA-P119B-TE.

g. *User's Introduction to VAX DEC/CMS*; Digital Equipment Corp, November 1984, Order number: AA-L371B-TE.

Other references:

[ALRM 83] American National Standards Institute, Inc.
Reference Manual for the Ada Programming Language.
Technical Report ANSI/MIL-STD 1815A-1983, ANSI, New York, NY, 1983.

[ARTEWG 86a] Ada Runtime Environment Working Group.
A Catalog of Interface Features and Options for the Ada Runtime Environment.
Technical Report Release 1.1, SIGAda, November, 1986.
Version 2.1 (dated December 1987) is also available, but not addressed by this document.

[ARTEWG 86b] Ada Runtime Environment Working Group.
A White Paper on Ada Runtime Environment Research and Development.
Technical Report, SIGAda, November, 1986.

[Dykstra 65] Dykstra, E. W.
Cooperating Sequential Processes.
In Genuys, F. (editor), *Programming Languages.*
Academic Press, 1965.

[Firth 87] Firth, R.
A Pragmatic Approach to Ada Insertion.
In *Proceedings of the International Workshop on Real-Time Ada Issues*, pages 24-26. May, 1987.

[KFD 88] Bamberger, J., Colket, C., Firth, R., Klein, D., Van Scoy, R.
Kernel Facilities Definition.
Technical Report CMU/SEI-88-TR-16, ADA198933, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, July, 1988.

[Rosenblum 87] Rosenblum, D.S.
An Efficient Communications Kernel for Distributed Ada Runtime

Tasking Supervisors.
Ada Letters, VII(2):102-117, March/April, 1987.

[Tanenbaum 81] Tanenbaum, S.
Network Protocols.
Computing Surveys, 13:453-489, 1981.

[Zimmermann 80] Zimmermann, H.
OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection.
IEEE Transactions on Communications, COM-28:425-432, 1980.

2. Kernel Models, Concepts and Restrictions

This chapter presents a set of models, concepts and restrictions on which the Kernel is based.

2.1. Definitions and Models

Two definitions are key to understanding Kernel models:

- *Ada task:* An Ada language construct that represents an object of concurrent execution managed by the Ada run-time environment (RTE) supplied as part of a compiler (under the rules specified in the *Ada Language Reference Manual* (LRM), see [ALRM 83].
- *Kernel process:* An object of concurrent execution managed by the Kernel outside the knowledge and control of the Ada RTE.

The preceding terminology is deliberately different from that of Ada. This is for two reasons:

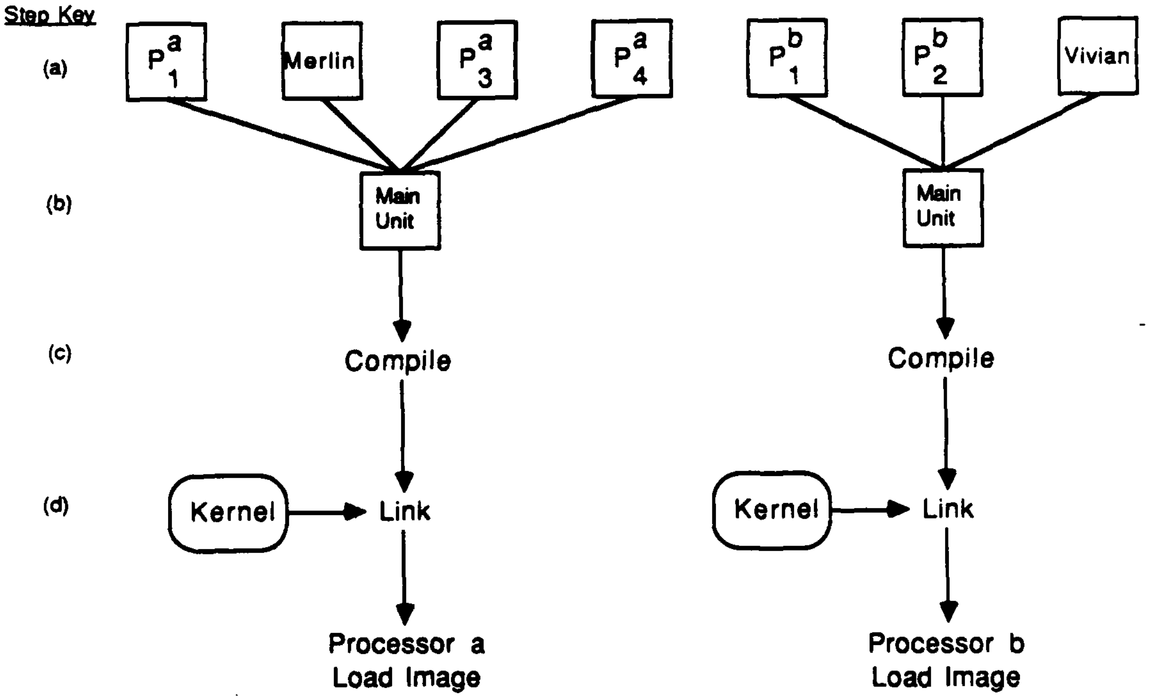
1. To remind the application developer to think not in Ada terms, but rather in the terms used by the Kernel.
2. To avoid the implication that the Kernel implements any specific function in a way that resembles an existing Ada feature with that function.

2.1.1. Process Model

The Kernel presents to the application the abstraction of a process; that is, a concurrent thread of execution. Elaborating on this general concept yields the Kernel's general process model:

1. Each process executes a unit of code, developed as a functional unit.
2. For each processor, the software engineer performs the following steps (illustrated in Figure 2-1):
    a. Develop the process code.
    b. Develop the Main Unit for the processor; the function of the Main Unit is explained later.
    c. Compile the code of the processes and the Main Unit.
    d. Link the Kernel, Main Unit, and processes together to form the load image for that processor.
3. The load image begins execution at the initialization point of the Kernel, which in turn invokes the Main Unit.

When developing a process, the software engineer need not know where the other processes will be located — on a single processor or across multiple processors. The Kernel-supplied communication primitives can be used for all inter-process communication, local or remote, with the Kernel optimizing the local case. The load image begins execution in the Ada Main Unit (after Kernel initialization). This Main Unit declares, creates, and schedules the processes in turn, and then declares that process creation is completed. After that, the Main Unit is descheduled while the processes continue to run independently.



Key

P^q_i : Process #i running on processor q.

Main Unit: The Ada Main Unit running on the processor.

Merlin and Vivian are named for use in examples.

Figure 2-1: Load Image Creation

The Main Unit is responsible for configuring the processor to meet the requirements of the application. This must include:

1. Participating in the network initialization protocol.
2. Declaring all remote communication partners.
3. Declaring and creating all locally executing processes.

There are several optional activities that may be performed by the Main Unit, including:

1. Allocating non-Kernel devices to processes.
2. Reading time-of-day clock (which is required for the Main Unit of the Master Processor).
3. Reporting system initialization failures to the external world.
4. Binding interrupt handlers.
5. Performing any system-dependent initializations (devices, buses, etc.).

In general, the Main Unit is the application entity that is responsible for configuring a processor in the manner needed by the application.

Given that there is one load image for each processor, which creates a set of processes as part of its initialization, the issue of multiprogramming is moot. The application really comprises thc set of processes; the Main Unit exists only to ensure that all the processes get linked together and started.

2.1.2. System Model

In light of the process model discussed previously, consideration must be given to the environment in which these collections of processes are executed. This requires stepping back from the "process-in-the-small" issues and considering some system-level or "process-in-the-large" issues. The system model on which the Kernel is based is shown in Figure 2-2. This view illustrates all the Kernel assumes about the target system:

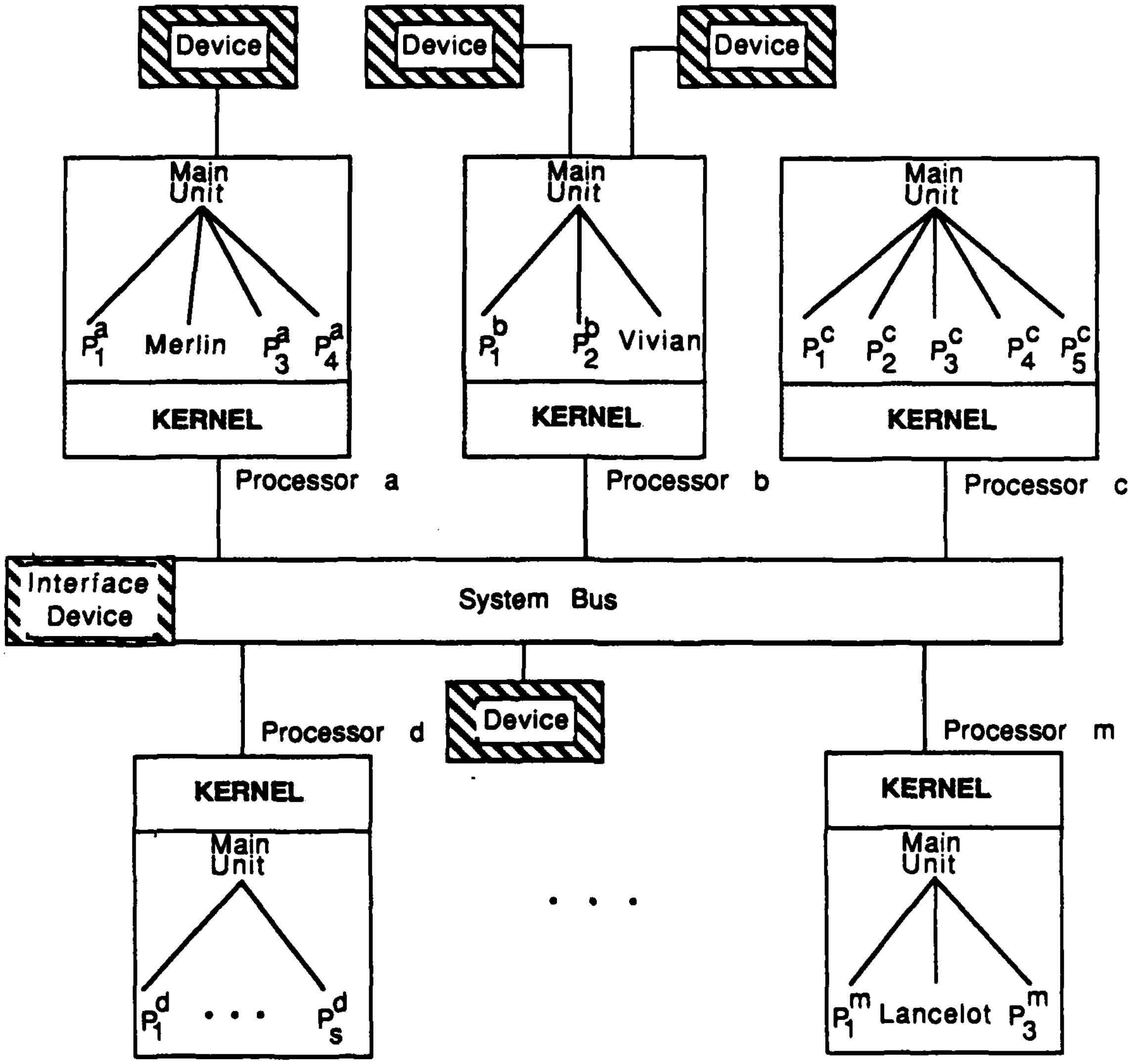
- Three types of hardware objects in the network:
    1. Kernel processors
    2. Non-Kernel processors or devices (attached to the system bus)
    3. Devices that may interrupt a processor
- No shared memory assumed (or excluded).
- No mass storage devices assumed (or excluded).
- Kernel alone interfaces directly to the system bus.

Given this model, the Kernel considers that:

> *The application comprises n Kernel processes formed into m Ada programs (load images) running on m processors.*

This requires that:

1. The user has a process or tool that allows for the static distribution of the *m* images over the *m* processors in the configuration.



Key

P_i^q : Process #i running on processor q.

Main Unit: The Ada Main Unit running on the processor.

Merlin, Vivian, and Lancelot are named for use in examples.

Figure 2-2: System View

2. The application developer has a process or tool to download the images into processor memory.
3. The application developer has a mechanism to commence execution of the loaded programs.
4. The application developer has tools to manipulate all needed disk/tape/bulk memory accesses (if these are available in the embedded configuration).

2.2. ISO-to-Kernel Mapping

The Kernel communication model presents a set of primitives to the application, and implements those primitives on an underlying set of distributed processors connected by data paths. The model, the implementation, and the intended mode of use, can all be related to the International Standards Organization (ISO) Reference Model (see [Zimmermann 80] and [Tanenbaum 81]), which provide a conceptual framework for organizing the Kernel primitives, as shown in Figure 2-3. The ISO Reference Model identifies seven layers, named, from lowest to highest:

1. Physical
2. Data Link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

The target hardware provides Layer 1. The Kernel implements Layers 2 to 4, and therefore presents to the application the Transport layer. The Kernel thus encapsulates within itself the Data Link and Network layers, rendering them invisible to the application. The application code can implement Layers 5 to 7, in part by using other Kernel primitives.

2.2.1. Physical Layer

The Physical layer is represented by the hardware data paths, which support the transmission of a serial bitstream between processors. These hardware data paths are used by the Kernel in a *packet switching* mode; that is, a sequence of bits—a *frame*—is sent at the discretion of the originator, with no implied reservation of resources or preservation of state between frames.

2.2.2. Data Link Layer

This is the layer at which basic error detection and recovery and flow control may be provided. The Kernel uses a simple *datagram* model, in which a frame is transmitted with no acknowledgment, no error correction, and no flow control. Minimal error detection is achieved by using a datagram checksum,[1] but any recovery is performed by application code, i.e., above the Transport layer. Similarly, datagram storage overflow is recognized and reported by the Transport layer.

[1] Null in the current implementation.

	Layer	Kernel Equivalent
7	Application	Created by user
6	Presentation	Created by user
5	Session	Kernel primitives: declare process, create process, allocate device receiver, and initialization complete
4	Transport	Kernel primitives: send message, send message and wait, and receive message
3	Network	Null (can be built by user)
2	Data Link	Datagram model
1	Physical	Built by using Kernel primitives: network configuration table, initialize master processor, and initialize subordinate processor

Figure 2-3: ISO Model to Kernel Mapping

2.2.3. Network Layer

Currently, the Kernel has a null Network layer. The Kernel assumes that point-to-point communication is available between any pair of nodes (processors). Routing is accomplished trivially in the sender by dispatching a point-to-point datagram directly to the receiver; no alternative routing is provided.

However, since the abstraction presented to the application is above this layer, a real Network layer could subsequently be added without requiring any application code to be changed.

2.2.4. Transport Layer

The Kernel builds the Transport layer by performing physical network connections and subsequent logical-to-physical mappings, actions that together implement the abstraction of direct process-to-process communication by means of messages.

The physical network is described by a Network Configuration Table (shown in Figure 2-4), a copy of which is maintained in each processor. This table is created by the application developer and is communicated to the Kernel during application initialization. Once that information is provided, the Kernel verifies the network connectivity and opens the physical connections between processors.

Subsequently, the logical *processes* and their physical *sites* are communicated to the Kernel. The model on which the Kernel is based assumes that all processes are created at initialization time, that a process never moves, and that a process once dead is never restarted. The Kernel therefore computes the logical-to-physical mapping once only and never subsequently changes it. Attempts to communicate with dead processes are treated as transport errors.

The Transport layer also performs the conversions between *messages* and the underlying datagrams. Currently, this is done trivially by using one datagram per message or per acknowledgment, and if necessary by restricting the maximum message size accordingly.

The Transport layer is the layer visible to the application. It supports unacknowledged send operations and end-to-end acknowledged send operations. All errors detected in this or any lower layer are reported at this layer, in the form of status codes returned by the Kernel primitives.

2.2.5. Session Layer

This layer is implemented by application code. Since it establishes logical connections between processes, its presence is required, and the application developer must write specific code to create it. This code is part of the application initialization code; it must be present on every processor and, in Ada terms, must be part of the Main Unit on that processor.

The model is one of a set of logical processes, each with an application-defined *name* and each with a single *message port* for the reception of messages from other processes.

The Kernel primitive *declare_process* indicates an intent to create or communicate with a given named process. It establishes the mapping between application-level process names and Kernel internal names.

The Kernel primitive *create_process* creates the process, establishes its message port, and makes that port available to the network. Thereafter, one process may communicate with another.

2.2.6. Presentation Layer

In the Kernel model, the Presentation layer performs no transformation of data. Rather, it performs the translation between Ada values — values of application-defined data types — and message values. This is done by application code. The purpose of the Presentation layer is to establish above the Transport layer the strong typing of the Ada language, by ensuring that communicating processes pass only strongly typed data and do so by referencing a common set of data conversion routines bound to a common Ada data type.

2.2.7. Application Layer

This layer uses the Presentation layer for whatever purpose the code requires. The model here is of parallel independent threads of control executing Ada code, identifying each other by application-level symbolic names, and communicating by passing values of Ada data types.

2.3. Processor Management

There are two steps to using the system model shown in Figure 2-2. Note that the initialization of the system topology has been deliberately kept simple. This facilitates the development of the Kernel, keeps the initialization interfaces simple and allows the users of the Kernel to develop more readily their own system-specific initialization software. First, the physical topology of the system must be defined; secondly, the system must be initialized. The approach taken to achieve the first step requires that the application engineer first define the network configuration in a manner that the Kernel understands. This is done using the Network Configuration Table (NCT) shown in Figure 2-4.

Logical Name	Physical Address	Kernel Device	Needed To Run	Allocated Process ID	Initialization Order	Initialization Complete

Figure 2-4: Sample Network Configuration Table (NCT)

This table provides the minimum information needed by the Kernel to perform system initialization and its inter-process communication functions. It is supplied by the application to the Kernel; it is implementation and hardware dependent and is available to the application for implementation of higher levels of network integrity. For each device accessible over the network, this table defines the following information:

- Logical name: Logical (string-valued) name for the device.
- Physical address: Hardware-specific information needed to access the device over the system bus.
- Kernel device: Identifies those devices that are able to respond to messages. It is possible to communicate with non-Kernel devices, but they are not expected to participate in the network initialization protocol or to understand the Kernel's datagram. Non-Kernel devices place the burden of initialization and message formatting upon the application. That is, the Kernel routes messages to and receives messages from non-Kernel devices, but it is the responsibility of the application to format and unformat these messages.
- Needed to run: Identifies those devices that must be available at initialization time in order for the application to begin execution. This could be used to mark failed or spare devices at startup.
- Allocated process ID: Identifies the recipient of all messages that originate from a non-Kernel device. This approach requires that the non-Kernel device be able to route the message to the appropriate node.
- Initialization order: Identifies the order in which the Kernel nodes of the network are to be initialized. The default, unless specifically overridden, is for the nodes to initialize in the order in which their entries occur in the NCT.
- Initialization complete: Identifies those Kernel nodes whose initialization sequence has successfully terminated.

To achieve the second step, the Kernel has defined a simple initialization protocol. This protocol requires that one processor, called the Master, be in charge of the initialization process. All other processors in the network are subordinate to this processor during the Kernel's initialization process. The Master is responsible for:

- Ensuring the consistency of the NCT among all the subordinate processors.
- Issuing the "Go" message to all the subordinate processors.

Some key points about this protocol are:

- The Master processor is a single point of failure in the system. If the Master fails to initialize, however, a subordinate may attempt recovery by declaring itself Master and attempting to reinitialize the network.
- The Master assumes it has the correct and complete version of the Network Configuration Table.
- The distinction between Master and subordinate used by the Kernel is in force only during system initialization.
- *All subordinates must be running before the Master may run.*
- If any of the following problems occurs at initialization, then the network may fail to become operational:

    1. No Master processor declares itself.

2. The Master processor fails to initialize successfully.

3. More than one Master processor declares its presence.

4. The Network Configuration Tables are found to be inconsistent.

These points can be addressed by application-specific fault tolerant techniques (redundant hardware, voting schemes, etc.), which are in the domain of the application, not the Kernel.

The primitives provided by the Kernel to support this functionality are:

- *Initialize_Master_processor.* Identifies the invoking processor as responsible for network initialization.
- *Initialize_subordinate_processor.* Identifies all other processors and instructs each to wait for the go command from the Master.

2.4. Schedule Management

The scheduling paradigm used by the Kernel is a simple, prioritized, event-driven model that permits the construction of preemptive, cyclic, and non-cyclic processes. To achieve this, there are four types of events in this model:

1. Receipt of a message (synchronous event).
2. Receipt of a message acknowledgment (asynchronous event).
3. Expiration of a primitive timeout (asynchronous event).
4. Expiration of an alarm (asynchronous event).

The scheduling primitives are discussed below, and the alarm primitives are discussed later. This paradigm allows an application process to be implemented as:

- A non-cyclic process that executes until preempted by a higher-priority process.
- A set of non-cyclic processes that execute in a round-robin, time sliced manner.
- An event-driven process that blocks when trying to receive a message. It is resumed, from the point of suspension, when it is able to proceed and when the priority admits.
- An event-driven process that blocks itself for a specified period of time (or equivalently, until a specific time) and is resumed at a specific priority (this allows a "hard" delay to be implemented).
- A cyclic process that continuously executes a body of code (and that can detect frame overrun).

To support these paradigms, the following set of scheduling attributes is defined:

- Priority:
  - Every process has a priority.
  - Priorities are relative within one processor; priorities are incommensurable across processors.
  - A process may change its priority dynamically.

    - Priorities are strict and preemptive; *higher-priority processes always shut out lower-priority processes.*
    - Blocking primitives allow the caller to specify a resumption priority, which may be different from the priority at the point of invocation. The resumption priority becomes the priority of the process when it unblocks.

- Timeslice:
    - The maximum length of time a process may run before another process of the same priority is allowed to run.
    - A property of a set of processes on the same processor and all of the same priority.
    - Time slicing cannot override priority; it applies only among processes of equal priority.
    - Any process *may enable or disable time slicing* for the *entire processor.*
    - Any process may set the timeslice quantum.
    - A process may allow (or disallow) itself to be sliced by setting its preemption status (if preemptable, the process may be time sliced; if not preemptable, the process may not be preempted by another time sliced process of the same priority).

Thus, the following Scheduler rules are universally applied:

1. Scheduler order does not change spontaneously.
2. Scheduler ordering is decided by:

    a. Higher priority before lower priority

    b. Prefer a process in an error state (to one in a normal state)

    c. First-in first-out (FIFO) order otherwise

    In other words, in all Scheduler situations, where priorities are equal, a process in an error state will be resumed preferentially; otherwise, the process first to become unblocked will be resumed.
3. When two processes become unblocked simultaneously, the process that has been blocked longest is considered to become unblocked first.

These scheduling rules are simple, fast and easy to implement. This allows for a quick implementation and a clean interface to be specified. Thus, when combined with the scheduling primitives outlined below, the application developer has the capability to tailor the Scheduler to meet the application's needs (rather than tailoring the application to fit into the Scheduler's regime).

Given this scheduling regime, a process is always in one of four states:

- *Running*: A running process is executing on its processor, and it continues to run *until something happens. If interrupts are enabled,* they *occur transparently unless* they cause a change of process state. A running process ceases to run when it: dies, invokes a blocking Kernel primitive, is time sliced, is killed by another process, or is preempted by a higher-priority process. The first three are voluntary actions on the part of the process, while the last two are actions performed by the Kernel.

- *Suspended*: A suspended process is able to run, but cannot run because a process of higher or equal priority is running. A process may be resumed when the running process blocks, lowers its own priority, or is time sliced.
- *Blocked*: A blocked process is unable to run. A process may only become blocked as a result of its own actions. These blocking actions are waiting for: the arrival of a message, the arrival of a message acknowledgment, a specific duration, a specific time, or the availability of a semaphore. A process becomes unblocked when the awaited event occurs (at which time the process transitions to the suspended state). An unblocked process does not immediately resume execution; it resumes execution only when the Scheduler so decides. But, the process can affect this decision by specifying a resumption priority in the primitive invocation.
- *Dead*: A dead process is unable to run again. A process dies in one of five ways: by completing execution, an unhandled exception, an unrecoverable error, by killing itself, or by being killed by another process. Processes are not expected to die, and any subsequent attempts to interact with a dead process result in errors.

These states and the transitions between them are shown in Figure 2-5.

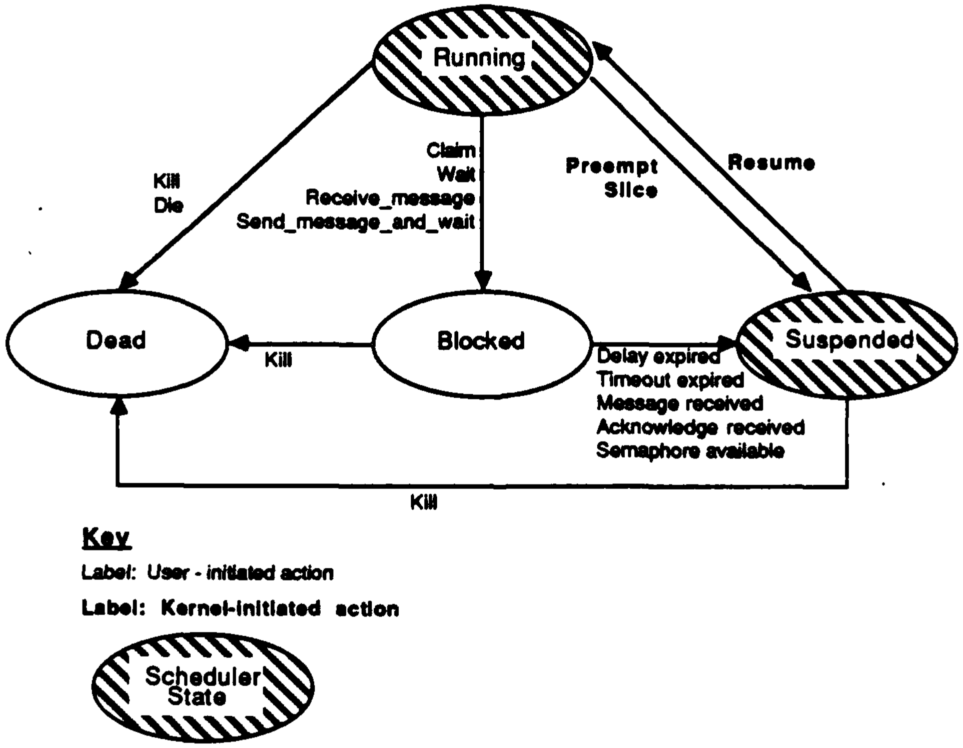


Figure 2-5: Process States

For instance, a running process becomes blocked by trying to receive when no message is pending. It becomes unblocked (but suspended) when the message arrives. It becomes running when its priority permits. A running process can also call wait, to block itself at any time. A blocked process becomes suspended and thus ready to run when its delay expires. Further, a running process may be preempted, that is, forcibly suspended by the Scheduler, to allow a higher-priority process to resume or to be time sliced.

The primitives provided by the Kernel to support this functionality are:

- *Set_process_priority*. A process may set its own priority.
- *Get_process_priority*. A process may get its own priority.
- *Set_process_preemption*. A process may set its own preemption status.
- *Get_process_preemption*. A process may get its own preemption status.
- *Wait*. The invoking process suspends itself for a specified duration or until a specified time occurs. The priority at which the process is to be resumed may also be specified.
- *Set_timeslice*. Defines the time slice quantum (only processes of equal priority are time sliced).
- *Enable_time_slicing*. Enables the Scheduler to perform round-robin, timeslice scheduling.
- *Disable_time_slicing*. Disables round-robin, timeslice scheduling.

2.5. Time Management

The concept of time permeates the entire Kernel. Many of the Kernel concepts and primitives rely on time, specifically:

- Network management uses time for initial clock synchronization and for timeout parameters in the primitives *initialize_Master_processor* and *initialize_subordinate_processor*.
- Process management requires time for the timeout parameter, in the *initialization_complete* primitive.
- Schedule management uses time for round-robin, timeslice scheduling and for delays via the *wait* primitive.
- Communication management requires time for timeout operations in the *receive_message* and *send_message_and_wait* primitives.
- Alarm management uses time for setting alarms via the *set_alarm* primitive.
- Semaphore management requires time for a timeout operation in the *claim* primitive.

To support these primitives, the Kernel contains facilities for time management, both for its own use and to make available to the application code. In all cases, two forms of delay are available to the application:

- Delay For: This computes the delay as elapsed time from the moment the primitive is called. The delay is therefore a value similar to Ada type DURATION.
- Delay Until: This delays until a specified time of resumption. The delay is therefore a value similar to Ada type TIME.

The rationale for the two types of delay is that they express fundamentally different concepts. For example, if a certain action should be performed daily at midnight, it is not correct to perform the action "every 24 hours," since successive midnights are not always 24 hours apart. Similarly, if an action should be performed every 5 minutes, it is not correct to schedule three such actions for

0155, 0200, and 0205, since 65 minutes might elapse between the second and third (i.e., the clock might have been reset).

The application programmer must be able to choose the type of delay needed. Note also that resetting the system time affects the two types of delay differently.

The current design assumes that all the target processors can use a common time base and record the passage of time at the same uniform rate. There are some real-time applications for which this assumption is unrealistic, however, since the processors will be distributed across several different inertial frames of reference, but it will serve for the current implementation.

At any moment, on any processor, the current time is given by a combination of three values:

1. Elapsed. The elapsed time is the number of ticks since the end of the application initialization process.
2. Epoch. The epoch time is a value representing the moment at which the processors began to compute elapsed time.
3. Base. The base time is the calendar date corresponding to an epoch of zero, i.e., the base of the representation of time.

The representation chosen for both epoch and elapsed is fine enough to allow accurate measurement and large enough to allow code to run for a very long time. Thus the current time of day = Base + Epoch + Elapsed.

Time is set initially on the Master processor by the application. This is done either by hand, during operator dialogue, or by reading a continuously running hardware device. The processors may then synchronize system time by having this processor use the Kernel primitive *synchronize.* This gives the application complete control over when to synchronize system time. Once the clocks are synchronized, the Kernel does not attempt to maintain the synchronization. The processors resynchronize only as a result of deliberate action by application code.

Three forms of resynchronization are supported:

1. The elapsed time for any processor can be changed by an explicit command. This is to be used when one processor's time computation has gone awry. It has the effect of changing pending delays of either kind, since increasing the number of elapsed ticks makes the machine think both that it has been running longer and that it is later in the day.
2. The epoch time of any processor may be changed. This is to be used if it is discovered that the original time setting was wrong. It has the effect of changing any pending delay-until actions, since increasing the epoch makes the machine think it is later in the day, but does not change how long it thinks it has been running.
3. The Kernel provides a primitive that explicitly synchronizes all the clocks in the network.

The primitives provided by the Kernel to support this functionality are:

- *Adjust_epoch_time.* Resets the local processor's epoch time to the specified date/time.

- *Adjust_elapsed_time*. Increases or decreases the local processor's elapsed time by the specified amount.
- *Read_clock*. Reads the current elapsed time from the local processor clock.
- *Synchronize*. Resets the clocks on all the processors in the system the time on the invoking processor.

2.6. Process Management

To use the process model, the application must have a globally unique name for each process. These names have two forms:

1. The logical name given to the process by the developer, encoded in Ada as a character string, and
2. The internal name given to the process at runtime by the Kernel.

Hereafter, the internal name of a process is called the *process ID* or *process identifier*; the term *process name* refers to the logical name of the process. However, knowing the name of a process does not guarantee the availability of the process at runtime. This is one class of faults that the Kernel is able to detect and report.

The primitives provided by the Kernel to support this functionality are:

- *Declare_process*. The Main Unit on a processor declares all locally executing processes and all remote processes and non-Kernel devices with which communication occurs.
- *Create_process*. The Main Unit on a processor creates all Kernel processes that are to execute on that processor (these may be cyclic or non-cyclic).
- *Initialization_complete*. The Main Unit indicates to the Kernel that all process declarations and creations are now complete.
- *Die*. A Kernel process may indicate that it is complete and ready to terminate normally. Once terminated, the process may not run again.
- *Kill*. A Kernel process may cause itself or another process to be abnormally terminated. This is an emergency stop operation on a process.

2.7. Semaphore Management

The Kernel provides the traditional Boolean ("Dykstra") semaphore facility [Dykstra 65], slightly modified to be consistent with the overall philosophy of the Kernel primitives.

A semaphore is an abstract data type. Objects of this type may be declared anywhere, but since semaphores are used to build process synchronization systems, they are clearly best declared in the Main Unit of a processor. A semaphore is visible only on the processor on which it is declared, and therefore can be used only by processes local to that processor.

At any time, a semaphore is in one of two states:

- FREE: The semaphore is free, or

- CLAIMED(N): The semaphore is claimed, and N processes are awaiting its release. These processes are blocked on a FIFO queue associated with the semaphore.

The primitives provided by the Kernel to support this functionality are:

- *Claim.* The invoking process attempts to claim the semaphore. The claiming process blocks until the semaphore becomes available or the timeout expires.
- *Release.* The invoking process releases a previously claimed semaphore.

2.8. Communication Management

The communication model is based on the following premises (and is similar to that presented in [Rosenblum 87]):

- All communication is point-to-point.
- A sender must specify the recipient.
- A recipient gets all messages and is told the sender of each.
- A recipient cannot ask to receive only from specific senders.
- Messages do not have priorities.

The purpose of a message is to convey information between processes. To the Kernel, a message is just a sequence of uninterpreted bits. The Kernel provides the untyped primitives; the application developers may build above them whatever application-specific functionality is needed. Communication between processes on a single processor is optimized.

Figure 2-6 illustrates this communication model. In this figure, process Merlin on *Processor a* sends a message to process Vivian on *Processor b*. This is accomplished by Merlin informing the Kernel of the message content and the logical destination of the message (i.e., Vivian). The Kernel on *Processor a* takes this message, formats the datagram to hold the message, and transmits the datagram over the network to *Processor b*, where it knows Vivian resides. When the message arrives at *Processor b*, the Kernel there rebuilds the message from the datagram and queues it for Vivian until Vivian requests the next message. If Merlin had wanted acknowledgment of message receipt by Vivian, the Kernel on *Processor b* would have formatted an acknowledgment datagram and sent it back to *Processor a* after Vivian had asked for (and received) the message.

The primitives provided by the Kernel to support this functionality are:

- *Send_message.* Sends a message from one process to another, without waiting for acknowledgment of message receipt.
- *Send_message_and_wait.* Sends a message from one process to another, and the sender blocks while waiting for acknowledgment of message receipt or until an optional timeout expires.
- *Receive_message.* Receives a message from another process, blocking until a message is available or an optional timeout expires. The Kernel automatically performs any required acknowledgments.

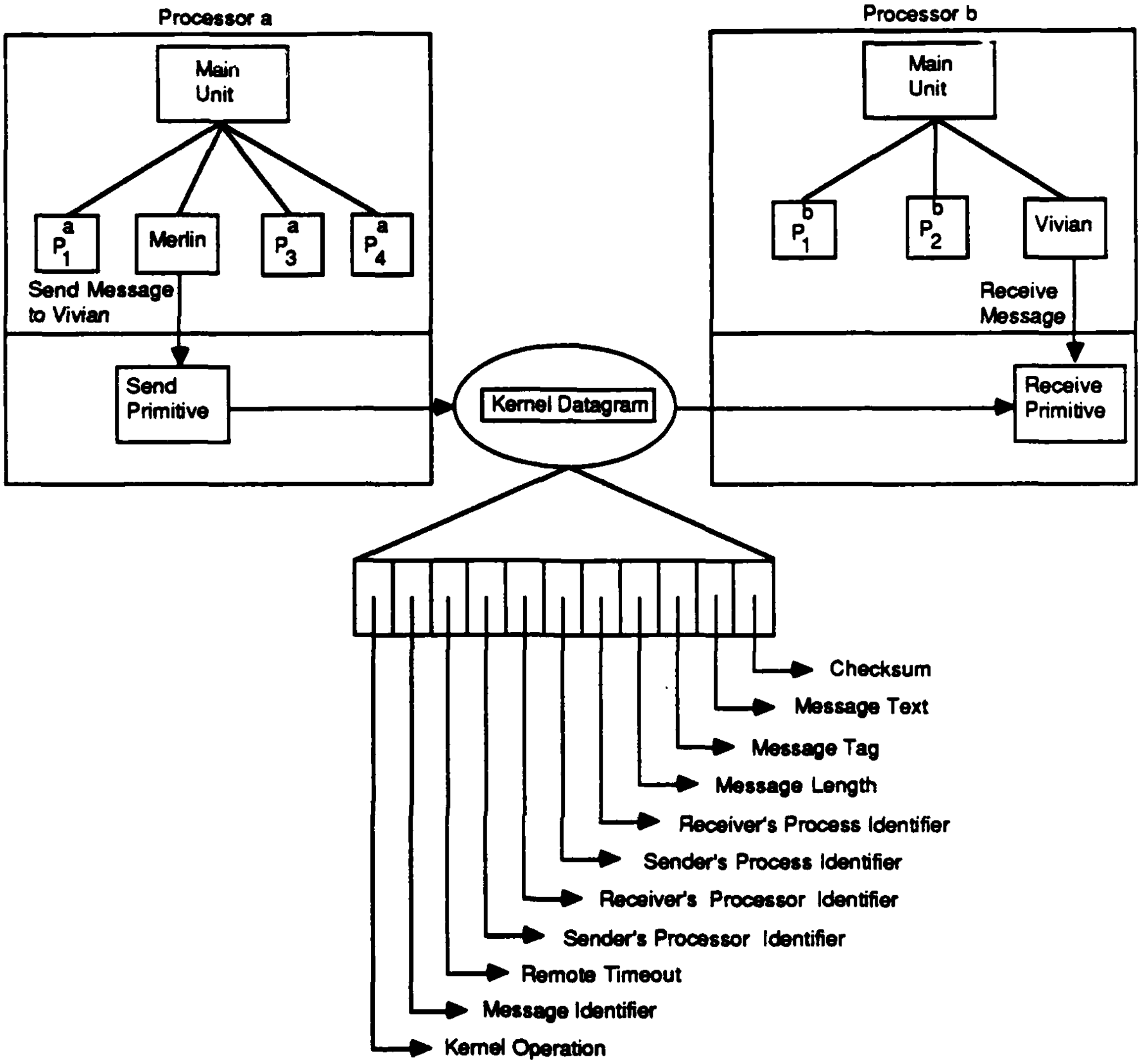


Figure 2-6: Datagram Network Model

- *Allocate_device_receiver.* Identifies a Kernel process to receive all messages from a non-Kernel device.

2.9. Interrupt Management

This section outlines the interrupt control primitives provided by the Kernel. There are two parts to the Kernel's view of interrupts: interrupts themselves and interrupt handlers. The interrupt model used by the Kernel is based on the following premises:

- There are devices that can interrupt the processor.
- There are three classes of interrupts:
  1. Those reserved by the hardware and the Ada runtime environment (*divide-by-zero, floating-point overflow, etc.*).

2. Those reserved by the Kernel (such as the clock interrupt).

3. Those available to the application (everything not in 1 and 2 above).

All the primitives described below apply only to the third class of interrupts.

- The device interrupt may be either enabled or disabled. If the interrupt is disabled, the device cannot interrupt, regardless of how badly it might want to.
- The Kernel does not queue interrupts nor does it hide hardware-level interrupt properties, such as queueing of interrupts, interrupt priorities, or non-maskable interrupts.
- Interrupts are events local to a processor and cannot be directly handled or bound by processes running on a different processor.

The model used for interrupt handlers is:

- An interrupt handler is an Ada procedure with no parameters (or some other code unit following the Ada procedure-call conventions of the target compiler).
- Interrupt handler code can access procedure local or processor global memory.
- Interrupt handler code has access to all the Kernel primitives; the only restriction is that a handler is not allowed to block its own execution.
- If an interrupt is enabled and a handler is bound, then the occurrence of the interrupt transfers control to the bound handler, which is code the application developer has supplied.

The primitives provided by the Kernel to support this functionality are:

- *Enable.* Allows processing for a specific interrupt to occur.
- *Disable.* Disallows processing for a specific interrupt from occurring.
- *Enabled.* Queries whether a specific interrupt is enabled or disabled.
- *Simulate_interrupt.* Simulates the occurrence of a specific interrupt in software.
- *Bind_interrupt_handler.* Asserts that an Ada procedure has been identified as an interrupt handler and is to be executed when the specified interrupt occurs.

2.10. Alarm Management

Alarms are:

- Enforced changes in process state.
- Caused by the expiration of a timeout.
- Asynchronous events that are allocated on a per-process basis (each process may have no more than one alarm).

Processes view alarms as a possible change in priority with an enforced transfer of control to an exception handler. Alarms are requested to expire at some specified time in the future. When an alarm expires, the Kernel raises the **alarm_expired** exception, which the process is expected to handle as appropriate. If a zero or negative duration or an absolute time in the past is specified, the alarm expires immediately. Alarms are intended for use in the construction of cyclical processes.

The primitives provided by the Kernel to support this functionality are:

- *Set_alarm.* Sets a timer to expire after the specified duration or at the specified time.
- *Cancel_alarm.* Cancels an unexpired alarm.

2.11. Tool Interface

The Kernel is a utility intended to support the building of distributed Ada applications. As such, it is important that the Kernel work in harmony with user-developed support tools. To provide that support, the Kernel must provide a window into its internal workings. It is envisioned that a tool is simply another Kernel process executing on one or more of the processors in the network. As such, the tool has access to all the Kernel primitives. Using the Kernel primitives along with the tool interface primitives, tools to monitor system activities (such as process performance, network performance, processor performance or message throughput) can be built.

Given the above motivation for the tool interface, the actual form of the interface is driven by the following concepts:

- The tool needs easy access to all the information of the Kernel without expending system resources to do so.
- The *extraction of information* based on what the Kernel knows is left to the tool (and indeed, it is deemed to be the function of a tool).
- The internal Kernel information must be provided in a manner that does not compromise the integrity of the Kernel; this implies a read-only access to the Kernel's internal data structures.
- The performance impact of using the tool interface must be predictable and bounded.
- An application should *never* have to be modified simply to use a tool (while this may not always be possible, it is nevertheless a desirable goal).

In general there are two classes of Kernel information that may be of interest to a tool: process information and interrupt information. The primitives defined below describe the information available and the mechanisms provided to access this information.

- *Begin_collection.* Begins logging state change information for the specified process. The data are logged via a message formatted by the Kernel and delivered to the tool process.
- *End_collection.* Terminates the logging of state change information for the specified process.
- *Read_process_table.* Copies the Kernel's process table into application memory.
- *Read_interrupt_table.* Copies the Kernel's interrupt table into application memory.

2.12. Error Model

All Kernel specifications document those assumptions, preconditions, and postconditions that hold for all Kernel primitives. This section describes the context in which that documentation is to be interpreted.

The Kernel provides the capability to selectively enable and disable error processing. This provides an application with the capability to detect and handle only those errors that cannot be asserted by the application never to occur once the application has become operational. This capability is also described in this section.

2.12.1. Assumptions

There are certain conditions for which the Kernel does not explicitly check but assumes hold true prior to any invocation of a Kernel primitive. These *assumptions* must be guaranteed valid by the application prior to any Kernel call. Examples include:

- The Kernel assumes that the Network Configuration Table is never modified by the application.
- The Kernel assumes that a process identifier has been created via a correct and legal invocation of the Kernel primitive *declare_process*.
- If error checking for *illegal_context_for_call* is disabled, the application is asserting, and the Kernel assumes, that no Kernel primitive will be called in an illegal context. If this assumption is violated, the Kernel protects itself, but does not guarantee correct or sensible execution from the point of view of the application.

2.12.2. Preconditions

There are certain conditions that the Kernel checks upon invocation. These *preconditions*, once validated, are assumed to remain valid during the processing of the Kernel primitive. Should a precondition not be satisfied (i.e., the validity test fails), then an error condition exists. If the respective error checking is enabled, the application is notified via a Kernel exception. If error checking is not enabled, the Kernel takes action to protect itself but then returns to the application program; no exception is raised. The following examples constitute error conditions:

- Precondition is false on call; there is an error on Kernel invocation. For example: the calling unit of the Kernel primitive *declare_process* is not the Ada Main Unit.
- Precondition is asynchronously invalidated before call terminates; an asynchronous problem arises. For example: a timeout on a call to the Kernel primitive *send_message_and_wait* has expired.

An error of the first kind, where the precondition is false on call, always results in an immediate return to the application, without blocking. An error of the second kind, where the precondition is invalidated before completion, causes a return after some interval of time, during which the caller is blocked.

2.12.3. Postconditions

The Kernel also requires that all expected postconditions be achievable. Should a required postcondition not be satisfied, then an error condition exists. The following example constitutes an error condition:

- The postcondition cannot be established; there is a failure of the virtual machine. For example: a hardware/network failure has occurred.

A postcondition error might be capable of being detected on invocation or might be detected only after some time, and so the caller might or might not have been blocked.

2.12.4. Mechanism for Error Reporting

The Kernel reports all error conditions to the invoking process by raising an Ada exception. A list of Kernel exceptions and the Kernel primitives and actions that may cause them to be raised is provided in Appendix B.

Wherever possible, the Kernel detects errors locally on the processor running the invoking process. To do this, the Kernel relies on its local copy of information representing global or remote state. A rule of this implementation is that a local copy might lag, but cannot lead, the true remote state it represents. For example, if a local process table indicates that a remote process is dead, that process has indeed died.

2.12.5. Enabling and Disabling Error Reporting

Many of the status codes reported by the Kernel are diagnostic in nature and appropriate only for software testing and integration. Given that the Kernel is intended for use in operational real-time systems, a means is provided to disable run-time error checking and reporting by the Kernel. Figure 2-7 presents the template used throughout the Kernel to achieve the selective enabling and disabling of the error checking it performs.

For each Kernel error check that could raise an exception (label (B) in Figure 2-7), the Kernel provides a generic formal parameter (label (A)) that is used to enable or disable error checking and reporting. The default status of all error checking parameters is enabled (i.e., true): error checking and reporting is to be performed upon every invocation.[2] In the body of the subprogram where the check is performed, there are two if-statements (labels (C) and (F)); tne first for the case where error checking is enabled; the second for the case where error checking is disabled. Within each alternative, the error check is performed (labels (D) and (G)); in this case it is a simple call to a subprogram that encapsulates the check. Should the check fail and checking be enabled, the appropriate exception will be raised (label (E)). Should the check fail and checking be disabled, the Kernel will do minimal processing and will return to the invoker of the Kernel primitive (label (H)).

[2] This capability is predicated upon the compiler's ability to detect, and thus not generate code for, non-reachable Ada source code. Error-checking code that is potentially to be eliminated would be wrapped within an if-statement having the generic formal parameter as its test. Since this depends upon a compiler-dependent optimization, all lines dealing with error checking and reporting have been tagged with the sentinal: --ERROR at the end of each line; this way, the code could be "stripped" of all lines carrying this sentinel, achieving the same effect as though the compile-time optimization had been performed.

```
generic

                                                      -- (A) --
  calling_unit_not_Main_Unit_enabled : in Boolean := true;
  ...
package generic_process_managers is

  calling_unit_not_Main_Unit : exception ... ;        -- (B) --
  ...
  procedure declare_process ( ... );
end generic_process_managers;

package body generic_process_managers is

  procedure declare_process ( ... ) is

  begin
    ...

    if calling_unit_not_Main_Unit_enabled             -- (C) -- check enabled alternative
      then

      if not calling_unit_is_Main_Unit                -- (D) -- check performed
        then

        raise calling_unit_not_Main_Unit;             -- (E) -- exception raised
      end if;     -- check fails
    end if;     -- check enabled

    if not calling_unit_not_Main_Unit_enabled         -- (F) -- check NOT enabled alternative
      then

      if not calling_unit_is_Main_Unit                -- (G) -- check performed
        then

        return process_table.null_process;            -- (H) -- self-protection processing
      end if;     -- check fails
    end if;     -- check not enabled
    ...
  end declare_process;
end generic_process_managers;
```

Figure 2-7: Template for Enabling / Disabling Kernel Error Checking

It should be noted that the Kernel always performs minimal error checking to ensure its own integrity; the Kernel never endangers itself.

Should the application developer choose, error checking may be disabled. This is an assertion to the Kernel that either the specific error does not occur, or if the error does occur, the application has other means of detecting that the requested action has not been performed as it would have been under non-error conditions.

2.13. Restrictions

A number of restrictions are imposed on the form of the application code. They are presented here with justification in italics:

1. The Kernel neither implements nor supports Ada tasking semantics. *This is in keeping with the design goal of making explicit control that is now implicit. This also reflects the desire of the project team to be as compiler independent as possible. Supporting Ada tasking semantics in addition to Kernel process semantics would require a higher degree of compiler-specific integration than is desired.*
2. Inter-process communication is provided by explicit use of Kernel primitives. *This is a manifestation of the explicit operation versus the implicit operation.*
3. No Ada tasking primitives may be used by the application. *This preserves the goal to replace the implicit operations of Ada tasking with explicit operations of Kernel processes and to avoid having two runtime systems in the processor competing for control of the hardware clock.*
4. Each processor has its memory completely loaded at download time. *This is a simplifying assumption for the Kernel implementation. The Kernel operates under the restriction that all processes and all data are memory resident at all times. This does not prohibit the application from building processes that can be rolled in and out of memory.*
5. The hardware configuration is static throughout the execution of the application. *This is a simplifying assumption imposed to make the development effort of the Kernel a manageable activity. Hardware description tables are modularized so that adding dynamic hardware configuration could readily be accomplished.*
6. All Kernel processes are created statically and scheduled dynamically. *This is a simplifying assumption imposed to make the development effort of the Kernel a manageable activity. All internal data structures and subprogram calls are sufficiently modularized so that adding dynamic reconfiguration could readily be accomplished. The goal of dynamic reconfiguration was kept in the foreground during design of the Kernel to achieve this.*
7. Initialization is not a time-critical function. *This is considered to be a simplification, as system and Kernel initialization is most often hardware- and application-specific. Initialization-related information is isolated so that an application can readily replace the Kernel-provided initialization modules. For this prototype, initialization is treated as a one-time operation done at system startup and, thus, is treated as not time critical.*
8. The Kernel does not implement fault tolerance, but it does detect the presence of certain classes of faults (defined in Chapter 4). These include: failure of the network and the processor on which the application executes. *The Kernel detects certain system faults, but it leaves the recovery from these faults in the hands of the application. The Kernel provides the capability for an application to build some rudimentary degree of fault tolerance.*
9. The Kernel does not use shared memory between processors. *The Kernel's reliance on special hardware, such as shared memory, would restrict the portability of the Kernel, which is a project goal. However, there is nothing to preclude a port of the Kernel to a system that does have shared memory. This task would require reworking of more than the expected pieces of the Kernel to take full advantage of shared memory.*
10. Any Kernel process may communicate with any other Kernel process. *This*

restriction simplifies the Kernel by placing the burden of restricting communications on the system or software engineer. Management of system process names thus becomes a configuration management issue within the application.

11. The Kernel does not implement any paging or memory management facilities. *The Kernel assumes all processes on one processor execute in the same unchanging address space.*

3. Concept of Operations

This chapter provides an overview of a Kernel process—what it is, how it is named, how it is created, what actions it may perform—and presents an outline of how to build an application using Kernel processes. This section does not provide details about tailoring the Kernel for any hardware or application (see Appendix C for that information) or descriptions on design methodologies (see Appendix G).

3.1. Kernel Processes

A Kernel process is a unit of code that executes in parallel with other units of code. It can communicate with other processes (see Section 4.7), can arrange to be executed at certain times (see Section 4.8), and is otherwise under the control of the DARK Scheduler (see Appendix D).

3.1.1. Form of a Kernel Process

In Ada terms, a process is a procedure with no parameters. The process begins execution at the start of its declarative region and ceases execution if it reaches the end of its statement sequence. This means that a process that is intended to run forever must be coded with an explicit loop statement.

A Kernel process may:

- Declare local variables
- Call Ada subprograms
- Call Kernel primitives
- Reference objects declared in packages that are Ada compilation units
- Call the Ada allocator

A Kernel process *must not*:

- Reference objects declared within other Ada subprograms (the Kernel's process encapsulation cannot set up the correct access paths to such objects)
- Be anything other than an outer-level procedure
- Use the Ada tasking features (*Ada Reference Manual*, Chapter 9)

3.1.2. Setting Up a Kernel Process

There are four steps in setting up a Kernel process:

1. Writing the code as an appropriate Ada procedure, in an appropriate context (see Section 4.6.1.2).
2. Naming the process uniquely and *declaring* it to the Kernel
3. Computing the resources that are required to execute the process
4. *Creating* the process environment

These steps are explained in detail below. Note that, even after it has been created, a process is not yet executing. This is discussed in Section 3.1.3.

3.1.2.1. Writing the Code

In general, the code of a Kernel process follows a standard pattern:

1. Initialization, followed by a
2. Transaction loop.

Initialization declares local variables, performs any handshake with other processes, sends or awaits startup messages, and so on. The transaction loop then executes repeatedly, as long as the process exists, performing one transaction per iteration.

A transaction represents a single execution of the main algorithm of the process, for example:

- Receiving, decoding, and executing a command.
- Receiving, inspecting and forwarding a message.
- Reading a sensor and updating a data store.

The transaction processing code often does the following:

1. Awaits preconditions,
2. Receives input data,
3. Performs computation,
4. Generates output data, and
5. Cleans up and prepares for the next iteration.

The preconditions must be true before the process may proceed with that iteration. Some typical preconditions are: data must be available, a fixed time must have elapsed since the last iteration, a device must have changed state. The process code therefore invokes the appropriate blocking Kernel primitive to wait for the precondition.

3.1.2.2. Naming the Process

The process should be given a logical name consistent with its function, such as:

- "comm interface"
- "data reduction"
- "control table manager"

These names must be globally unique across the distributed network, and indeed they are the only names that have a network-wide meaning.

In addition, a process must have an internal *process identifier* on the node where it is sited and on every other node that has to be aware of it. These *process identifiers* are execution-time values that are usually stored in named Ada variables. These variables should also be given useful names, which might simply reflect the process logical name:

```
comm_interface_ID := declare_process ("comm interface");
```

or which might reflect the role the process plays, as viewed by the node in question:

```
data_source := declare_process ("Position Sensor Reader");
```

The name of the *process identifier* is used in all subsequent invocations of Kernel primitives.

3.1.2.3. Computing the Resources

For each process, the application programmer must specify the main resources that it requires:

- Incoming message queue, and
- Process stack.

The first, the incoming message queue, holds messages sent to the process that have arrived at its local site but have not yet been requested by the receiver via a call to the Kernel primitive *receive_message*. This buffer should be large enough to hold, in effect, the largest anticipated queue of pending messages. Its size therefore depends on the logic of the other processes that are sending to this process, but a reasonable estimate is:

```
(number of messages expected in one itèration) *
(number of iterations the process is allowed to "fall behind"
 the message producers)
```

If this resource is exceeded, arriving messages are handled in a manner specified during process environment creation via the parameter *message_queue_overflow_handling*.

The other resource, the process stack, is for the allocation of Ada local variables. This depends in part on the allocation strategy of the compiler, and can be determined only by an analysis of the generated code or by runtime monitoring. The maximum stack space required is the sum of all local variables and stack frames down the longest call path. It is probably better to obtain this value by experiment rather than by analysis. If this resource is exceeded, the Kernel encapsulation ensures that no other process environment is compromised, and the predefined exception *storage_error* is raised.

These two values are set by the application when invoking the Kernel primitive *create_process*. The parameters that set these values are *message_queue_size* and *stack_size*. See Section 4.6.2.2 for more information.

A message output buffer is provided automatically by the Kernel to maintain messages originated by the process but not yet forwarded to its recipient. These resources are allocated during Kernel initialization, so no tailoring information is required. See the documentation that will be provided with the code for more information about the structure and sizes of these buffers.

3.1.2.4. Creating the Process Environment

When a process is created, the Kernel provides it with a data structure to hold its state and with an encapsulation to control it. The internal resources are allocated by the Kernel in response to parameters supplied at process creation time.

The remainder of the process environment is the set of *external* resources that it requires. Typically, these include:

- Communication partners,
- Global data objects, and
- Library subprograms, including Kernel primitives.

Providing a process with communication partners is straightforward. The process expects to call a communication primitive, fcr example *send_message*, and supplies as a parameter the name of the process to which the message is to be sent. This name must therefore be visible to the process, and one obvious way to achieve this is by enclosing it in a package that the process code imports:

```
with process_table;
package ground_comm_area is

   comm_interface_ID : process_table.process_identifier;
   ...

end ground_comm_area;
```

This variable is initialized by the Main Unit (see Section 2.1.1 for an overview and Section 4.6.2.1 for more details), and can subsequently be used by any process on that node.

Alternatively, since a *process identifier* is a value available at execution time, the process can be sent a message that holds the *process identifiers* of its communication partners.

The global data objects required by a process can likewise be embedded in package specifications.

```
package global_data is

   track_table : array (...) of track;

end global_data;
```

The process may access these objects in the normal way, by simply using the Ada name:

```
with global_data;
procedure comm_interface_process_code is
begin

   update (global_data.track_table (this_bogey));
```

```
end comm_interface_process_code;

with global_data;
procedure control_table_manager_process_code is
begin

   read (global_data.track_table (this_bogey));

end control_table_manager_process_code;
```

However, it must be remembered that such access is *unprotected* ; there is no guard against two processes (such as *comm_interface* and *control_table_manager*) simultaneously modifying the global data. One way to control access to global data is to use a semaphore (see Section 4.11).

Finally, any library subprograms, and of course all the Kernel primitives, may be accessed merely by importing their defining packages and then calling them explicitly, in the normal Ada manner.

3.1.3. Process Life Cycle

Between its creation and final extinction, a process is always in one of these states:

- Running,
- Suspended, or
- Blocked.

A *running* process is actually executing; exactly one process may be *running* on a processor at any given time. A *suspended* process is able to run, but currently not running because the Scheduler has chosen another process to execute. A *blocked* process is unable to run because a precondition is not satisfied: it has called a Kernel primitive that is as yet unable to return.

When it is initially created, a Kernel process is in the suspended state, and the point of suspension is the beginning of the subprogram that is the process code. All created processes remain suspended until initialization is complete, at which point the Scheduler selects one process to execute.

A process runs until either it blocks itself by calling a Kernel primitive, or until it is forcibly suspended to allow another process to run instead. For example, a running process may block by calling the Kernel primitive *wait*, or it may be forcibly suspended because a prior call to *wait* issued by another process expires, and that other process is at a sufficiently high priority to be selected to execute.

Whenever two or more processes are able to run, the Scheduler chooses which one to run, according to deterministic rules presented in Appendix D.

A process is not expected to die. However, a process may cease to exist as a result of certain actions:

- Calling the Kernel primitive *die*.

- Returning from the subprogram that is the code of the Kernel process (which is equivalent to a call of *die*).
- Propagating an exception from the subprogram that is the code of the Kernel process.
- Being killed by another Kernel process.

Once a Kernel process is dead, it may not be restarted, and all attempts to communicate with it are rejected by the Kernel.

3.1.4. Examples

The scenario demonstrated here is a subset of that illustrated in Figure 2-2 on page 14.

3.1.4.1. Network Configuration and NCT Initialization

There are two Kernel nodes in the network: **processor a** and **processor b**. There is one non-Kernel device: **device**. **Processor a** is at bus address: 16#01#; **processor b** is at bus address: 16#02#; **device** is at bus address: 16#03#. The NCT for this network configuration is:

```
procedure make_NCT;

with network_configuration;
with process_table;
procedure make_NCT is
begin

   network_configuration.NCT :=
   (
      (logical_name            => "processor a     ",
       physical_address        => 16#01#,
       Kernel_device           => true,
       needed_to_run           => true,
       allocated_process_ID    => process_table.null_process,
       initialization_order    => 1,
       initialization_complete => false
      ),
      (logical_name            => "processor b     ",
       physical_address        => 16#02#,
       Kernel_device           => true,
       needed_to_run           => true,
       allocated_process_ID    => process_table.null_process,
       initialization_order    => 2,
       initialization_complete => false
      ),
      (logical_name            => "device          ",
       physical_address        => 16#03#,
       Kernel_device           => false,
       needed_to_run           => false,
       allocated_process_ID    => process_table.null_process,
       initialization_order    => 3,
       initialization_complete => false
      )
   );
end make_NCT;
```

Assumptions (i.e., values determined by tailoring the Kernel):

- *Maximum_length_of_processor_name* is defined as 16.
- Type *bus_address* is defined to handle values assigned to it.

For each of the three devices, **processor a**, **processor b**, and **device**, the *logical_name* and *physical_address* fields of the NCT are initialized as described above. Both **processor a** and **processor b** are Kernel devices required to run—thus the settings for the *Kernel_device* and *needed_to_run* fields. **Device** is a non-Kernel device, so *Kernel_device* is set false, as is the *needed_to_run* field. Since a non-Kernel device is, by definition, unable to participate in Kernel protocols, and since the *needed_to_run* field is used to identify those network nodes participating in the Kernel network initialization protocol, this field is false by convention. The *allocated_process_ID* field is explicitly initialized to the *null_process* in all three NCT entries; *initialization_order* is set to indicate a prescribed initialization order (**processor a** first, **processor b** second, **device** not at all); and *initialization_complete* is initialized to false, as initialization has not yet begun.

Two Ada Main Units are required: one to configure **processor a** and one to configure **processor b**. As **device** is a non-Kernel device, it is configured outside Kernel semantics.

3.1.4.2. Software Configuration for Processor a

On **processor a**, there are two Kernel processes of interest: **Merlin** (as shown on Figure 2-2) and **Arthur** (introduced here for this example). On **processor b**, there is one Kernel process of interest: **Vivian** (as shown on Figure 2-2). **Merlin** sends messages to and receives messages from **Arthur**; **Merlin** sends messages to **Vivian**; **Vivian** receives messages from the non-Kernel device.

Based on this information, **processor a** needs *process identifiers* for **Merlin** and **Arthur**, as those two processes are to execute on **processor a**, and a *process identifier* for **Vivian**, as messages are sent to **Vivian**. **Processor a** is defined to be the Master processor during initialization. The following is a template for some general support packages and for the Main Unit for **processor a**.

```
with time_globals;
package timeouts is

   function "+" (left, right : time_globals.elapsed_time)
      return time_globals.elapsed_time
      renames time_globals."+";

   Master_base_time : constant time_globals.epoch_time :=
      time_globals.create_epoch_time
      (
         day    => 0,
         second => 0.0
      );

   Master_timeout : constant time_globals.elapsed_time :=
      time_globals.create_elapsed_time
      (
         day    => 0,
         second => 5.0
      );

   subordinate_timeout : constant time_globals.elapsed_time :=
      time_globals.seconds
      (
         an_integral_duration => 5
      );

   init_complete_timeout : constant time_globals.elapsed_time :=
      Master_timeout + subordinate_timeout;

end timeouts;

----------------------------------------------------------------

with process_managers_globals;
package application_unique_names is

   arthur : process_managers_globals.process_name_type:=
      "arthur                         ";

   device : process_managers_globals.device_name_type :=
      "device                         ";

   merlin : process_managers_globals.process_name_type :=
      "merlin                         ";

   vivian : process_managers_globals.process_name_type :=
      "vivian                         ";

end application_unique_names;

----------------------------------------------------------------

with process_table;
package processor_a_comm_area is

   merlin_ID : process_table.process_identifier;
   arthur_ID : process_table.process_identifier;
   vivian_ID : process_table.process_identifier;

end processor_a_comm_area;

----------------------------------------------------------------

with processor_a_comm_area;
procedure arthur_process_code is
begin
   loop
      -- do arthur's algorithm
      null;
   end loop;
end arthur_process_code;

with processor_a_comm_area;
```

```
procedure merlin_process_code is
begin
   loop
      -- do merlin's algorithm
      null;
   end loop;
end merlin_process_code;

----------------------------------------------------------------

with hardware_interface;
with process_managers;
with process_managers_globals;
with processor_management;
with schedule_types;
with application_unique_names;
with arthur_process_code;
with merlin_process_code;
with processor_a_comm_area;
with timeouts;
with make_NCT;
procedure processor_a_Main_Unit is
begin

   -- do any processor- and application-specific initialization

   make_NCT;

   processor_management.initialize_Master_processor
   (
      base_epoch    => timeouts.Master_base_time,
      timeout_after => timeouts.Master_timeout
   );

   processor_a_comm_area.merlin_ID :=
      process_managers.declare_process
      (
         application_unique_names.merlin
      );

   processor_a_comm_area.arthur_ID :=
      process_managers.declare_process
      (
         application_unique_names.arthur
      );

   processor_a_comm_area.vivian_ID :=
      process_managers.declare_process
      (
         application_unique_names.vivian
      );

   process_managers.create_process
   (
      process_ID          => processor_a_comm_area.merlin_ID,
      address             =>
```

```
            hardware_interface.hw_address (merlin_process_code'address),
         stack_size         => 4_096,
         message_queue_size => 100,
         initial_priority   => schedule_types.highest_priority
      );

      process_managers.create_process
      (
         process_ID         => processor_a_comm_area.arthur_ID,
         address            =>
            hardware_interface.hw_address (arthur_process_code'address),
         stack_size         => 2_048,
         message_queue_size => 10,
         initial_priority   => 4,
         preemptable        => schedule_types.enabled
      );

      -- complete remaining processor- and application-specific initialization

      processor_management.initialization_complete
      (
         timeout_after => timeouts.init_complete_timeout
      );

   end processor_a_Main_Unit;
```

Assumptions (i.e., values determined by tailoring the Kernel):

- *Maximum_length_of_process_name* is defined as 32 (see Section 4.6.3.1).
- Type *priority* is defined to handle values assigned to it (see Section 4.3).

Package *timeouts* sets up some timeout values that are used across the example network. This package demonstrates a variety of ways to obtain *elapsed_time* and *epoch_time* values from Ada types and constants; it does not always provide the most direct way to obtain a Kernel time value. The Kernel uses a different abstraction of time than that provided in Ada, so some strategy of translating from the Ada time model to the Kernel time model must be used by any application. In this example, *Master_base_time* is set to zero. Thus, if any *epoch_time* values are to be referenced by day/month/year components within the application, the application would need to track the day/month/year at the time *initialize_Master_processor* is invoked (this is when the base time on the Master processor is initialized) and offset it by the *epoch_time* of interest. Package *time_globals* provides arithmetic functions to accomplish this. Two *elapsed_time* timeout values are defined and initialized by invoking creation or conversion primitives from *time_globals*. A third *elapsed_time* timeout value is created by adding two others, simply to demonstrate the existence and use of arithmetic functions provided to support the Kernel abstraction of time in package *time_globals*.

Package *application_unique_names* encapsulates the string-valued names for Kernel processes. These are required to be unique across the application. This package is referenced by both Kernel nodes, **processor a** and **processor b**.

Package *processor_a_comm_area* is used to maintain information about the processes of interest on **processor a**: in this case, **Merlin**, **Arthur**, and **Vivian**. A *process identifier* variable is defined for each, as per Section 3.1.2.2 and Section 3.1.2.4.

The code for the **Merlin** process and the **Arthur** process is defined in parameterless procedures *merlin_process_code* and *arthur_process_code* respectively.

The Ada procedure *processor_a_Main_Unit* is the Main Unit that configures **processor a**. (For an overview of the Main Unit, see Section 2.1.1.) As **processor a** is designated the Master processor for initialization, *initialize_Master_processor* is called. The two processes that execute on **processor a** are declared, as is **Merlin**'s communication partner.

The **Merlin** process is created. The process code to be executed, *merlin_process_code*, is associated with the *process identifier* by which **Merlin** is referenced, *processor_a_comm_area.merlin_ID*, in all ensuing Kernel operations. The *stack_size* and *message_queue_size* parameters to *create_process* are set to values deemed appropriate by the application designers. As the application designers have determined that **Merlin** has a high priority, the *initial_priority* parameter is set to the highest possible. All other parameters, *preemptable* and *message_queue_overflow_handling*, have been left to their default values.

The **Arthur** process is created. The process code to be executed, *arthur_process_code*, is associated with the *process identifier* by which **Arthur** is referenced, *processor_a_comm_area.arthur_ID*, in all ensuing Kernel operations. The *stack_size* and *message_queue_size* parameters to *create_process* are set to values deemed appropriate by the application designers. As the application designers have determined that **Arthur** does not have the highest priority, the *initial_priority* parameter is set to some appropriate value (4). As the application designers have determined that **Arthur** should be preemptable if there is another process of equal priority available to run, the *preemptable* parameter is set to *enabled*. The other parameter, *message_queue_overflow_handling*, has been left to its default value.

Once all processes of interest to **processor a** have been declared and created, *initialization_complete* is called. Upon completion of *initialization_complete*, the Kernel processes **Merlin** and **Arthur** begin execution (see Section 3.1.3). The process code for **Merlin** and **Arthur** may reference global data, access library subprograms, and make calls to Kernel primitives.

3.1.4.3. Software Configuration For Processor b

The following is a template for some general support packages and for the Main Unit for **processor b**. Because there is much similarity between this Main Unit and the Main Unit for **processor a**, only those points that differ are discussed here.

```
with process_table;
package processor_b_comm_area is

   merlin_ID : process_table.process_identifier;
   vivian_ID : process_table.process_identifier;
   device_ID : process_table.process_identifier;

end processor_b_comm_area;
```

```
----------------------------------------------------------------

with processor_b_comm_area;
procedure vivian_process_code is
begin
   loop
      -- do vivian's algorithm
      null;
   end loop;
end vivian_process_code;

----------------------------------------------------------------

with communication_management;
with hardware_interface;
with process_managers;
with process_managers_globals;
with processor_management;
with application_unique_names;
with processor_b_comm_area;
with timeouts;
with vivian_process_code;
with make_NCT;
procedure processor_b_Main_Unit is
begin

   -- do any processor- and application-specific initialization

   make_NCT;

   processor_management.initialize_subordinate_processor
   (
      timeout_after => timeouts.subordinate_timeout
   );

   processor_b_comm_area.merlin_ID :=
      process_managers.declare_process
      (
            application_unique_names.merlin
      );

   processor_b_comm_area.device_ID :=
      process_managers.declare_process
      (
            application_unique_names.device
      );


   processor_b_comm_area.vivian_ID :=
      process_managers.declare_process
      (
            application_unique_names.vivian
      );

   process_managers.create_process
   (
      process_ID         => processor_b_comm_area.vivian_ID,
      address            =>
         hardware_interface.hw_address (vivian_process_code'address),
      stack_size         => 8_096,
      message_queue_size => 1_000,
      initial_priority   => 1
   );

   communication_management.allocate_device_receiver
   (
      receiver_process_ID => processor_b_comm_area.vivian_ID,
      device_ID           => 3
   );

   -- complete remaining processor- and application-specific initialization

   processor_management.initialization_complete
   (
      timeout_after => timeouts.init_complete_timeout
   );

end processor_b_Main_Unit;
```

Package *processor_b_comm_area* is used to maintain information about the processes of interest on **processor b**: in this case, **Merlin**, **Vivian**, and the non-Kernel device **Device**.

The code for the **Vivian** process is defined in parameterless procedure *vivian_process_code*. There is no code associated with the non-Kernel device.

The Ada procedure *processor_b_Main_Unit* is the Main Unit that configures **processor b**. As **processor b** is not designated the Master processor for initialization, *initialize_subordinate_processor* is called. The process that executes on **processor b** is declared, as are **Vivian**'s communication partners: **Merlin** and the non-Kernel device.

The **Vivian** process is created. The process code to be executed, *vivian_process_code*, is associated with the *process identifier* by which **Vivian** is referenced, *processor_b_comm_area.vivian_ID*, in all ensuing Kernel operations. The *stack_size* and *message_queue_size* parameters to *create_process* are set to values deemed appropriate by the application designers. As the application designers have determined that **Vivian** has a high priority, the *initial_priority* parameter is set to the 1, which is the highest possible. All other parameters, *preemptable* and *message_queue_overflow_handling*, have been left to their default values.

Vivian is designated as the sole receiver of messages from **Device**. Thus, the Main Unit creates this "binding" by calling *allocate_device_receiver*. The *receiver_process_ID* is the Vivian's *process identifier*; the *device_ID* is the index into the NCT that corresponds to the entry for **Device**.

Once all processes of interest to **processor b** have been declared and created, *initialization_complete* is called. Upon completion of *initialization_complete*, the Kernel process **Vivian** begins execution.

Further examples of using Kernel primitives are found in each of the sections describing a set of Kernel primitives in Chapter 4 and in Appendix E.

3.2. Preparing the Kernel for Use

This section provides an outline of the steps required to prepare the Kernel for use by an application. Tailoring details are provided in Appendix C; a description of actually building the Kernel is provided in the documentation that will be provided with the code.

It is quite conceivable that a single application may execute on more than one instantiation of the Kernel. Some of the tailoring parameters require network-wide consistency; these are described in Section C.1. Other parameters do not require that level of consistency; these are described in Section C.2. Thus, a single application may require a family of Kernels; this is an application design and configuration issue.

1. Set the parameters that require network-wide tailoring as specified in Section C.1.
2. Set the parameters that require processor-specific tailoring as specified in Section C.2.
3. Recompile the Kernel using the information in the documentation that will be provided with the code.

At this point, the Kernel or Kernels are now ready for application use.

3.3. Building an Application Using the Kernel

Section 2.1 and previous sections of this chapter provide a description and some examples of building an application using the Kernel. This section summarizes that in an outline format.

1. Prepare the NCT:
    a. Import *network_configuration* via Ada WITH-clause.
    b. Define a procedure that assigns appropriate values to NCT entries and fields; suggest one global procedure invoked by the Main Unit on all processors.[3]
    c. Compile this application-specific NCT procedure.

   The application-specific NCT is now ready for application use.
2. For each processor running the Kernel:
    a. Define specifications for Kernel processes.

[3] This is the style used by the DARK development team for testing and consistency purposes. There is no reason why the NCT initialization may not be performed inline by Main Unit code.

b. Compile specifications for Kernel processes.

Process specifications now ready for application use.

c. Define the Ada Main Unit.

i. Import *processor_management* via Ada WITH-clause.

ii. Do any preliminary hardware or device initializations.

iii. Call *initialize_Master_processor* or *initialize_subordinate_processor.*

Hardware (processors, devices, and network) is ready for application use.

iv. Import *process_managers* via Ada WITH-clause.

v. Import specifications of processes that are to be created via Ada WITH-clause.

vi. Call *declare_process* and *create_process* to declare and create all processes of interest. (*Create_process* takes the *'address* of the process code, so its name must be known.)

vii. Optionally import packages *interrupt_management, communication_management,* and *process_table* via Ada WITH-clauses.

viii. Perform application-specific initialization (e.g., bind interrupt handlers, associate a Kernel process with a non-Kernel device, declare semaphores for critical, shared resources).

ix. *Compile the Ada Main Unit.*

Main Unit ready to be linked into complete application program.

d. Define the process code.

i. Import any Kernel specification via an Ada WITH-clause **EXCEPT**: *processor_management* or *process_managers*; they are to be used ONLY by the Ada Main Unit.

ii. Write code bodies for Kernel processes (this may be done in parallel with the construction of Ada Main Unit), which may access any of the Kernel primitives.

Process code ready to be linked into complete application program.

e. Link the Ada Main Unit, the code of the Kernel processes, and the Kernel itself.

The executable image is now ready for downloading into hardware.

f. Download the executable image into the target hardware.

g. Start program execution on all subordinate processors.

h. Start program execution on the Master processor.

The application is now running.

4. Kernel Primitives

There are a number of data type packages that are required to support the application's use of the Kernel. These packages provide one or more abstractions to the application program from the Kernel. These abstractions are:

1. Hardware Interface (Section 4.1) - This package provides an interface to compiler-specific primitive types.
2. Time Globals (Section 4.2) - The packages providing this capability abstract the Kernel's concept of time.
3. Schedule Types (Section 4.3) - This package defines the abstractions of priority, preemption, state, and quanta of time.
4. Network Configuration Table (Section 4.4) - The packages providing this capability provide the abstraction of the network configuration.

Each of these sections describes the abstraction in detail:

1. Its **purpose** and the packages that implement the capability,
2. The **mechanism** by which the implementation of the abstraction is provided,
3. The **exported constants**, **exported types**, and **exported data structures** that define the abstraction,
4. The **subprograms** to manipulate values of the exported type or object, and
5. **Related Information**, including: **referenced constants**, **referenced types**, and **relevant generic parameters**.

The Kernel functionality is provided through a number of primitives that may be invoked by an application program. These primitives are grouped into functional areas so that related primitives can be discussed together. The Kernel functional areas are:

1. Processor Management (Section 4.5) - These primitives support the creation and maintenance of the physical network configuration (i.e., the NCT).
2. Process Managers (Section 4.6) - These primitives support the declaration and creation of the logical processor configuration (i.e., communication partners and the Process Table).
3. Communication Management (Section 4.7) - These primitives support communication among Kernel processes and non-Kernel devices.
4. Process Attribute Modifiers (Section 4.8) - These primitives support the modification of attributes of already existing Kernel processes.
5. Process Attribute Readers (Section 4.9) - These primitives support the read-only access to certain Kernel process attributes.
6. Interrupt Management (Section 4.10) - These primitives provide the abstraction of hardware interrupts and their control to the application.
7. Semaphore Management (Section 4.11) - These primitives provide the abstraction of classical (Dykstra) semaphores to control process synchronization and mutual exclusion.
8. Alarm Management (Section 4.12) - These primitives provide the capability to set and cancel alarms (time-triggered events) and to detect the expiration of an alarm.

9. Time Management (Section 4.13) - These primitives provide the manipulation of the abstraction of Kernel time.

10. Timeslice Management (Section 4.14) - These primitives support the round-robin, timeslice scheduling of processes.

For each Kernel functional area, the following information is provided:

1. Its **purpose** and the packages that implement the functionality,
2. The **mechanism** by which the implementation of the functionality is provided,
3. The **subprograms** that implement the functionality being provided, and for each subprogram:
    a. A description of its purpose and use,

    b. One or more samples of its **invocation**, and

    c. **Conditions for blocking** that apply to the subprogram.
4. **Related information**, including: the **exported constants**, **exported types**, and **exported data structures** that support the functionality; and **referenced constants**, **referenced types**, and **relevant generic parameters**.

    The information about **exported constants**, **exported types**, and **exported data structures** is presented in a stylized manner, as follows:

    Name of the constant, type, or data structure
    Description of the constant, type, or data structure
    Value of the constant, type, or data structure

    The information about **referenced constants**, **referenced types**, and **referenced data structures** is also presented in a stylized manner, as follows:

    Name of the constant, type, or data structure - use; cross-reference

Information about resource consumption by each Kernel primitive is target specific and is provided in Appendix H.

This chapter concludes with an index mapping all exported names into the packages that export them.

The basis for most of the examples in the following sections is the example given in Chapter 3.

4.1. Hardware Interface

4.1.1. Introduction

The hardware interface capability comprises the following packages:

1. *Hardware_interface*

See also the documentation that will be provided with the code for more details.

4.1.1.1. Purpose

The Kernel package *hardware_interface* provides an interface to compiler-specific primitive types. Within the Kernel itself, there are no references to the predefined types in Ada package Standard; all references to primitive types use names declared in package *hardware_interface*. By doing this, certain implementation-dependent details are abstracted away from both the Kernel and the application in a uniform manner. It is recommended that applications avoid using package Standard entirely and use package *hardware_interface* for ready compatibility with Kernel primitives.

This strategy facilitates porting Kernel and application software across machines and across compilers. For example, the Ada pre-defined type *integer* could be implemented as a 16-bit integer or a 32-bit integer. When the Kernel requires a 32-bit integer, the exported type *hw_long_integer* is used; when the Kernel requires a 16-bit integer, the exported type *hw_integer* is used. Were the standard type *integer* used, the application programmer would not know from compiler to compiler which size of integer was used without searching through compiler documentation. The Kernel makes this distinction explicit within the Kernel, and provides that same capability to the application as well.

4.1.1.2. Mechanism

Package *hardware_interface* provides this "shield" from variations in the Ada pre-defined types. Exported information includes:

1. Constants that define the Kernel's understanding of hardware layout,
2. Types that interface to compiler primitive types, and
3. Types and conversion functions to manipulate untyped storage within the Kernel.

4.1.2. Exported Constants

Bits_per_byte
The number of bits in a byte
8

Byte
The number of bytes in a byte storage unit
1 (i.e., an 8-bit storage unit)

Longword
The number of bytes in a longword storage unit
4 (i.e., a 32-bit storage unit)

Null_hw_address
A system-wide null address value
0

Word
The number of bytes in a word storage unit
2 (i.e., a 16-bit storage unit)

There are no representation specifications relevant to any of these constants.

4.1.3. Exported Types

Hw_address
Interface to system.address
Values implementation-dependent

Hw_duration
Interface to standard.ouration
-86_400 .. +86_400

Hw_integer
16-bit integer
-32_768 .. +32_767

Hw_long_integer
32-bit integer
-2_147_483_648 .. +2_147_483_647

Hw_long_natural
32-bit 0 .. maximum positive value that can be represented in 32 bits
0 .. +2_147_483_647

Hw_long_positive
32-bit 1 .. maximum positive value that can be represented in 32 bits
1 .. +2_147_483_647

Hw_natural
16-bit 0 .. maximum positive value that can be represented in 16 bits
0 .. +32_767

Hw_positive
16-bit 1 .. maximum positive value that can be represented in 16 bits
1 .. +32_767

Hw_string
Interface to standard.string
Values identical to standard.string

There are representation specifications relevant to each of the following types to ensure exact representation:

- *Hw_duration*
- *Hw_integer*
- *Hw_long_integer*
- *Hw_long_natural*
- *Hw_long_positive*
- *Hw_natural*
- *Hw_positive*

The following types are also exported by package *hardware_interface* but are only used internally within the Kernel:

- *Hw_bits8*
- *Hw_bits8_ptr*
- *Hw_byte*
- *Hw_byte_ptr*

These types are used to support untyped data manipulation within the Kernel itself.

There is a representation specification relevant to type *hw_bits8*; see Appendix H.

4.1.4. Exported Data Structures

None.

4.1.5. Subprograms

A number of functions are provided to manipulate hardware addresses and untyped storage. These are:

- *To_hw_address*, which converts from a value of the Kernel type *hw_long_integer* to a *hw_address.*
- *To_hw_bits8*, which converts from a value of the internal Kernel type *hw_byte* to a *hw_bits8.*
- *To_hw_bits8_ptr*, which converts from a value of the predefined type *system.address* to a *hw_bits8_ptr.*
- *To_hw_bits8_ptr*, which converts from a value of the internal Kernel type *hw_byte_ptr* to a *hw_bits8_ptr.*
- *To_hw_byte_ptr*, which converts from a value of the predefined type *system.address* to a *hw_byte_ptr.*

4.1.6. Related Information

4.1.6.1. Referenced Constants

None.

4.1.6.2. Referenced Types

1. Primitive types in compiler-supplied package Standard and package System.

4.1.6.3. Relevant Generic Parameters

Error checking: none.

Others: none.

4.2. Time Globals

4.2.1. Introduction

The time globals capability comprises the following packages:

1. *Generic_Kernel_time, Kernel_time*
2. *Generic_time_globals, Time_globals*

These packages export to the application the objects that embody the Kernel's concept of time, as described in Section 2.5. These objects include the various data types and constants, the appropriate operations, and conversions between Kernel types and the Ada data type *duration*. All Kernel primitives reference time in terms of *elapsed_time* and *epoch_time*, exported by package *time_globals*, so the appropriate abstractions must be provided.

Packages *generic_Kernel_time* and *Kernel_time* are really internal Kernel packages. However, because package *generic_Kernel_time* exports a tailoring parameter and package *Kernel_time* exports the corresponding constant value, these packages are introduced in this section. Except for these two values, ***the application should never directly access anything exported from these packages***, as this may violate the integrity of the Kernel and the application program. For more information about the representation of time within the Kernel, see Section 5.2.5.

4.2.1.1. Purpose

The purpose of package *time_globals* is to provide the application with a concept of time that can be used for the measurement of elapsed time, the representation of absolute (calendar) time, and the control of events that are required to occur at specific times or after specific intervals.

The model exported by this package is defined in terms of abstract data types and appropriate operations. This model, in turn, is built upon a concrete data type defined within the Kernel, in package *generic_Kernel_time*, which is used by the Kernel for its own time-based operations. The application communicates time values to the Kernel using these abstract data types, and the Kernel, in turn, performs efficient and accurate computations upon them.

4.2.1.2. Mechanism

The representation of Kernel time chosen differs from the Ada types *time* and *duration* in three main ways:

1. A single representation is used internally by the Kernel for both absolute time and for intervals of time.
2. The Kernel's representation of time can accommodate much longer intervals of time than can the Ada type *duration*.
3. The unit of representation of Kernel time is based on decimal fractions of a second, not binary fractions.

This representation is captured by an internal type *Kernel_time*, which is exported to the Kernel by package *Kernel_time* but is not exported to the application.

Two abstract data types are derived from *Kernel_time* and are available to the application via package *time_globals*. They are: *elapsed_time* (relative time) and *epoch_time* (absolute time). These two types have exactly the same concrete semantics, but their abstract semantics are different, in that *elapsed_time* captures the concept of relative time (e.g., time between iterations, time since last message, time to perform computation) and *epoch_time* captures the concept of absolute time (e.g., clock time, calendar time, time-of-day).

Every Kernel primitive that expects a time value comes in two forms, one that takes a relative time and one that takes an absolute time. For example, an application may *delay for* ten seconds, or it may *delay until* 09:30:00; the former uses relative time and the latter absolute time.

The appropriate operations are defined on both abstract types to provide necessary functionality with appropriate safety. For example, two values of type *elapsed_time* may be added, but two values of type *epoch_time* may not; two values of *epoch_time* may be subtracted to give a value of type *elapsed_time*.

Finally, appropriate constants and constructor functions are provided to allow the application to generate specific time values, and conversions are available from the Ada type *duration*.

The application program should never access *Kernel_time*; it should only reference time in terms of *elapsed_time* or *epoch_time*. The *Kernel_time* packages should only be used when building the Kernel itself and tailoring to the real-time clock via the parameter *ticks_per_second*. See Section C.1.2 for more information.

4.2.2. Exported Constants

Ticks_per_second
Set via a tailoring parameter
See Section C.1.2

Zero_elapsed_time
Zero time represented as an *elapsed_time* value
Zero days, zero seconds

Zero_epoch_time
Zero time represented as an *epoch_time* value
Zero days, zero seconds

There are no representation specifications relevant to any of these constants other than those of their base types.

4.2.3. Exported Types

Elapsed_time
Relative time
A high and a low component capable of representing a time of up
to 150_000 years in the future (i.e., 2 ** 63 microseconds)

Epoch_time
Absolute time
A high and a low component capable of representing a time of up
to 150_000 years in the future (i.e., 2 ** 63 microseconds)

Integral_duration
An integral number of seconds
A 32-bit integer

There are no representation specifications relevant to any of these types other than those of their base types.

4.2.4. Exported Data Structures

1. None

4.2.5. Subprograms

4.2.5.1. Base_time

This function returns the *epoch_time* that is the base of the representation of time on the processor. This is the value that is set during processor initialization via the Kernel primitive *initialize_Master_processor* and the time included in the "Go" message that *initialize_Master_processor* sends to each of the subordinate processors.

4.2.5.2. Creation

The creation functions provide the capability to create the abstraction of an *elapsed_time* and an *epoch_time* value from basic, visible types. The two forms of this function are:

1. *Create_elapsed_time*, which takes a *day* and a *second* as parameters and returns an *elapsed_time* value.
2. *Create_epoch_time*, which takes a *day* and a *second* as parameters and returns an *epoch_time* value.

4.2.5.3. Arithmetic Operations Returning Elapsed Time

Arithmetic operations returning *elapsed_time* values are:

- *Elapsed_time + elapsed_time*
- *Elapsed_time - elapsed_time*
- *Epoch_time - epoch_time*
- *Elapsed_time * hw_integer*
- *Hw_integer * elapsed_time*
- *Elapsed_time / hw_integer*

An invocation of any of these functions will raise the predefined exception *numeric_error* if the result of the operation causes an overflow.

4.2.5.4. Arithmetic Operations Returning Epoch Time

Arithmetic operations returning *epoch_time* values are:

- *Epoch_time + elapsed_time*
- *Epoch_time - elapsed_time*

An invocation of any of these functions will raise the predefined exception *numeric_error* if the result of the operation causes an overflow.

4.2.5.5. Comparison Operations on Elapsed Time

Comparison operations taking *elapsed_time* parameters and returning a Boolean result are: "<" "<=" ">" ">=". The default "=" and "/=" operators (i.e., bitwise comparison) are automatically available and yield the correct result.

4.2.5.6. Comparison Operations on Epoch Time

Comparison operations taking *epoch_time* parameters and returning a Boolean result are: "<" "<=" ">" ">=". The default "=" and "/=" operators (i.e., bitwise comparison) are automatically available and yield the correct result.

4.2.5.7. Conversion Functions

A number of functions are provided to convert between the Ada type *duration* and Kernel types that encapsulate time. These are:

- *Seconds*, which converts from a value of Ada type *duration* to an *elapsed_time*.
- *Seconds*, which converts from a value of the Kernel type *integral_duration* to an *elapsed_time*.
- *Milliseconds*, which converts from a value of the Kernel type *integral_duration* to an *elapsed_time*.
- *Microseconds*, which converts from a value of the Kernel type *integral_duration* to an *elapsed_time*.
- *To_elapsed_time*, which converts from a value of Ada type *duration* to an *elapsed_time*.
- *To_Ada_duration*, which converts from a value of the Kernel type *elapsed_time* to an Ada *duration*. Since the Kernel type *elapsed_time* spans a much larger range than does Ada type *duration*, an invocation of this function will raise the predefined exception *constraint_error* if its argument exceeds the range of its result.

The following conversions are also exported by package *time_globals* but are only used internally within the Kernel:

- *To_elapsed_time*, which converts from a value of the internal Kernel type *Kernel_time* to an *elapsed_time*.
- *To_epoch_time*, which converts from a value of the internal Kernel type *Kernel_time* to an *epoch_time*.
- *To_Kernel_time*, which converts from a value of the Kernel type *elapsed_time* to a *Kernel_time*.
- *To_Kernel_time*, which converts from a value of the Kernel type *epoch_time* to a *Kernel_time*.

4.2.6. Related Information

4.2.6.1. Referenced Constants

None.

4.2.6.2. Referenced Types

None.

4.2.6.3. Relevant Generic Parameters

Error checking: none.

Others:

1. *Ticks_per_second_value*: see Section C.1.2.

4.3. Schedule Types

4.3.1. Introduction

The schedule types capability comprises the following packages:

1. *Generic_schedule_types, schedule_types*

4.3.1.1. Purpose

The Kernel package *schedule_types* define the abstractions of priority, preemption, and process state.

4.3.1.2. Mechanism

The Kernel provides the capability for an application to specify and modify the priority of a Kernel process. The priority of a process is initially assigned when the process is created; it may then be modified via calls to a Kernel primitive that just modifies a process's priority (see Section 4.8.2.4), to a Kernel primitive that sets an alarm (see Section 4.12.2.1), or to a Kernel primitive that potentially blocks (see Sections 4.11.2.1, 4.7.2.2, 4.7.2.3, 4.13.2.3, and 4.8.2.5). Type *priority* is an integral type. The lower the value, the higher the priority of the process. The highest priority a process may have is 1; the lowest priority a process may have is set when the Kernel is tailored (see Section C.2.2). The priority value of 0 is a special value for the Kernel; it means that the priority value should not change from its current value.

The Kernel provides the capability for an application to indicate which processes are candidates for round-robin timeslice processing. Type *preemption* is used for this.

A process is always in one of four states (as described in Section 2.4): running, suspended, blocked, or dead. The Kernel uses type *process_state* to represent this.

4.3.2. Exported Constants

Current_process_priority
  Indication that a process's priority should not be modified
    0

Default_preemption
  Default used by the Kernel when preemption is not specified
    Enabled

Default_priority

Default priority used by the Kernel when priority is not specified and "current priority" doesn't make sense
Lowest_priority

Default_process_state
Default process state used by the Kernel
Suspended

Highest_priority
Value the application may specify for the highest priority of a Kernel process
1

Lowest_priority
Bound on type *priority* for process at the lowest priority
Set via a tailoring parameter; see Section C.2.2

An application should *not* use the values *priority'first* or *priority'last*, as they both have special meaning to the Kernel. The highest priority an application process may have is represented by the constant *highest_priority*; the lowest priority an application process may have is represented by the constant *lowest_priority*. Any values in the range *highest_priority* .. *lowest_priority* are legal values for application process priorities.

There are no representation specifications relevant to any of these constants, other than those of their base types.

4.3.3. Exported Types

Preemption
Indication of whether or not process participates in timeslice scheduling
Enumerated (*enabled*, *disabled*)

Priority
Measure of urgency with which a process executes
0 .. *lowest_priority_value*

Process_state
Indication of process execution state
Enumerated (*running*, *suspended*, *blocked*, *dead*)

There are no representation specifications relevant to any of these types, other than those of their base types.

4.3.4. Exported Data Structures

None.

4.3.5. Subprograms

None.

4.3.6. Related Information

4.3.6.1. Referenced Constants

None.

4.3.6.2. Referenced Types

None.

4.3.6.3. Relevant Generic Parameters

Error checking: see Section C.4.

Others:

1. *Lowest_priority_value* - see Section C.2.2.

4.4. Network Configuration Table

4.4.1. Introduction

The network configuration table (NCT) capability comprises the following packages:

1. *Generic_network_configuration*, *network_configuration*
2. *Generic_network_globals*, *network_globals*

4.4.1.1. Purpose

The Kernel package *network_configuration* provides the abstraction of the network configuration to both the Kernel and the application. This is accomplished via the NCT data structure that is exported by *network_configuration.*

The NCT provides the minimum information needed by the Kernel to perform system initialization and its inter-process communication functions. The NCT creates a logical link (via each entry) to a particular hardware device (via the *physical_address* and *Kernel_device* components), thus allowing the rest of the Kernel and the entire application to be hardware independent after initialization.

The packages that describe the network configuration define:

1. The bus address type and related information,
2. The configuration of the NCT,
3. Types used to define the NCT, and
4. Process index table information, which is used by the Kernel as an internal representation of the network configuration.

4.4.1.2. Mechanism

The Kernel provides a mechanism to define legal bus addresses. The range of values allowed are specific to the communication protocol used and must be set accordingly.

The Kernel provides a mechanism to specify the number of nodes in the network. This information is used to configure internal data structures (e.g., the Process Index Table) as well as the NCT.

The Kernel provides the capability for a logical, string-valued name to be associated with each processor in the network, and the capability to configure the size of that string.

The Process Index Table provides the Kernel with fast access to network and process information via internal representations. In fact, the Process Index Table is an inverted index on the Process Table, using the *physical_address* component of the NCT as one of the keys.

Details of the NCT data structure are provided in Section 5.1.1.

4.4.2. Exported Constants

First_bus_address
Lower bound on type *bus_address*, must take actual hardware configuration into consideration, the *physical_address* component of the NCT must not have a lower value than this
Set via a tailoring parameter; see Section C.1.1

Last_bus_address
Upper bound on type *bus_address*, must take actual hardware configuration into consideration, the *physical_address* component of the NCT must not have a higher value than this
Set via a tailoring parameter; see Section C.1.1

Maximum_length_of_processor_name
The maximum length of the *logical_name* component of the NCT
set via a tailoring parameter; see Section C.1.4

Null_address
A null value for type *bus_address*, must take actual hardware configuration into consideration
Set via a tailoring parameter; see Section C.1.1

Number_of_nodes
The number of nodes in the network, the number of entries in the NCT
Set via a tailoring parameter; see Section C.1.1

There are no representation specifications relevant to any of these constants, other than those of their base types.

4.4.3. Exported Types

Bus_address
Range of addresses legal within the network on which the Kernel executes
First_bus_address_value .. last_bus_address_value

Configuration_table
Type to construct NCT
Array

NCT_entry
Type to construct NCT

Record; see Section 5.1.1

Process_index_type

Uniquely identifies a process across the network via its *node_number* and *process_number* relative to that node

Node_number => *first_bus_address_value* .. *last_bus_address_value*,
Process_number => 16-bit integer

Processor_identifier

Index into the NCT, uniquely identifies a processor

1 .. *last_bus_address_value*

There are no representation specifications relevant to any of these types other than those of their base types.

4.4.4. Exported Data Structures

NCT

Logical constant defining the network configuration

See Section 5.1.1

There are no representation specifications relevant to the NCT at this level.

4.4.5. Subprograms

None.

4.4.6. Related Information

4.4.6.1. Referenced Constants

None.

4.4.6.2. Referenced Types

None.

4.4.6.3. Relevant Generic Parameters

Error checking: see Section C.4.

Others:

1. *First_bus_address_value* - see Section C.1.1.
2. *Last_bus_address_value* - see Section C.1.1.
3. *Maximum_length_of_processor_name_value* - see Section C.1.4.
4. *Null_bus_address_value* - see Section C.1.1.
5. *Number_of_nodes_value* - see Section C.1.1.

4.5. Processor Management

4.5.1. Introduction

The processor management capability comprises the following packages:

1. *Generic_processor_management, processor_management*

4.5.1.1. Purpose

Before an application using the Kernel actually begins to execute, the application must inform the Kernel of the actual network configuration on which it is to execute. While a number of network parameters can be configured and verified at compile time, some runtime initialization is required. The *processor_management* package provides one portion of this support.

Initialization requires the creation of an Ada procedure, called the Main Unit, that has the responsibility for configuring the processor for application execution, as well as for configuring the Kernel for that application. As described in Section 2.1.1, the Main Unit is responsible for configuring the processor to meet the requirements of the application. This includes participating in the network initialization protocol—which is exported by package *processor_management*.

The purpose of the subprograms in the *processor_management* package is to verify the physical topology of the system and to initialize Kernel data structures and Kernel-controlled devices (e.g., event timers and the real-time clock). The NCT must be initialized by the application program before these initialization subprograms may execute. The initialization subprograms are then run, and the network topology is defined to the Kernel and the connectivity verified. When initialization is complete, that is, after all processes have been declared and created using the capabilities described in Section 4.6, one final initialization check is made before the Ada Main Unit is permanently descheduled and initialization is determined to be successful.

If failure should occur anywhere during the initialization process, the entire network fails to initialize. This is a simplifying assumption, one that may not be appropriate for all systems.

4.5.1.2. Mechanism

The following data structures are initialized and referenced during processor initialization: the NCT, the Process Table, and the Process Index Table. Once the NCT has been initialized on each processor, the application may invoke either *initialize_Master_processor* or *initialize_subordinate_processor*. There must be one and only one Master processor for the duration of network initialization, and it must invoke *initialize_Master_processor* to coordinate network-wide initialization. All other participating processors are subordinate processors, and must invoke *initialize_subordinate_processor*. Upon successful execution of these initialization subprograms, the Main Unit performs other processor initializations and declares and creates processes. When all initialization work is complete, the Main Unit calls *initialization_complete* to assert to the Kernel that this processor is completely configured and ready to begin application processing.

The Main Unit on all subordinate processors must be running before the Main Unit on the Master processor may run.

Initializing the NCT

The application must have initialized four fields in the NCT prior to invocation of the initialization subprograms. These are: *logical_name*, *physical_address*, *Kernel_device*, and *needed_to_run*. See Section 4.4 and Section 5.1.1 for a description of the NCT.

Process Table and Process Index Table

The declaration and creation of processes cause entries to be created in the Process Table and information to be filled out in the Process Index Table. During the execution of *initialization_complete*, extraneous information is eliminated from these data structures, and some error checking is done. See Section 5.1.3 and Section 5.2.3.

4.5.2. Subprograms

4.5.2.1. Initialize_Master_processor

Initialize_Master_processor must be called once by only one of the Ada Main Units responsible for configuring processors executing the Kernel. No other Kernel primitives may be called prior to its execution.

Following its execution, any of the *process_managers* subprograms and *interrupt_management* subprograms may be invoked by the Main Unit (see Section 4.6 and Section 4.10).

This primitive identifies the invoker as the Kernel processor controlling network initialization. It takes a timeout parameter that controls how long the Master processor waits for any one subordinate to reply to any initialization protocol message. The expiration of this timeout informs the Master processor that network-wide initialization has failed. It is the responsibility of the invoking Main Unit to relay this failure information to the appropriate parties. *Initialize_Master_processor* also takes an *epoch_time* parameter to initialize the clock on the invoking processor and to be used as the initial basis of time across the entire network for all processors running the Kernel.

If this primitive fails for any reason, the network failure message is broadcast to all nodes, and the failure is reported back to the Main Unit. Consequently, a subordinate processor may invoke this primitive, thus effectively declaring itself the new Master and attempting recovery and reinitialization of the network, or the Master may try again.

Invocation

```
processor_management.initialize_Master_processor
(
  base_epoch    => timeouts.Master_base_time,
  timeout_after => timeouts.Master_timeout
);
```

Conditions for Blocking

This procedure always blocks until one of the following conditions occurs:

1. All required processors in the network have acknowledged receipt of the "Go" message, or
2. The initialization timeout expires.

4.5.2.2. Initialize_subordinate_processor

Initialize_subordinate_processor must be called once by all Ada Main Units that are not the Master processor during initialization. No other Kernel primitives may be called prior to its execution.

Following its execution, any of the *process_managers* subprograms and *interrupt_management* subprograms may be invoked by the Main Unit (see Section 4.6 and Section 4.10).

This primitive identifies the invoker as a subordinate Kernel processor participating in network initialization. It takes a timeout parameter that controls how long the subordinate processor waits for any message from the Master processor and for receipt of all "initialization complete" messages from all Kernel processors in the network. The expiration of this timeout informs the subordinate processor that network-wide initialization has failed. It is the responsibility of the invoking Main Unit to relay this failure information to the appropriate parties. *Initialize_subordinate_processor* also awaits the starting *epoch_time* from the Master processor and sets its own time based on that value.

If this primitive fails for any reason, the network failure message is broadcast to all nodes and the failure is reported back to the Main Unit.

Invocation

```
processor_management.initialize_subordinate_processor
(
   timeout_after => timeouts.subordinate_timeout
);
```

Conditions for Blocking

This procedure always blocks until one of the following conditions occurs:

1. The Master has requested the processor's NCT and the subordinate has acknowledged the "Go" message, or
2. The initialization timeout expires.

4.5.2.3. Initialization_complete

Initialization_complete must be called once by all Ada Main Units, Master and subordinate. The only Kernel primitives that may be called prior to its execution are: any of the *process_managers* subprograms and *interrupt_management* subprograms (see Section 4.6 and Section 4.10) and *initialize_Master_processor* or *initialize_subordinate_processor.* It must be called before the Kernel begins execution of any Kernel process.

This primitive asserts that the definition of the physical network topology is complete and that the declaration and creation of all processes on this processor, defining the logical topology, is also complete. This primitive effectively tells the Kernel that it is ready to begin execution of the application, and the Kernel on this processor relays that information to all other Kernels. This primitive takes an optional timeout parameter to detect processor failure after network initialization. It is the responsibility of the invoking Main Unit to relay this failure information to the appropriate parties.

If this primitive fails for any reason, the network failure message is broadcast to all nodes and the failure is reported back to the Main Unit.

Invocation

```
processor_management.initialization_complete
(
   timeout_after => timeouts.init_complete_timeout
);
```

Conditions for Blocking

This procedure always blocks until one of the following conditions occurs:

1. All needed process creation acknowledgements are received, or
2. The initialization timeout expires.

4.5.3. Related Information

None of these subprograms may be invoked from an interrupt service routine.

4.5.3.1. Exported Constants

None.

4.5.3.2. Exported Types

None.

4.5.3.3. Exported Data Structures

None.

4.5.3.4. Referenced Constants

None.

4.5.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2.
2. *Epoch_time* - used for parameters; see Section 4.2.

4.5.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others: none.

4.6. Process Managers

4.6.1. Introduction

The process managers capability comprises the following packages:

1. *Generic_process_managers, process_managers*
2. *Generic_process_managers_globals, process_managers_globals*

4.6.1.1. Purpose

Before initialization is complete, the application program must define a logical configuration of Kernel processes to the Kernel on which it is running and to all Kernels across the network. This logical configuration identifies all communication partners, those that are executing the Kernel and all non-Kernel devices, and all processes that are executing on a single Kernel processor. In addition to describing the logical topology of the processor, the execution environment for each process to execute on the processor must also be created. The subprograms exported by the *process_managers* package provide this capability.

4.6.1.2. Mechanism

A string-valued logical name is used by the application to initially identify processes and communication partners to the Kernel. This name must be unique across the entire application. An internally generated "handle" is returned by the Kernel to the application program, and this handle is then used in all ensuing Kernel activities. The Kernel primitive *declare_process* accomplishes this.

All Kernel processes that execute on a processor do so within an execution environment that must be created by the Kernel. The Kernel primitive *create_process* accomplishes this, as well as providing the initial scheduling profile of the process.

Kernel Process—Ada Code

The code that may be a Kernel process is an outer-level Ada procedure that takes no parameters. "Outer-level" restricts the subprogram to being a library unit itself or being directly visible within a library unit. Chapter 3 provides additional information.

Naming Processes

As described in Chapter 3, a process is referenced in three different ways throughout its life:

1. The application programmer assigns an application-wide unique *logical name* to every process. This *logical name* is a string-valued name.
2. The Kernel takes the *logical name* and associates with that a *process identifier* that is returned to the application via the Kernel primitive *declare_process.* It is via this handle that the application references the Kernel process in all ensuing operations.

3. The Kernel creates an internal *process index table* that it uses to translate *process identifiers* to and from a form that is appropriate for use by the low-level *communication protocol*.

All three representations denote a single process; each of the three is used by different developers and software at different times during the process life cycle.

The Process Execution Environment

When a process is executing, it consists of two parts: the code of the process (i.e., the algorithm being obeyed), and the environment of the process (i.e., the virtual machine the code perceives while running).

This environment consists of three main parts:

1. External resources available to the process,
2. Internal resources available to the process, and
3. The process encapsulation.

The external resources available are all global data objects and all visible subprograms, including the Kernel primitives. These resources are shared by all processes that have visibility into them. The Kernel objects and primitives protect themselves from improper concurrent usage; other global objects must be protected by the application if necessary. For example semaphores (see Section 4.11) may be used to protect shared data areas.

The internal resources available to a process are private to it and are set up when the process is created. These resources include:

1. The stack, which is used for the process's local variables and for the call frames and local variables of any subprograms it calls.
2. The outgoing message buffer, which is managed by the Kernel.
3. The incoming message queue data structure, which is also managed by the Kernel but the size of which is under application control.
4. The Process Table, which holds all process state information and which is managed entirely by the Kernel.

These resources are initialized during process initialization, managed during process execution, and destroyed during process termination.

The process encapsulation contains everything necessary to establish the process as a parallel thread of control, with proper initiation and termination conditions. It performs the following actions:

1. Sets up the initial process state and binds all allocated internal resources to the process.
2. Introduces the process to the Scheduler.
3. Causes process execution to begin at the start of the Ada subprogram identified as a Kernel process.

4. Ensures orderly process termination when the Kernel process subprogram exits or propagates an unhandled exception.

4.6.2. Subprograms

4.6.2.1. Declare_process

The Kernel primitive *declare_process* has three purposes:

1. To declare a Kernel process that will execute on this processor (i.e., be referenced by an ensuing call to *create_process*);
2. To declare a Kernel process with which communication is desired; and
3. To declare a non-Kernel device with which communication is desired.

Declaring a process to the Kernel associates an application-provided, string-valued *name* with a Kernel-generated handle. It is via this handle, called the *process identifier*, that the application references the process in all ensuing Kernel invocations.

The string-valued *name* is not used by the Kernel; it is maintained by the Kernel to aid in application debugging. The length of this name is tailored by setting *maximum_length_of_process_name*.

In the declaration of a Kernel process, the process name may be any Ada string, with the substring of length *maximum_length_of_process_name* maintained by the Kernel.

In the declaration of a non-Kernel device, the device name may be any Ada string that matches a *logical_name* field of a non-Kernel device in the NCT (i.e., the corresponding *Kernel_device* field is false).

Invocation

There are two forms of this primitive: one for declaring a Kernel process and one for declaring a non-Kernel device.

```
processor_b_comm_area.merlin_ID :=
   process_managers.declare_process
   (
      process_managers_globals.process_name_type'
         (application_unique_names.merlin)
   );

processor_b_comm_area.vivian_ID :=
   process_managers.declare_process
   (
      process_managers_globals.device_name_type'
         (application_unique_names.device)
   );
```

Conditions for Blocking

This procedure does not block.

4.6.2.2. Create_process

The Kernel primitive *create_process* creates a process that has previously been declared and initializes the environment in which the designated code is to execute. For each Kernel process to be created, the application specifies:

1. The *address* of the procedure to be executed as a Kernel process;
2. The number of bytes to be allocated for process local information — local variables of the process and all subprograms it calls directly or indirectly, including any call frames generated by the compiler for those called subprograms (i.e., *stack_size*);
3. The maximum number of messages that may be waiting for this process at any time (i.e., *message_queue_size*);
4. The method by which messages are to be handled when the maximum (specified in *message_queue_size*) is reached (i.e., *message_queue_overflow_handling*);
5. Initial scheduling attributes (i.e., *initial_priority* and *preemption*).

If the *stack_size* value is exceeded during data and subprogram access by the process, the predefined exception *storage_error* is raised.

From this information, the Kernel constructs the process execution environment as described previously in this section and enters the process in the set of processes eligible to run. The initial scheduling state of the process is now *suspended.*

Invocation

```
process_managers.create_process
(
   process_ID         => processor_a_comm_area.merlin_ID,
   address            =>
      hardware_interface.hw_address (merlin_process_code'address),
   stack_size         => 4_096,
   message_queue_size => 100,
   initial_priority   => schedule_types.highest_priority,
   preemptable        => schedule_types.disabled
);

process_managers.create_process
(
   process_ID         => processor_a_comm_area.arthur_ID,
   address            =>
      hardware_interface.hw_address (arthur_process_code'address),
   stack_size         => 2_048,
   message_queue_size => 10,
   initial_priority   => 4
);
```

Conditions for Blocking

This procedure does not block.

4.6.3. Related Information

None of these subprograms may be invoked from an interrupt service routine.

4.6.3.1. Exported Constants

Maximum_length_of_process_name
Maximum numbers of characters that are maintained in the Process Table
for the *logical_name* of a process
Set via a tailoring parameter; see Section C.2.5

There are no representation specifications relevant to any of these constants other than those of their base types.

4.6.3.2. Exported Types

Device_name_type
Used to indicate a non-Kernel device name
Variant of *hw_string*

How_to_handle_message_queue_overflow
Indication of how the Kernel should handle the case where more messages
arrive than the incoming message queue is capable of handling
Enumerated (*drop_newest_message*)

Process_name_type
Used to indicate a Kernel process name
Variant of *hw_string*

There are no representation specifications relevant to any of these types other than those of their base types.

4.6.3.3. Exported Data Structures

None.

4.6.3.4. Referenced Constants

1. *Default_preemption* - used for parameters; see Section 4.3
2. *Default_priority* - used for parameters; see Section 4.3

4.6.3.5. Referenced Types

1. *Preemption* - used for parameters; see Section 4.3
2. *Priority* - used for parameters; see Section 4.3
3. *Process_identifier* - used for parameters; see Section 5.1.3

4.6.3.6. Relevant Generic Parameters

Error checking: see section C.4.

Others:

1. *Maximum_length_of_process_name_value* - see Section C.2.5.

2. *Maximum_message_queue_size_value* - see Section C.2.1.

3. *Maximum_process_stack_size_value* - see Section C.2.1.

4.7. Communication Management

4.7.1. Introduction

The communication management capability comprises the following packages:

1. *Generic_communication_globals, communication_globals*
2. *Generic_communication_management, communication_management*

The communication model is presented in Section 2.8. The communication primitives provided by the Kernel are untyped; an application may readily build typed message passing on top of them. An example of such a package is provided in Section E.1.

4.7.1.1. Purpose

The *communication_management* package provides the capability for independent threads of control (i.e., Kernel processes) to communicate among themselves and with non-Kernel devices. This communication is done point-to-point, either synchronously or asynchronously.

4.7.1.2. Mechanism

During process initialization, communication partners are identified via calls to the Kernel primitive *declare_process* (see Section 4.6.2.1). The handles, the *process identifiers*, for these declared processes must be available to the application to be used in interprocess communication. This is discussed in Chapter 3.

The communication management subprograms are:

1. *Send_message* - to send a message with no waiting for any kind of acknowledgement of receipt. This is a "blind" asynchronous send.
2. *Send_message_and_wait* - to send a message and wait for an acknowledgement of receipt of the message by the receiving Kernel process. This is a synchronous send.
3. *Receive_message* - to obtain a message sent by a process. This receives any message from any process, and may be used synchronously or asynchronously.
4. *Allocate_device_receiver* - to assign a Kernel process to be the sole receiver of messages from a non-Kernel device.

All communication primitives appear the same to the application code whether the processes are Kernel or non-Kernel processes, whether they are sited on the same processor (local) or on different processors (remote). The Kernel optimizes local communication by using its knowledge of where the receiver's incoming message queue is located. The Kernel places sent messages directly in that queue, as opposed to incurring network traffic for local messages.

4.7.2. Subprograms

4.7.2.1. Send_message

This primitive sends a message from one process to another, without waiting for acknowledgement of message receipt. Any Kernel process may invoke this primitive at any time. It may be used to send a message to Kernel and non-Kernel processes. For each message to be sent, the caller specifies:

1. The *process identifier* of the intended *receiver* of the message;
2. The application-defined *message_tag* that can be used by the *receiver* to decode the text of the message;
3. The *message_length* of the message to be sent; and
4. The *address* of the buffer containing the *message_text* itself.

When sending to a Kernel device, as can be determined from the NCT, this information is bundled into a *datagram* as described in Section 2.8 and sent to the intended *receiver.* When sending to a non-Kernel device, the message text itself must contain all necessary communication protocol information; the Kernel simply passes the message through to the address of the receiver.

Invocation

```
-- from within the body of merlin_process_code

-- vivian is a remote process

communication_management.send_message
(
   receiver       => processor_a_comm_area.vivian_ID,
   message_tag    => processor_a_comm_area.type_1_message,
   message_length => processor_a_comm_area.type_1_message_length,
   message_text   =>
      hardware_interface.hw_address
         (local_message_buffer'address)
);

-- arthur is a local process; no difference

communication_management.send_message
(
   receiver       => processor_a_comm_area.arthur_ID,
   message_tag    => processor_a_comm_area.type_1_message,
   message_length => processor_a_comm_area.type_1_message_length,
   message_text   =>
      hardware_interface.hw_address
         (local_message_buffer'address)
);
```

Conditions for Blocking

This procedure does not block.

4.7.2.2. Send_message_and_wait

This primitive sends a message from one Kernel process to another Kernel process and waits for acknowledgement of message receipt by the receiving Kernel process. Any Kernel process may invoke this primitive at any time. It may be used to send a message only to Kernel processes. For each message to be sent, the caller specifies:

1. The *process identifier* of the intended *receiver* of the message;
2. The application-defined *message_tag* that can be used by the *receiver* to decode the text of the message;
3. The *message_length* of the message to be sent;
4. The *address* of the buffer containing the *message_text* itself;
5. An optional timeout of one of two kinds:
    - A *timeout_after* - a relative time after which the Kernel on which the receiving process is executing aborts the attempt to communicate with the receiver; or
    - A *timeout_at* - an absolute time at which the Kernel on which the receiving process is executing aborts the attempt to communicate with the receiver.

    A timeout of zero or some previous time implies that the *receiver* must be pending on a call to the Kernel primitive *receive_message*; if it is not, then a negative acknowledgement indicating the contrary must be returned immediately from the receiver's Kernel.
6. An optional *resumption_priority* to take effect when the sending process becomes unblocked.

This information is bundled into a *datagram* as described in Section 2.8 and sent to the intended *receiver.*

If the message is not received by the specified timeout, a negative acknowledgement is returned by the receiver's Kernel to the sender's Kernel, and that information is propagated to the sending process.

Invocation

There are three forms of this primitive: one for an infinite timeout (i.e., there is no timeout parameter); one for an *elapsed_time* timeout; one for an *epoch_time* timeout.

```
communication_management.send_message_and_wait
(
   receiver            => processor_a_comm_area.arthur_ID,
   message_tag         => processor_a_comm_area.type_2_message,
   message_length      => processor_a_comm_area.type_2_message_length,
   message_text        =>
      hardware_interface.hw_address
         (local_outgoing_message_buffer'address),
   resumption_priority => 2
);
```

```
communication_management.send_message_and_wait
(
   receiver            => processor_a_comm_area.arthur_ID,
   message_tag         => processor_a_comm_area.type_2_message,
   message_length      => processor_a_comm_area.type_2_message_length,
   message_text        =>
      hardware_interface.hw_address
         (local_outgoing_message_buffer'address),
   timeout_after       => time_globals.milliseconds (100)
);

communication_management.send_message_and_wait
(
   receiver            => processor_a_comm_area.arthur_ID,
   message_tag         => processor_a_comm_area.type_2_message,
   message_length      => processor_a_comm_area.type_2_message_length,
   message_text        =>
      hardware_interface.hw_address
         (local_outgoing_message_buffer'address),
   timeout_at          => time_globals.create_epoch_time (0, 0.100)
);
```

Conditions for Blocking

This procedure always blocks until one of the following conditions occurs:

1. The receiving process has requested and received the message (i.e., the message has been copied into the receiving process's buffer), or
2. The message timeout expires.

4.7.2.3. Receive_message

This primitive receives a message from another process. Any Kernel process may invoke this primitive at any time. It may be used to receive messages from Kernel devices and non-Kernel devices. For each message to receive, the caller receives the following information:

1. The *process identifier* of the *sender* of the message;
2. The application-defined *message_tag* that can be used by the *receiver* to decode the text of the message (not valid for a message from a non-Kernel device);
3. The *message_length* of the message received;
4. The *address* of the *message_buffer* into which the message text itself is placed;
5. The *buffer_size* of the *message_buffer*;

In addition, the caller may specify:

1. An optional *resumption_priority* to take effect when the sending process becomes unblocked.
2. An optional timeout of one of two kinds:
   - A *timeout_after* - a relative time after which the Kernel on which the receiving process is executing aborts the attempt to receive a message; or
   - A *timeout_at* - an absolute time at which the Kernel on which the receiving process is executing aborts the attempt to receive a message.

A timeout of zero or some previous time prevents the calling process from blocking; if no message is available at the time of call, *receive_message* returns immediately to the caller.

3. A required flag *messages_lost* indicating whether or not the Kernel has had to lose messages as the receiver's incoming message queue is full.

When receiving from a Kernel device, as can be determined from the NCT, this information is collected from the *datagram* as described in Section 2.8 and passed onto the intended *receiver.* When receiving from a non-Kernel device, the message text itself must contain all necessary communication protocol information; the Kernel simply passes the message through to the incoming message queue of the receiver.

If the message is not available by the specified timeout, that information is propagated to the receiving process.

Invocation

There are three forms of this primitive: one for an infinite timeout (i.e., there is no timeout parameter); one for an *elapsed_time* timeout; one for an *epoch_time* timeout.

```
communication_management.receive_message
(
   sender              => local_sender,
   message_tag         => local_tag,
   message_length      => local_length,
   message_buffer      => hardware_interface.hw_address
      (local_receive_buffer'address),
   buffer_size         =>
      processor_a_comm_area.arthur_max_incoming_message_length,
   resumption_priority => 3,
   messages_lost       => local_messages_lost
);

communication_management.receive_message
(
   sender              => local_sender,
   message_tag         => local_tag,
   message_length      => local_length,
   message_buffer      => hardware_interface.hw_address
      (local_receive_buffer'address),
   buffer_size         =>
      processor_a_comm_area.arthur_max_incoming_message_length,
   timeout_after       => time_globals.milliseconds (1_000),
   messages_lost       => local_messages_lost
);

communication_management.receive_message
(
   sender              => local_sender,
   message_tag         => local_tag,
   message_length      => local_length,
   message_buffer      => hardware_interface.hw_address
      (local_receive_buffer'address),
    buffer_size         =>
       processor_a_comm_area.arthur_max_incoming_message_length,
    timeout_at          =>
       time_globals.create_epoch_time (0, 1_000.0),
    messages_lost       => local_messages_lost
);
```

Conditions for Blocking

This procedure blocks only if there is no message currently available for the process. If no message is available, then it blocks until one of the following conditions occurs:

1. A message arrives for the process, or
2. The timeout expires.

4.7.2.4. Allocate_device_receiver

This primitive assigns a specific Kernel process to be the receiver of all messages originating from a specific non-Kernel device. The Kernel itself does nothing with messages from non-Kernel devices other than passing them to their surrogates as identified via a call to *allocate_device_receiver*; thus, the receiving Kernel process must know the format of such messages. The caller specifies:

1. The *receiver_process_ID*, the *process_identifier* of the Kernel process to receive the message.
2. A *device_ID*, a *processor_identifier* index of the entry in the NCT of the non-Kernel device.

Invocation

```
communication_management.allocate_device_receiver
(
    receiver_process_ID => processor_b_comm_area.vivian_ID,
    device_ID           => 3
);
```

Conditions for Blocking

This procedure does not block.

4.7.3. Related Information

Only the non-blocking subprograms *send_message* and *allocate_device_receiver* may be invoked from an interrupt service routine.

4.7.3.1. Exported Constants

Maximum_message_length

The maximum number of bytes that may be sent in a single message
Set via a tailoring parameter; see Section C.1.3

There are no representation specifications relevant to any of these constants other than those of their base types.

4.7.3.2. Exported Types

Message_length_type
16-bit value defining the range of number of bytes that may be sent in a single message
0 .. *maximum_message_length_value*

Message_tag_type
16-bit value that may be used by the application to indicate the type of the message
-32_768 .. +32_767

There are no representation specifications relevant to any of these types other than those of their base types.

4.7.3.3. Exported Data Structures

None.

4.7.3.4. Referenced Constants

1. *Current_process_priority* - used for parameters; see Section 4.3

4.7.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2
2. *Epoch_time* - used for parameters; see Section 4.2
3. *Priority* - used for parameters; see Section 4.3
4. *Process_identifier* - used for parameters; see Section 5.1.3
5. *Processor_identifier* - used for parameters; see Section 4.4

4.7.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others:

1. *Maximum_message_length_value* - see Section C.1.3.

4.8. Process Attribute Modifiers

4.8.1. Introduction

The process attribute modifiers capability comprises the following packages:

1. *Generic_process_attribute_modifiers, process_attribute_modifiers*

4.8.1.1. Purpose

By invoking any Kernel primitive that may block, a Kernel process has the ability to modify its state (to become blocked) and its resumption priority (any legal priority value to be assigr.ed to the process when it becomes unblocked and eligible for scheduling). In addition, a process may modify certain of its own scheduling attributes specifically: its state, its ability to participate in timeslice scheduling, and its priority. A Kernel process may also cause another specified process to die.

4.8.1.2. Mechanism

The Kernel primitives that provide a Kernel process the capability to modify its state specifically are: *die* - which causes the process to terminate itself irrevocably; and *wait* - which causes the process to block itself unconditionally for a relative time or until an absolute time. The Kernel primitive that provides a Kernel process the capability to specify whether or not it is to participate in timeslice scheduling is: *set_process_preemption.* The Kernel primitive that provides a Kernel process the capability to specify its execution priority is: *set_process_priority.* The Kernel primitive that provides a Kernel process the capability to kill another Kernel process is: *kill.*

4.8.2. Subprograms

4.8.2.1. Die

This primitive terminates the calling process. It may be invoked by any process at any time after initialization is complete. When a process dies, all messages pending are discarded and no further messages are queued for it; all negative acknowledgements to any pending messages sent via the Kernel primitive *send_message_and_wait* are returned as required; all space allocated to the messages is reclaimed.

Only Kernel processes may be terminated by this primitive. Terminating a non-Kernel process is the application's responsibility.

Invocation

```
process_attribute_modifiers.die;
```

Conditions for Blocking

This procedure does not block.

4.8.2.2. Kill

This primitive aborts the specified process. It may be invoked by any process at any time after initialization is complete. When a process is killed, all messages pending are discarded and no further messages are queued for it; negative acknowledgements to any pending messages sent via the Kernel primitive *send_message_and_wait* are returned as required; all space allocated to the messages is reclaimed.

The calling process specifies the *process_ID* of the process, local or remote, to be killed.

Invocation

```
process_attribute_modifiers.kill
(
   process_ID => processor_a_comm_area.arthur_ID
);
```

Conditions for Blocking

This procedure does not block.

4.8.2.3. Set_process_preemption

This primitive changes the preemption status of the calling process. This primitive may be invoked by any process any time after initialization.

The preemption status of a Kernel process indicates whether or not it is to participate in timeslice scheduling. If the *preemption* is *enabled*, then the Kernel process is eligible for timeslice scheduling; if the *preemption* is *disabled*, then the process is ineligible. See Section 4.14 and Appendix D for more information about time slicing.

Invocation

```
process_attribute_modifiers.set_process_preemption
(
   preemptable => schedule_types.disabled
);
```

Conditions for Blocking

This procedure does not block.

4.8.2.4. Set_process_priority

This primitive changes the priority of the calling process. This primitive may be invoked by any process any time after initialization.

The priority of a Kernel process indicates the urgency with which its processing should be executed. Lower numeric values for *new_priority* represent greater urgency; higher numeric values represent lesser urgency. The constant *current_process_priority* is used to indicate that no change in the process priority should occur.

Invocation

```
process_attribute_modifiers.set_process_priority
(
   new_priority => 1
);

process_attribute_modifiers.set_process_priority
(
   new_priority => schedule_types.lowest_priority
);
```

Conditions for Blocking

This procedure does not block.

4.8.2.5. Wait

This primitive suspends the caller for a specified relative time (via the *for_elapsed_time* parameter) or until a specified absolute time (via the *until_epoch_time* parameter). A timeout of zero or some previous time prevents the calling process from blocking; the wait does not occur. This primitive may be invoked by any process any time after initialization.

As *wait* is always potentially blocking, the *resumption_priority* parameter, providing a new priority at which the process is to execute when it unblocks, is provided.

Invocation

There are two forms of this primitive: one for an *elapsed_time* suspension; one for an *epoch_time* suspension.

```
process_attribute_modifiers.wait
(
   until_epoch_time    => time_globals.base_time + five_seconds,
   resumption_priority => schedule_types.highest_priority
);

process_attribute_modifiers.wait
(
   for_elapsed_time    => five_seconds
);
```

Conditions for Blocking

This procedure always blocks until the delay (i.e., the specified elapsed or absolute time) expires.

4.8.3. Related Information

None of these subprograms may be invoked from an interrupt service routine.

4.8.3.1. Exported Constants

None.

4.8.3.2. Exported Types

None.

4.8.3.3. Exported Data Structures

None.

4.8.3.4. Referenced Constants

1. *Current_process_priority* - used for parameters; see Section 4.3

4.8.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2
2. *Epoch_time* - used for parameters; see Section 4.2
3. *Preemption* - used for parameters; see Section 4.3
4. *Priority* - used for parameters; see Section 4.3

5. *Process_identifier* - used for parameters; see Section 5.1.3

4.8.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others: none.

4.9. Process Attribute Readers

4.9.1. Introduction

The process attribute readers capability comprises the following packages:

1. *Generic_process_attribute_readers, process_attribute_readers*

4.9.1.1. Purpose

The functions exported by package *process_attribute_readers* provide the capability for a Kernel process to query some of its scheduling attributes and to ascertain its own identity. In addition, a query to determine the string-valued *logical_name* of any other process is also provided.

4.9.1.2. Mechanism

The functions provided for a process to obtain information about itself are:

1. *Get_process_preemption* - determines whether or not the process is participating in timeslice scheduling.
2. *Get_process_priority* - ascertains the priority at which the process is currently executing.
3. *Who_am_I* - obtains the *process identifier* of the process itself.

The function provided to obtain the string-valued name of another process is: *name_of.*

4.9.2. Subprograms

4.9.2.1. Get_process_preemption

This function returns the current value of the preemption status of the calling process. This primitive may be invoked by any process any time after initialization. If the return value is *enabled*, then this process is a participant in timeslice scheduling; if the return value is *disabled*, then this process is not a participant.

Invocation

```
merlin_preemption := process_attribute_readers.get_process_preemption;
```

Conditions for Blocking

This procedure does not block.

4.9.2.2. Get_process_priority

This function returns the current value of the priority of the calling process. This primitive may be invoked by any process any time after initialization. The highest priority process executes at *priority* value of 1; the lowest priority process executes at a *priority* value that is specified by a tailoring parameter; see Section C.2.2.

Invocation

```
merlin_priority := process_attribute_readers.get_process_priority;
```

Conditions for Blocking

This procedure does not block.

4.9.2.3. Who_am_I

This function returns the *process identifier* of the calling process. This primitive may be invoked by any process any time after initialization.

Invocation

```
duplicate_merlin_ID := process_attribute_readers.who_am_I;
```

Conditions for Blocking

This procedure does not block.

4.9.2.4. Name_of

This function returns the string-valued *logical_name* (the name provided when the Kernel primitive *declare_process* was invoked) of the calling process. This primitive may be invoked by any process any time after initialization.

Invocation

```
vivian_name :=
   process_attribute_readers.name_of
   (
      process_ID => processor_a_comm_area.vivian_ID
   );

my_name :=
   process_attribute_readers.name_of
   (
      process_ID => process_attribute_modifiers.who_am_I
   );
```

Conditions for Blocking

This procedure does not block.

4.9.3. Related Information

Only the subprogram *name_of* may be invoked from an interrupt service routine.

4.9.3.1. Exported Constants

None.

4.9.3.2. Exported Types

None.

4.9.3.3. Exported Data Structures

None.

4.9.3.4. Referenced Constants

None.

4.9.3.5. Referenced Types

1. *Preemption* - used for parameters; see Section 4.3
2. *Priority* - used for parameters; see Section 4.3
3. *Process_identifier* - used for parameters; see Section 5.1.3

4.9.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others: none.

4.10. Interrupt Management

4.10.1. Introduction

The interrupt management capability comprises the following packages:

1. *Generic_interrupt_globals, interrupt_globals*
2. *Generic_interrupt_management, interrupt_management*

Real-time embedded system applications usually require very fast processing, some kind of low-level interface, and a way to respond to asynchronous events. Asynchronous events are detected and processed in one of two basic ways: by polling or through interrupts. This section covers only asynchronous events and, in particular, interrupts. An interrupt is defined as an event that causes a temporary asynchronous change in control from normal processing.

Interrupt generation and interrupt handling are the two facets of interrupts. Interrupt generation is hardware and application specific and is not covered here. Interrupt handling is the primary topic of this section.

Two terms are defined here and will be used throughout the ensuing discussion: interrupt servicing and interrupt handling. The term *interrupt servicing* refers *only* to that processing done to acknowledge the event which caused the interrupt. The processing steps involved in servicing an interrupt are encapsulated in one routine called an Interrupt Service Routine (ISR). ISRs are sometimes called interrupt handlers. *Interrupt handling*, on the other hand, is more encompassing and refers to *all* the processing in hardware and software performed in response to an interrupt. Interrupt servicing is therefore only part of interrupt handling.

Two further concepts are important to an understanding of the Kernel interrupt mechanism.

The first concept is that of *interrupt priorities*. It applies only in a priority-based interrupt architecture, such as that found in the Motorola MC680X0 processor family. During normal processing, any interrupt may initiate interrupt handling. However, if the processor is already handling one interrupt, a new interrupt request is recognized if and only if it has a higher priority than that interrupt currently being handled. For the sake of discussion, it is assumed that no interrupt handling is in progress when an interrupt occurs.

The second concept to be defined is *interrupt vector* or *interrupt number*. Every interrupt is associated with a number called the interrupt number, or more commonly, the interrupt vector. Each processor architecture provides a mechanism for accomplishing this.

The Kernel supports both *non-preemptive* and *preemptive* interrupts. A *non-preemptive* interrupt causes the currently running Kernel process to be temporarily suspended to allow the ISR to execute. After the ISR has completed, the processor resumes executing the suspended Kernel process at the point where it was suspended. This type of interrupt is sometimes called a "fast interrupt." During the processing of a *preemptive* interrupt, however, the Kernel Scheduler is invoked instead of returning to the suspended Kernel process. This latter case is used when interrupt handling could possibly cause a change in process attributes such that a Kernel process other than the currently executing process would become eligible for execution. This would then facilitate changing the state of a process upon the expiration of a timeout. This type of interrupt requires that the full context of a process be saved, as it may not be resumed when the ISR completes, and, as such, is not as fast as a *non-preemptive* interrupt. The application may specify whether an ISR is *non-preemptive* or *preemptive*.

4.10.1.1. Purpose

The Kernel's interrupt management facility provides the primitives necessary for interrupt handling. It allows the application to service interrupts in a consistent and flexible manner, and protects the developer from needing to know too much about the target. This enables the application code to be reasonably target independent. The interrupt management facility has been implemented in a way that provides the most efficient interrupt handling for the target implementation. All of the target dependencies have been encapsulated in one place, so porting the application (and the Kernel) to other targets should be relatively simple.

4.10.1.2. Mechanism

Definitions and Terminology

The target hardware provides support for a set of interrupts, each of which is represented within the Kernel by an integer value called the *interrupt_name*. For completeness, the Kernel represents all possible *interrupt_names* and maintains information on each one. This information is maintained in an internal data structure called the Interrupt Table, which is described in detail in Section 5.2.4. However, the Kernel provides the application with access to only some of the interrupts defined by the target hardware.

The Kernel identifies each interrupt as being *reserved* for use exclusively by the hardware or Ada runtime environment, reserved for exclusive use by the *Kernel*, used by the *application*, or *absent* (not used). Interrupts that are not *reserved*, used by the *Kernel*, or claimed by the *application* are defined to be *absent*. The Kernel restricts the application from accessing those interrupts that are classified as *reserved* or *Kernel* interrupts.

The Kernel identifies each interrupt as being either *bound* or *unbound*. A *bound* interrupt has an interrupt service routine (ISR) associated with it. This association is established explicitly by the application via the Kernel primitive *bind_interrupt_handler* or transparently by the Kernel during Kernel initialization.

When binding is performed, the interrupt is identified as either *non-preemptive* or *preemptive*. Interrupt entry is the same for both kinds of interrupts; the behavior differs at interrupt exit, as described in Section 4.10.1.

The Kernel supports dynamic binding of interrupt handlers; the application may bind a different ISR to an interrupt at any time, by invoking the Kernel primitive *bind_interrupt_handler*, and as often as necessary.

An interrupt is usually generated by an external device, but may also be simulated by software via the Kernel primitive *simulate_interrupt*. The Kernel tracks the *interrupt source* (hardware - *external* or software - *internal*) while it is being handled.

An interrupt is either *enabled* or *disabled*. For an interrupt that is *enabled*, the Kernel executes the ISR bound to it when the interrupt occurs or is simulated via the Kernel primitive *simulate_interrupt*; the Kernel simply dismisses any interrupt that is *disabled*. An interrupt may always be enabled via the Kernel primitive *enable*. Sometimes, an interrupt must also be enabled at the hardware device itself. This capability is outside the control of the Kernel and is the responsibility of the application. Before an interrupt may be *enabled*, an interrupt handler must be bound to it. An interrupt may be *disabled* via the Kernel primitive *disable*.

Interrupt Handling

Interrupt handling is provided via both hardware and software. Virtually all processors are designed with some kind of interrupt mechanism. The Kernel builds a layer of abstraction on top of the hardware so that a common interrupt handling mechanism is provided, no matter on what processor family the application executes.

Interrupt handling involves several steps: initialization, interrupt recognition, table lookup, ISR execution, and process resumption.

Initialization. Both the Kernel and application perform operations to initialize interrupt handling. The Kernel initializes the Interrupt Table with all information known about reserved and Kernel interrupts. The binding of these interrupts is done automatically by the Kernel at Kernel initialization time. The application may then bind any interrupts it requires via the Kernel primitive *bind_interrupt_handler*. The Interrupt Table maintains all the information known to the Kernel about all interrupts, such as the ISR address, the binding status, etc. See Section 5.2.4 for more details about the Interrupt Table. Before an interrupt is expected, an ISR must be bound to the interrupt, and the interrupt must be *enabled*, in that order. If an interrupt is not *enabled* (i.e., the interrupt is still *disabled*), it is totally ignored.

Interrupt recognition. In order for an interrupt to be recognized, it must compete with other interrupts that occur at the same time or that are currently being processed. That is, it must have a higher priority than the current processor priority and then any simultaneous interrupts. Many targets use interrupt priority to choose which interrupt to recognize. This arbitration is handled solely by the hardware. Interrupt handling that is started via the Kernel primitive *simulate_interrupt* does not have to contend for recognition.

Table lookup. Once the interrupt is recognized, the current running process is suspended, and the entry in the Interrupt Table associated with the interrupt is checked to see if the interrupt is currently *enabled*. If the interrupt is not *enabled*, the interrupt is dismissed, and the process is resumed with minimal delay.

However, if the interrupt is *enabled*, the current context is saved, the *interrupt source* is set to indicate from where the interrupt originated (i.e., externally via a hardware device or internally via the Kernel primitive *simulate_interrupt*), and the associated ISR, as determined from the Interrupt Table, is called.

ISR execution. The ISR then executes. Unless it is interrupted by another interrupt, it executes to completion. It is not unusual to have the first operation within the ISR disable the interrupt and the last operation re-enable the interrupt. One reason to disable interrupts may be that once the application begins processing a specific interrupt, it is not ready to service another interrupt. After the processing has completed, the interrupts are enabled so they may be serviced again.

Process resumption. After the interrupt is serviced, one of two things happens, depending on whether the interrupt was identified as *non-preemptive* or *preemptive*. As described previously, a *non-preemptive* interrupt would simply cause the suspended process or ISR to be resumed after restoring its context; a *preemptive* interrupt would cause the Kernel Scheduler to be invoked.

Interrupt Names and Reserved Interrupts

The application references interrupts by an *interrupt_name* (i.e., an integer value) with a hardware-specific range. For the Motorola 68020 implementation, this range is defined in Appendix H. The names are mapped directly to interrupt vectors. Some names are reserved by the hardware or the Kernel and may not be used by the application.

Interrupt_names that are reserved by the Kernel or hardware (and, thus, are unavailable to the application) are identified in Appendix H for the Motorola 68020 implementation.

Interrupt Handler—Ada Code

An interrupt handler may be declared as an outer-level Ada procedure with no parameters or as an assembly language routine that follows Ada's linkage conventions. One or more handlers may be declared for each interrupt, but only one may be bound to an interrupt at any one time. An interrupt handler must be bound before the corresponding interrupt is enabled. The interrupt handler encapsulates all of the steps necessary to service the interrupt. Good program design dictates that an interrupt handler be written to be short and efficient to minimize the length of time normal processing is suspended. In support of this, the Kernel does not allow any blocking Kernel primitives to execute within an ISR.

The following is an example of a typical interrupt handler definition:

```
procedure receiver_interrupt_handler is
begin

  --
  -- disable device interrupt capability
  --
  -- check receiver status
  --
  -- if status is good then
  --    get data received and store in buffer
  -- end if
  --
  -- re-enable device interrupt capability
  --

end receiver_interrupt_handler;
```

An interrupt handler is not allowed to invoke any Kernel primitive that may block, as this could result in deadlock.

4.10.2. Subprograms

4.10.2.1. Bind_Interrupt_handler

This Kernel primitive identifies an interrupt service routine that is to be called when the named interrupt occurs. It also identifies whether the interrupt is preemptive or non-preemptive.

When the application invokes *bind_interrupt_handler*, the Kernel locates and checks the appropriate entry in the Interrupt Table. Improper values for the interrupt name or handler address raise *illegal_interrupt* or *illegal_interrupt_handler_address* respectively. If the interrupt name corresponds with a reserved interrupt, *reserved_interrupt* exception is raised. Otherwise, the address of the ISR is stored in the Interrupt Table, and, when the named interrupt occurs and is enabled, the associated code is executed as the ISR.

Bind_interrupt_handler may be invoked anytime it is necessary to change the ISR for an interrupt. *Replacing_previous_interrupt_handler* is raised each time after the initial call.

Invocation

```
IO_device : constant interrupt_globals.interrupt_name := 200;
   --
   -- on one processor, the IO_device has already been set up with
   -- an interrupt vector of 200; this declaration asserts that
   -- information to the Kernel
   --

interrupt_management.bind_interrupt_handler
(
   interrupt    => IO_device,
   handler_code =>
      hardware_interface.hw_address (receiver_interrupt_handler'address),
   can_preempt  => false
);
```

Conditions for Blocking

This procedure does not block.

4.10.2.2. Disable

Disable causes the Kernel to ignore all subsequent interrupts from the specified device by not calling its associated interrupt handler.

Invocation

```
interrupt_management.disable
(
   interrupt => IO_device
);
```

Conditions for Blocking

This procedure does not block.

4.10.2.3. Enable

Enable is used both initially to enable an interrupt and to re-enable an interrupt after it has been disabled. When a specified interrupt is enabled and the interrupt occurs, the bound interrupt handler is called.

Invocation

```
interrupt_management.enable
(
   interrupt => IO_device
);
```

Conditions for Blocking

This procedure does not block.

4.10.2.4. Enabled

Enabled is used to query the status of an interrupt.

Invocation

```
device_status := interrupt_management.enabled (interrupt => IO_device);
```

Conditions for Blocking

This procedure does not block.

4.10.2.5. Simulate_Interrupt

Simulate_interrupt calls the handler for an interrupt as if the associated hardware interrupt had occurred. An interrupt must be bound to a handler and enabled to simulate the interrupt. Calling *simulate_interrupt* may or may not return to the invoking process, depending on the type of the interrupt being simulated. If the interrupt being simulated is a *non-preemptive* interrupt, then control is resumed in the invoking Kernel process immediately after the call to *simulate_interrupt.* However, if the interrupt being simulated is *preemptive*, control returns to the Kernel Scheduler, which might elect to suspend the invoking process and resume another.

Invocation

```
interrupt_management.simulate_interrupt
(
   interrupt => IO_device
);
```

Conditions for Blocking

This procedure does not block.

4.10.3. Related Information

Any of these subprograms may be invoked from an interrupt service routine.

4.10.3.1. Exported Constants

Null_handler
The address of a null interrupt service routine
Set via a *tailoring parameter; see Section* C.1.1

There are no representation specification specifications relevant to any of these constants other than those of their base types.

4.10.3.2. Exported Types

More details are provided in Section 5.2.4.

Interrupt_condition
Indication of whether a handler is bound to an interrupt or not
Enumerated (*bound, unbound*)

Interrupt_name
Range corresponding to hardware vector assignments of the target
See Appendix H for actual range of values for the 68020

Interrupt_owner

Indication of how the interrupt is used

Enumerated (*absent, reserved, Kernel, application*)

Interrupt_state

Indication of whether enabled or disabled

Enumerated (*enabled, disabled*)

Interrupt_source

Origin of the interrupt

Enumerated (*internal, external*)

Interrupt_table_entry

Type to construct Interrupt Table

Record; see Section 5.2.4

There are no representation specifications relevant to any of these types other than those of their base types.

4.10.3.3. Exported Data Structures

Interrupt_table

Logical constant used to maintain all the information known to the Kernel about all interrupts

See Section 5.2.4

There are no representation specifications relevant to the Interrupt Table at this level.

The following data structure is exported by package *interrupt_globals* but is only used internally within the Kernel:

Interrupt_vector

The transfer vector actually used by the Kernel, a logical subset of the *interrupt_table*

See Section 5.2.4

There are no representation specifications relevant to the *interrupt_vector* at this level.

4.10.3.4. Referenced Constants

None.

4.10.3.5. Referenced Types

None.

4.10.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others:

1. *Number_of_interrupt_names_used_by_application* - see Section C.2.4.
2. *Number_of_interrupt_names_used_by_Kernel* - see Section C.2.4.

4.11. Semaphore Management

4.11.1. Introduction

The semaphore management capability comprises the following packages:

1. *Generic_semaphore_management, semaphore_management*

4.11.1.1. Purpose

The *semaphore_management* package provides the abstraction of ("Dykstra") Boolean semaphores to an application. As described in Section 5.1.2, a *semaphore* consists of three components: a count of the number of processes waiting to access the semaphore, a FIFO queue of the waiting processes, and a last-in, first-out (LIFO) list of semaphores claimed by the process that owns this one. A Kernel process may register a request to reserve a semaphore, thus asserting sole ownership of the resource it guards, and may give up that shared resource.

4.11.1.2. Mechanism

This mutual exclusion of Kernel processes from concurrently accessing the same resource is accomplished via the data type *semaphore*, described in Section 5.1.2, and two subprograms to reserve and give up the semaphore: *claim* and *release* respectively.

If a process *claims* a *semaphore*, that process owns the *semaphore*, and any subsequent process is blocked on the *claim* request until the owning process *releases* the *semaphore*.

4.11.2. Subprograms

4.11.2.1. Claim

This primitive attempts to claim the specified *semaphore*. If the *semaphore* is *free*, the primitive succeeds and the invoking process continues execution. If the *semaphore* is not *free*, the invoking process may be inserted into the *semaphore*'s waiting queue and block until the *semaphore* becomes *free*. *Claim* takes the following parameters:

1. The *semaphore_name* to be claimed.
2. An optional timeout of one of two kinds:
   - *Within_elapsed_time* - a relative time after which the *claim* request is rescinded; or
   - *By_epoch_time* - an absolute time at which the *claim* request is rescinded.

   A timeout of zero or some previous time prevents the calling process from blocking; if the semaphore is not available, the claiming process does not wait for it to become available.
3. An optional *resumption_priority* to take effect when the claiming process becomes unblocked.

Invocation

There are three forms of this primitive: one for an infinite timeout (i.e., there is not timeout parameter); one for an *elapsed_time* timeout; one for an *epoch_time* timeout.

```
semaphore_management.claim
```

```
(
   semaphore_name => global_resources.position_data
);

semaphore_management.claim
(
   semaphore_name      => global_resources.position_data,
   within_elapsed_time => five_seconds,
   resumption_priority => 2
);

semaphore_management.claim
(
   semaphore_name      => global_resources.position_data,
   by_epoch_time       => time_globals.base_time + five_seconds,
   resumption_priority => 1
);
```

Conditions for Blocking

This procedure blocks only when the state of the requested semaphore is CLAIMED (N) and the timeout parameter is for some future time. It will unblock when one of the following conditions occurs:

1. The semaphore is released and the invoking process is at the head of the wait queue, or
2. The claim timeout expires.

4.11.2.2. Release

This primitive releases a semaphore previously claimed. If there are no further waiting processes, then the *semaphore* becomes *free*; if there are waiting processes, then the number of waiting processes is decremented by one and the *semaphore* is given to the process at the head of the *semaphore* queue.

Invocation

```
semaphore_management.release
(
   semaphore_name => global_resources.position_data
);
```

Conditions for Blocking

This procedure does not block.

4.11.3. Related Information

None of these subprograms may be invoked from an interrupt service routine.

4.11.3.1. Exported Constants
None.

4.11.3.2. Exported Types
None.

4.11.3.3. Exported Data Structures
None.

4.11.3.4. Referenced Constants

1. *Current_process_priority* - used for parameters; see Section 4.3

4.11.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2
2. *Epoch_time* - used for parameters; see Section 4.2
3. *Priority* - used for parameters; see Section 4.3
4. *Semaphore* - used for parameters; see Section 5.1.2

4.11.3.6. Relevant Generic Parameters
Error checking: see Section C.4.

Others: none.

4.12. Alarm Management

4.12.1. Introduction
The alarm management capability comprises the following packages:

1. *Generic_alarm_management, alarm_management*

4.12.1.1. Purpose
Alarms are:

- Enforced changes in process state.
- Caused by the expiration of a timeout.
- Asynchronous events that are allocated on a per-process basis.

A process views an alarm as a possible change in priority with an enforced transfer of control to an exception handler. An alarm is requested to expire at some specified time in the future. When an alarm expires, the Kernel raises the *alarm_expired* exception, which the process is expected to handle as appropriate. There is only a single alarm per process.

4.12.1.2. Mechanism

The *alarm_management* package exports an exception *alarm_expired* that is raised in the process by the Kernel when the specified timeout expires.

The *set_alarm* Kernel primitive defines the timeout at which point the Kernel raises *alarm_expired* if the alarm is not cancelled via the *cancel_alarm* primitive.

Alarm_expired Exception

When the Kernel raises the *alarm_expired* exception, this indicates that the timeout specified via the *set_alarm* Kernel primitive has, in fact, expired. It is intended that the Kernel process setting the alarm also provides a mechanism to handle the possible occurrence of the exception. This could be used as a paradigm to create cyclic processes and handle frame overruns. See Appendix E.3 for an example.

As *alarm_expired* is like any Ada exception, it may be handled in an exception handler, either explicitly or by a *when others* clause. If not handled, it is propagated like any other Ada exception.

4.12.2. Subprograms

4.12.2.1. Set_alarm

This primitive defines an alarm that interrupts the invoking Kernel process if it expires. This primitive may be invoked by any Kernel process any time after initialization. The interruption causes the process to become suspended in an error state; when the process is resumed, the *alarm_expired* exception is raised in it.

The expiration of the alarm is expressed as a timeout of one of two kinds:

- *After_elapsed_time* - a relative time after which the Kernel readies the *alarm_expired* exception to be raised; or
- *For_epoch_time* - an absolute time at which the Kernel readies the *alarm_expired* exception to be raised.

If the expiration time of the alarm is the current time or in the past, the timeout expires immediately, and *alarm_expired* is immediately raised.

An optional *expiration_priority* to take effect when the invoking process begins processing the exception handler for *alarm_expired* may also be specified.

Invocation

There are two forms of this primitive: one to set an alarm after an *elapsed_time*; one to set an alarm for an *epoch_time*.

```
alarm_management.set_alarm
(
   after_elapsed_time  => five_seconds
);

alarm_management.set_alarm
```

```
(
   for_epoch_time       => time_globals.base_time + five_seconds,
   expiration_priority => 7
);
```

Conditions for Blocking

This procedure does not block.

4.12.2.2. Cancel_alarm

This primitive disables an alarm that was set but has not yet expired. This primitive may be invoked by any Kernel process any time after initialization.

Invocation

```
alarm_management.cancel_alarm;
```

Conditions for Blocking

This procedure does not block.

4.12.3. Related Information

None of these subprograms may be invoked from an interrupt service routine.

4.12.3.1. Exported Constants

None.

4.12.3.2. Exported Types

None.

4.12.3.3. Exported Data Structures

None.

4.12.3.4. Referenced Constants

1. *Current_process_priority* - used for parameters; see Section 4.3

4.12.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2
2. *Epoch_time* - used for parameters; see Section 4.2
3. *Priority* - used for parameters; see Section 4.3

4.12.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others: none.

4.13. Time Management

The abstraction of time permeates the entire Kernel. Time comes in two kinds: relative time (the Kernel-exported type *elapsed_time*) and absolute time (the Kernel-exported type *epoch_time*). All Kernel primitives that may block take a timeout parameter of either kind (for *wait*, see Section 4.8.2.5, for *claim*, see Section 4.11.2.1, for *send_message_and_wait*, see Section 4.7.2.2, for *receive_message*, see Section 4.7.2.3, for *synchronize*, see Section 4.13.2.3). Alarms may be set to expire in terms of both kinds of time (see Section 4.12.2.1). The Kernel's overall view of time is described in Section 4.2.

4.13.1. Introduction

The time management capability comprises the following packages:

1. *Generic_time_management, time_management*

4.13.1.1. Purpose

The *time_management* package exports subprograms to: adjust the elapsed time counter, adjust the epoch time counter, synchronize both elapsed and epoch times across the entire Kernel network, and obtain the time that has elapsed since the Kernel on this processor initialized.

4.13.1.2. Mechanism

The Kernel primitive *adjust_elapsed_time* is provided to adjust the elapsed time counter. This is to be used when one processor's local clock has drifted. This has the effect of changing pending delays of either kind, since increasing the number of elapsed ticks makes the machine think both that it has been running longer and that it is later in the day.

The Kernel primitive *adjust_epoch_time* is provided to adjust the epoch time. This is to be used if it is discovered that the original time setting was incorrect or needs to be adjusted (e.g., if the application needs to take daylight savings time into account). This has the effect of changing any pending delay-until actions, since increasing the epoch makes the machine think it is later in the day but does not change how long it has been running.

The Kernel primitive *synchronize* is provided to synchronize time across the entire network (both elapsed and epoch time).

The Kernel primitive *read_clock* is provided to obtain the elapsed time on the invoking processor.

4.13.2. Subprograms

4.13.2.1. Adjust_elapsed_time

This primitive allows the application to increment or decrement the current local elapsed time by a specified amount. This primitive may be invoked by any Kernel process any time after initialization. This affects pending delays expressed in terms of both *elapsed_time* and *epoch_time.*

Invocation

```
time_management.adjust_elapsed_time
(
   adjustment => five_seconds
);
```

Conditions for Blocking

This procedure does not block.

4.13.2.2. Adjust_epoch_time

This primitive allows the application to increment or decrement the current local epoch time by a specified amount. This primitive may be invoked by any Kernel process any time after initialization. This affects pending delays expressed only in terms of *epoch_time*.

Invocation

```
time_management.adjust_epoch_time
(
   new_epoch_time => time_globals.base_time + one_hour
);
```

Conditions for Blocking

This procedure does not block.

4.13.2.3. Synchronize

This primitive forces all local processor clocks on Kernel devices to synchronize time with the local clock on the invoking processor. This primitive may be invoked by any Kernel process any time after initialization.

A timeout of one of two kinds is specified:

- *Timeout_after* - a relative time before which the Kernel guarantees that the clocks will be synchronized; or
- *Timeout_at* - an absolute time at which the Kernel guarantees that the clocks will be synchronized.

A timeout of zero or some previous time prevents the calling process from blocking; the synchronization does not occur.

An optional *resumption_priority* that takes effect when the invoking process becomes unblocked may also be specified.

The postconditions of this primitive are:

- If it completes successfully, all clocks are synchronized.
- If it terminates with an error, the exact state of network time is not known.

Invocation

There are two forms of this primitive: one for an *elapsed_time* timeout; one for an *epoch_time* timeout.

```
time_management.synchronize
(
   timeout_after => five_seconds
);

time_management.synchronize
(
   timeout_at          => time_globals.base_time + five_seconds
   resumption_priority => 1
);
```

Conditions for Blocking

This procedure does not block.

4.13.2.4. Read_clock

This primitive reads the local processor clock and returns the elapsed time. This primitive may be invoked by any Kernel process any time after initialization.

Invocation

```
running_time := time_management.read_clock;
```

Conditions for Blocking

This procedure does not block.

4.13.3. Related Information

Any of these subprograms may be invoked from an interrupt service routine.

4.13.3.1. Exported Constants

None.

4.13.3.2. Exported Types

None.

4.13.3.3. Exported Data Structures

None.

4.13.3.4. Referenced Constants

1. *Current_process_priority* - used for parameters; see Section 4.3

4.13.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2
2. *Epoch_time* - used for parameters; see Section 4.2
3. *Priority* - used for parameters; see Section 4.3

4.13.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others: none.

4.14. Timeslice Management

4.14.1. Introduction

The timeslice management capability comprises the following packages:

1. *Generic_timeslice_management, timeslice_management*

4.14.1.1. Purpose

The *timeslice_management* package supports round-robin execution of processes of the same priority. Processes voluntarily indicate whether or not they participate in timeslicing by enabling or disabling their preemption (see Sections 4.6.2.2 and 4.8.2.3); by default, processes do participate.

4.14.1.2. Mechanism

Package *timeslice_management* provides the capability to set the processor-wide timeslice quantum via the Kernel primitive *set_timeslice*, and the capability to enable and disable round-robin execution of processes at the same priority level via Kernel primitives *enable_time_slicing* and *disable_time_slicing* respectively.

Timeslice Model

The model used by the Kernel is that processes of equal priority that are participating in timeslice processing are allocated a quantum of time in which to execute. The length of this timeslice quantum is set via *set_timeslice*. If a timesliced process blocks, it relinquishes the rest of its timeslice, allowing another process at the same priority level and also participating in timeslice processing to execute. If an interrupt occurs, however, that time is included as part of the process's allocated timeslice quantum.

4.14.2. Subprograms

4.14.2.1. Disable_time_slicing

This primitive disables round-robin, timeslice scheduling. This primitive may be called by any Kernel process at any time after initialization. After execution of this primitive, scheduling is purely priority-based preemption. Appendix D provides the details of this algorithm.

Invocation

```
timeslice_management.disable_time_slicing;
```

Conditions for Blocking

This procedure does not block.

4.14.2.2. Enable_time_slicing

This primitive enables round-robin, timeslice scheduling among processes of equal priority. This primitive may be called by any Kernel process at any time after initialization. After execution of this primitive, scheduling mingles the pure priority preemption scheduling with timeslice scheduling. Appendix D provides the details of this algorithm.

Invocation

```
timeslice_management.enable_time_slicing;
```

Conditions for Blocking

This procedure does not block.

4.14.2.3. Set_timeslice

This primitive sets the timeslice quantum for the processor. This primitive may be called by any Kernel process at any time after initialization.

There is no Kernel-imposed limitation on the maximum length of a timeslice quantum; however, there is a Kernel-imposed minimum. This is set via the tailoring parameter *minimum_slice_time_value*. The actual, reasonable value to use as the parameter to *set_timeslice* requires analysis of the application requirements, as well as the Kernel's performance. See Section C.2.3 for an illustration of the analysis required.

Invocation

```
timeslice_management.set_timeslice
(
   quantum => time_globals.milliseconds (100)
);
```

Conditions for Blocking

This procedure does not block.

4.14.3. Related Information

Any of these subprograms may be invoked from an interrupt service routine.

4.14.3.1. Exported Constants

Minimum_slice_time

The minimum amount of time that may be specified as a timeslice interval

Set via a tailoring parameter; see Section C.2.3

There are no representation specifications relevant to any of these constants other than those of their base types.

4.14.3.2. Exported Types

None.

4.14.3.3. Exported Data Structures

None.

4.14.3.4. Referenced Constants

None.

4.14.3.5. Referenced Types

1. *Elapsed_time* - used for parameters; see Section 4.2

4.14.3.6. Relevant Generic Parameters

Error checking: see Section C.4.

Others: none.

4.15. Index of Kernel Names

This section contains two tables providing a quick index to Kernel names. The first table is sorted alphabetically by name (of subprogram, exception, type, constant, data structure), and for each name, the exporting package is given, along with the section in this document where more information may be found. The second table contains the same information sorted by exporting package first, then the names exported by that package, and the Kernel User's Manual section reference.

Index of Kernel Names		
Name	Exporting Package	Section
adjust_elapsed_time	time_management	4.13
adjust_epoch_time	time_management	4.13
alarm_expired	alarm_management	4.12
allocate_device_receiver	communication_management	4.7
arithmetic operators: "+," "-," "*," "/"	time_globals	4.2
base_time	time_globals	4.2
bind_interrupt_handler	interrupt_management	4.10
bits_per_byte	hardware_interface	4.1

Index of Kernel Names

Index of Kernel Names

Index of Kernel Names

Index of Kernel Names

This index of Kernel names is sorted by the exporting package, then the names exported by the package, and the Kernel User's Manual section reference.

Index of Kernel Exporting Packages		
Exporting Package	Name	Section
alarm_management	alarm_expired	4.12
alarm_management	cancel_alarm	4.12
alarm_management	set_alarm	4.12
communication_globals	maximum_message_length	4.7
communication_globals	message_length_type	4.7
communication_globals	message_tag_type	4.7
communication_management	allocate_device_receiver	4.7
communication_management	receive_message	4.7
communication_management	send_message	4.7
communication_management	send_message_and_wait	4.7
hardware_interface	bits_per_byte	4.1
hardware_interface	byte	4.1
hardware_interface	hw_address	4.1
hardware_interface	hw_bits8	4.1
hardware_interface	hw_bits8_ptr	4.1
hardware_interface	hw_byte	4.1
hardware_interface	hw_byte_ptr	4.1
hardware_interface	hw_duration	4.1
hardware_interface	hw_integer	4.1
hardware_interface	hw_long_integer	4.1
hardware_interface	hw_long_natural	4.1
hardware_interface	hw_long_positive	4.1
hardware_interface	hw_natural	4.1
hardware_interface	hw_positive	4.1
hardware_interface	hw_string	4.1
hardware_interface	longword	4.1
hardware_interface	null_hw_address	4.1
hardware_interface	to_hw_address	4.1
hardware_interface	to_hw_bits8	4.1
hardware_interface	to_hw_bits8_ptr	4.1
hardware_interface	to_hw_byte_ptr	4.1
hardware_interface	word	4.1

Index of Kernel Exporting Packages

Index of Kernel Exporting Packages

Index of Kernel Exporting Packages		
Exporting Package	Name	Section
time_globals	to_Kernel_time	4.2
time_globals	to_elapsed_time	4.2
time_globals	to_epoch_time	4.2
time_globals	zero_elapsed_time	4.2
time_globals	zero_epoch_time	4.2
time_management	adjust_elapsed_time	4.13
time_management	adjust_epoch_time	4.13
time_management	read_clock	4.13
time_management	synchronize	4.13
timeslice_management	disable_time_slicing	4.14
timeslice_management	enable_time_slicing	4.14
timeslice_management	minimum_slice_time	4.14
timeslice_management	set_timeslice	4.14

4.16. Summary of Example

This section expands the example that was begun in Chapter 3. Each of the Kernel primitives in Chapter 4 presented an example invocation of the appropriate Kernel primitive. These calls have been collected into the bodies of processes **Merlin**, **Arthur**, and **Vivian**. When this example is completed in the next version of this document, these bodies will be documented to indicate the behavior of the three processes when executing on the Kernel and, thus, may be used as one sample test case for ensuring correct behavior of the Kernel after installation.

```
procedure make_NCT;

with hardware_interface;
with network_configuration;
with process_table;
procedure make_NCT is

begin

   network_configuration.NCT :=
   (
      (logical_name            => "processor a     ",
       physical_address        => 16#01#,
       Kernel_device           => true,
       needed_to_run           => true,
       allocated_process_ID    => process_table.null_process,
       initialization_order    => 1,
       initialization_complete => false
      ),
      (logical_name            => "processor b     ",
       physical_address        => 16#02#,
       Kernel_device           => true,
       needed_to_run           => true,
       allocated_process_ID    => process_table.null_process,
       initialization_order    => 2,
       initialization_complete => false
      ),
      (logical_name            => "device          ",
       physical_address        => 16#03#,
       Kernel_device           => false,
       needed_to_run           => false,
       allocated_process_ID    => process_table.null_process,
       initialization_order    => 3,
       initialization_complete => false
      )
   );
end make_NCT;

-------------------------------------------------------------------

with time_globals;
package timeouts is

   function "+" (left, right : time_globals.elapsed_time)
      return time_globals.elapsed_time
      renames time_globals."+";

   Master_base_time : constant time_globals.epoch_time :=
      time_globals.create_epoch_time
      (
         day => 0,
         second => 0.0
      );

   Master_timeout : constant time_globals.elapsed_time :=
      time_globals.create_elapsed_time
      (
         day => 0,
         second => 5.0
      );

   subordinate_timeout : constant time_globals.elapsed_time :=
      time_globals.seconds
      (
         an_integral_duration => 5
      );

   init_complete_timeout : constant time_globals.elapsed_time :=
      Master_timeout + subordinate_timeout;

end timeouts;

----------------------------------------------------------------

with hardware_interface;
with process_managers_globals;
package application_unique_names is

   arthur : process_managers_globals.process_name_type :=
      "arthur                          ";

   device : process_managers_globals.device_name_type :=
      "device                          ";

   merlin : process_managers_globals.process_name_type :=
      "merlin                          ";

   vivian : process_managers_globals.process_name_type :=
      "vivian                          ";

end application_unique_names;

----------------------------------------------------------------

with communication_globals;
with hardware_interface;
with process_table;
package processor_a_comm_area is

   merlin_ID : process_table.process_identifier;
   arthur_ID : process_table.process_identifier;
   vivian_ID : process_table.process_identifier;

   type_1_message_tag : constant
      communication_globals.message_tag_type := 1;
   type_1_message_length : constant
      communication_globals.message_length_type := 10;
   type_1_message_text : constant
      hardware_interface.hw_string
         (1 .. positive (type_1_message_length)) :=
            "type 1 msg";
   type_2_message_tag : constant
      communication_globals.message_tag_type := 2;
   type_2_message_length : constant
      communication_globals.message_length_type := 20;
   type_2_message_text : constant
      hardware_interface.hw_string
         (1 .. positive (type_2_message_length)) :=
            "type 2 message      ";
   type_2a_message_text : constant
      hardware_interface.hw_string
         (1 .. positive (type_2_message_length)) :=
            "type 2a message     ";
   type_2b_message_text : constant
      hardware_interface.hw_string
         (1 .. positive (type_2_message_length)) :=
            "type 2b message     ";

   arthur_max_incoming_message_length : constant positive := 100;
   merlin_max_outgoing_message_length : constant positive := 100;

end processor_a_comm_area;

-----------------------------------------------------------------

with communication_globals;
with communication_management;
with hardware_interface;
with process_table;
with time_globals;
with processor_a_comm_area;
procedure arthur_process_code is

   local_receive_buffer : hardware_interface.hw_string
   (
      1 .. processor_a_comm_area.arthur_max_incoming_message_length
   ) := (others => ' ');

   local_length : communication_globals.message_length_type;
   local_messages_lost : Boolean := false;
   local_sender : process_table.process_identifier;
   local_tag : communication_globals.message_tag_type;

begin
   -- do arthur's algorithm

   receive_loop:
   for i in 1 .. 4 loop
      case i is
         when 1 | 4 =>
            communication_management.receive_message
            (
               sender                  => local_sender,
               message_tag             => local_tag,
               message_length          => local_length,
               message_buffer          => hardware_interface.hw_address
                  (local_receive_buffer'address),
               buffer_size         =>
                  communication_globals.message_length_type
                  (processor_a_comm_area.
                   arthur_max_incoming_message_length),
               resumption_priority => 3,
               messages_lost       => local_messages_lost
            );

            local_receive_buffer := (others => ' ');

         when 2 =>
            communication_management.receive_message
            (
               sender                => local_sender,
               message_tag           => local_tag,
               message_length        => local_length,
               message_buffer        => hardware_interface.hw_address
                  (local_receive_buffer'address),
               buffer_size           =>
                  communication_globals.message_length_type
                  (processor_a_comm_area.
                   arthur_max_incoming_message_length),
               timeout_after         => time_globals.milliseconds (1_000),
               messages_lost         => local_messages_lost
            );

            local_receive_buffer := (others => ' ');

         when 3 =>
            communication_management.receive_message
            (
               sender                => local_sender,
               message_tag           => local_tag,
               message_length        => local_length,
               message_buffer        => hardware_interface.hw_address
                  (local_receive_buffer'address),
               buffer_size           =>
                  communication_globals.message_length_type
                  (processor_a_comm_area.
                   arthur_max_incoming_message_length),
               timeout_at            =>
                  time_globals.create_epoch_time (0, 1_000.0),
               messages_lost         => local_messages_lost
            );

            local_receive_buffer := (others => ' ');

      end case;

      local_receive_buffer := (others => ' ');

   end loop receive_loop;

   null;
end arthur_process_code;
```

```
with communication_management;
with hardware_interface;
with time_globals;
with processor_a_comm_area;
procedure merlin_process_code is

   dummy : hardware_interface.hw_long_integer :=
      hardware_interface.hw_long_integer'first;

   function "+" (left, right : hardware_interface.hw_long_integer)
      return hardware_interface.hw_long_integer
      renames hardware_interface."+";

   local_outgoing_message_buffer : hardware_interface.hw_string
   (
      1 .. processor_a_comm_area.merlin_max_outgoing_message_length
   ) := (others => ' ');

begin
   -- do merlin's algorithm

   busy_wait_to_let_vivian_pend_on_receive:
   for i in 1 .. 10_000 loop
      for j in 1 .. 30_000 loop
         dummy := dummy + 1;
      end loop;
   end loop busy_wait_to_let_vivian_pend_on_receive;

   ------------

   local_outgoing_message_buffer
   (1 .. positive (processor_a_comm_area.type_1_message_length)) :=
      processor_a_comm_area.type_1_message_text;

   communication_management.send_message
   (
      receiver       => processor_a_comm_area.vivian_ID,
      message_tag    => processor_a_comm_area.type_1_message_tag,
      message_length => processor_a_comm_area.type_1_message_length,
      message_text   =>
         hardware_interface.hw_address
            (local_outgoing_message_buffer'address)
   );

   communication_management.send_message
   (
      receiver       => processor_a_comm_area.arthur_ID,
      message_tag    => processor_a_comm_area.type_1_message_tag,
      message_length => processor_a_comm_area.type_1_message_length,
      message_text   =>
         hardware_interface.hw_address
            (local_outgoing_message_buffer'address)
   );

   local_outgoing_message_buffer := (others => ' ');
```

```
------------

local_outgoing_message_buffer
(1 .. positive (processor_a_comm_area.type_2_message_length)) :=
   processor_a_comm_area.type_2_message_text;

communication_management.send_message_and_wait
(
   receiver             => processor_a_comm_area.arthur_ID,
   message_tag          => processor_a_comm_area.type_2_message_tag,
   message_length       => processor_a_comm_area.type_2_message_length,
   message_text         =>
      hardware_interface.hw_address
         (local_outgoing_message_buffer'address),
   resumption_priority => 2
);

local_outgoing_message_buffer := (others => ' ');

------------

local_outgoing_message_buffer
(1 .. positive (processor_a_comm_area.type_2_message_length)) :=
   processor_a_comm_area.type_2a_message_text;
communication_management.send_message_and_wait
(
   receiver             => processor_a_comm_area.arthur_ID,
   message_tag          => processor_a_comm_area.type_2_message_tag,
   message_length       => processor_a_comm_area.type_2_message_length,
   message_text         =>
      hardware_interface.hw_address
         (local_outgoing_message_buffer'address),
   timeout_after        => time_globals.milliseconds (100)
);

local_outgoing_message_buffer := (others => ' ');

------------

local_outgoing_message_buffer
(1 .. positive (processor_a_comm_area.type_2_message_length)) :=
   processor_a_comm_area.type_2b_message_text;

communication_management.send_message_and_wait
(
   receiver             => processor_a_comm_area.arthur_ID,
   message_tag          => processor_a_comm_area.type_2_message_tag,
   message_length       => processor_a_comm_area.type_2_message_length,
   message_text         =>
      hardware_interface.hw_address
         (local_outgoing_message_buffer'address),
   timeout_at           => time_globals.create_epoch_time (0, 0.100),
   resumption_priority => 6
);

local_outgoing_message_buffer := (others => ' ');
```

```
   ------------

   null;
end merlin_process_code;

----------------------------------------------------------------

with hardware_interface;
with process_managers;
with process_managers_globals;
with processor_management;
with schedule_types;
with application_unique_names;
with arthur_process_code;
with merlin_process_code;
with processor_a_comm_area;
with timeouts;
with make_NCT;
procedure processor_a_Main_Unit is
begin

   -- do any processor- and application-specific initialization

   make_NCT;

   processor_management.initialize_Master_processor
   (
      base_epoch    => timeouts.Master_base_time,
      timeout_after => timeouts.Master_timeout
   );

   processor_a_comm_area.merlin_ID :=
      process_managers.declare_process
      (
         application_unique_names.merlin
      );

   processor_a_comm_area.arthur_ID :=
      process_managers.declare_process
      (
         application_unique_names.arthur
      );

   processor_a_comm_area.vivian_ID :=
      process_managers.declare_process
      (
         application_unique_names.vivian
      );

   process_managers.create_process
   (
      process_ID         => processor_a_comm_area.merlin_ID,
      address            =>
         hardware_interface.hw_address (merlin_process_code'address),
      stack_size         => 4_096,
      message_queue_size => 100,
      initial_priority   => schedule_types.highest_priority,
      preemptable        => schedule_types.disabled
   );

   process_managers.create_process
   (
      process_ID         => processor_a_comm_area.arthur_ID,
      address            =>
         hardware_interface.hw_address (arthur_process_code'address),
      stack_size         => 2_048,
      message_queue_size => 10,
      initial_priority   => 4
   );

   -- complete remaining processor- and application-specific initialization

   processor_management.initialization_complete
   (
      timeout_after => timeouts.init_complete_timeout
   );

end processor_a_Main_Unit;

----------------------------------------------------------------

with process_table;
package processor_b_comm_area is

   merlin_ID : process_table.process_identifier;
   vivian_ID : process_table.process_identifier;
   device_ID : process_table.process_identifier;

end processor_b_comm_area;

----------------------------------------------------------------

with processor_b_comm_area;
procedure vivian_process_code is
begin
   -- do vivian's algorithm
   null;
end vivian_process_code;

----------------------------------------------------------------

with communication_management;
with hardware_interface;
with process_managers;
with process_managers_globals;
with processor_management;
with application_unique_names;
with processor_b_comm_area;
with timeouts;
with vivian_process_code;
with make_NCT;
procedure processor_b_Main_Unit is
```

```
begin

   -- do any processor- and application-specific initialization

   make_NCT;

   processor_management.initialize_subordinate_processor
   (
      timeout_after => timeouts.subordinate_timeout
   );

   processor_b_comm_area.merlin_ID :=
      process_managers.declare_process
      (
         application_unique_names.merlin
      );

   processor_b_comm_area.device_ID :=
      process_managers.declare_process
      (
         application_unique_names.device
      );

   processor_b_comm_area.vivian_ID :=
      process_managers.declare_process
      (
         application_unique_names.vivian
      );

   process_managers.create_process
   (
      process_ID         => processor_b_comm_area.vivian_ID,
      address            =>
         hardware_interface.hw_address (vivian_process_code'address),
      stack_size         => 8_096,
      message_queue_size => 1_000,
      initial_priority   => 1
   );

   communication_management.allocate_device_receiver
   (
      receiver_process_ID => processor_b_comm_area.vivian_ID,
      device_ID => 3
   );

   -- complete remaining processor- and application-specific initialization

   processor_management.initialization_complete
   (
      timeout_after => timeouts.init_complete_timeout
   );

end processor_b_Main_Unit;
```

5. Kernel Data Structures

This section describes those Kernel data structures that are accessed directly by the application program, built as part of the initialization process, or consume storage space in response to calls of Kernel primitives. This includes:

- Network Configuration Table (NCT)
- Semaphores
- Process Table
- Datagram Queues
- Time Event Queue
- Process Index Table
- Interrupt Table

For each of these data structures, the following information is presented:

1. The **exporting package** or packages,
2. The details of the **structure** and organization of the data structure (garnered from commentary in the Kernel code and the actual Kernel code definitions themselves, along with schematic figures illustrating the data structures and the scenarios determining that structure),
3. The **initialization** requirements for that data structure, (e.g., whether the application is required to perform it, the Kernel does it automatically),
4. Any **additional allocation requirements**, notably for dynamic data structures, and
5. All **constraints on usage** by the application or the Kernel.

The internal type *Kernel_time* is also presented in this section, as it permeates internal Kernel data structures and functionality, and it provides the basis for the derivation of the abstractions of Kernel time available to the application: *elapsed_time* and *epoch_time*. The importance of determining the suitability of the granularity of this representation in the application domain is introduced in this section and continued in Section C.2.3.

5.1. External Data Structures

5.1.1. Network Configuration Table

The Network Configuration Table (NCT) describes the physical connectivity of each node in the network. The application initializes the NCT, and the Kernel uses that information for network and processor initialization and for inter-processor communication.

5.1.1.1. Exporting Package

Generic_network_configuration, network_configuration

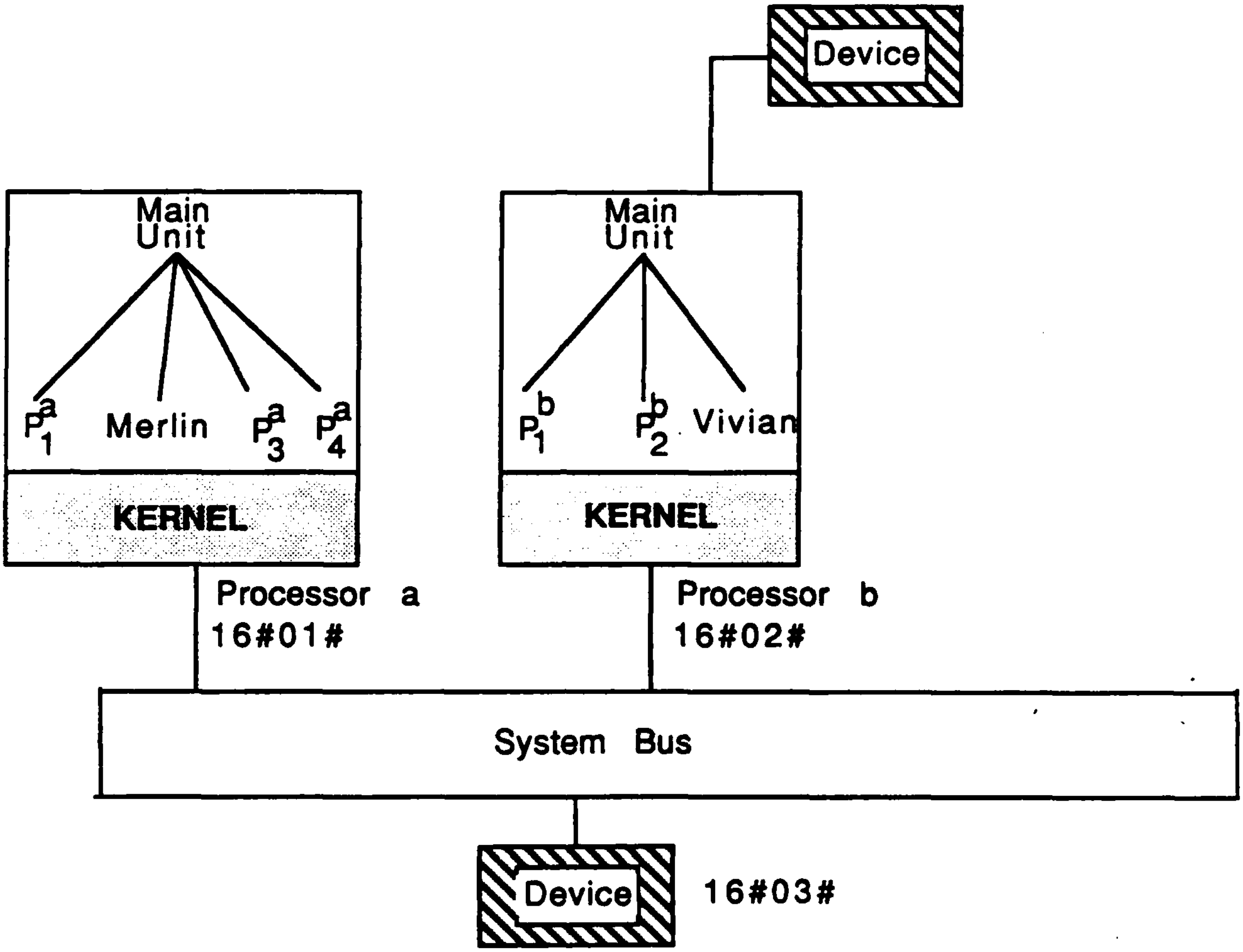
5.1.1.2. Structure

The NCT is a static data structure; it is fully defined at compile time. The NCT is an array of *NCT_entry* records, indexed by a logical *processor_identifier.* Each *NCT_entry* record fully describes the network connectivity information for one node used by the application. All nodes have entries in the NCT: those nodes executing the Kernel and those nodes that are non-Kernel devices.

Figure 5-1 illustrates the structure of the NCT along with the scenario it represents.

```
--
--  logical_name
--    the string-valued name given by the application engineer (this is
--    mapped to the device_ID - which is just an index into the NCT -
--    during initialization; once that is  done, the Kernel refers to
--    the processor by device_ID)
--
--    default value:
--      none
--
--    this value should never change after initialization in Main Unit
--
--  physical_address
--    the actual bus address at which the device is located; this value
--    is used in the datagram packet wrapper to identify the network
--    node that is to receive the packet containing the datagram
--
--    default value:
--      null_address
--
--    this value should never change after initialization in Main Unit
--
--  Kernel_device
--    indication of whether or not the device is also running the Kernel
--    (e.g., whether or not the device responds to the Kernel
--    communication protocols)
--
--    possible values:
--      true (the processor is running the Kernel)
--      false (the processor is not running the Kernel)
--
--    default value:
--      true (the processor is running the Kernel)
--
--    this value should never change after initialization in Main Unit
--
--  needed_to_run
--    indication of whether or not the device must successfully complete
--    the Kernel initialization protocol (i.e., always should be false
--    when Kernel_device is false)
--
--    possible values:
--      true (the device does participate in the initialization
--        protocol)
--      false (the device does not participate in the initialization
```

Scenario: Network and NCT as described in Section 3.1.4, page 40.



Logical Name	Physical Address	Kernel Device	Needed To Run	Allocated Process ID	Initialization Order	Initialization Complete
"processor a"	16#01#	true	true	null_process	1	false
"processor b"	16#02#	true	true	null_process	2	false
"device"	16#03#	false	false	null_process	3	false

Figure 5-1: Network Configuration Table Structure

```
--          protocol)
--
--      default value:
--        true (the device does participate in the initialization
--          protocol)
--
--      this value should never change after initialization in Main Unit
--
--    allocated_process_ID
--      identifier of "surrogate process" allocated to receive messages
--      from the specified non-Kernel device
--
--      default value:
--        null_process
--
--      this value may change via a call to the Kernel primitive:
--        allocate_device_receiver
--
--    initialization_order
--      order in which the processors identified in the NCT are to be
--      initialized
--
--      default value:
--        0 (all NCT entries have the same initial value; initialization
--          proceeds following each entry in the NCT)
--
--      this value should never change after initialization in Main Unit
--
--    initialization_complete
--      indication of whether or not the initialization protocol for this
--      processor has been completed
--
--      possible values:
--        true (initialization has completed)
--        false (initialization has not completed)
--
--      default value:
--        false (initialization has not completed)
--
--      this value is set by the Kernel during initialization and should
--      never change after initialization is complete
--


   type NCT_entry
   is record
      logical_name :
         hw_string (1 .. positive (maximum_length_of_processor_name_value) );
      physical_address : network_globals.bus_address :=
         network_globals.null_address;
      Kernel_device : boolean := true;
      needed_to_run : boolean := true;
      allocated_process_ID : process_table.process_identifier :=
         process_table.null_process;
      initialization_order : hw_natural := 0;
      initialization_complete : boolean := false;
```

```
end record;

type configuration_table is array (
   network_globals.processor_identifier range <>) of NCT_entry;

NCT : configuration_table (network_globals.processor_identifier
   range 1 .. network_globals.processor_identifier (number_of_nodes));
```

There are no representation specifications relevant to the NCT.

5.1.1.3. Initialization

The NCT is completely allocated at application initialization time.

The application initializes the NCT as appropriate for the hardware configuration on which it is to run. Only the following fields are initialized:

1. *Logical_name*
2. *Physical_address*
3. *Kernel_device*
4. *Needed_to_run*
5. *Initialization_order* (optional)

Non-Kernel devices do **not** participate in the Kernel initialization protocol. For each NCT entry where the *Kernel_device* field is false, the *needed_to_run* field must also be false.

If *initialization_order* is specified, that value is used by the Kernel to define an order in which processor nodes are initialized. *Initialization_order* is followed in increasing order, with nodes at the same order value processed in an order determined by the Kernel. The *initialization_order* field for the NCT entry representing the Master processor **must** have a lower value than the *initialization_order* field for all other NCT entries where *needed_to_run* is true. If the *needed_to_run* field is false, the *initialization_order* field is not used.

The Ada Main Unit that configures each processor node requires an NCT. To ensure network-wide consistency, an application could define a single NCT package that is imported into each Main Unit across the entire network.

5.1.1.4. Additional Allocation Requirements

No additional allocation is required. The maximum size of the NCT is constrained by the tailoring parameters: *number_of_nodes_value* (the number of entries in the NCT) and *maximum_length_of_processor_name_value* (the length of the processor name stored in the NCT).

5.1.1.5. Constraints on Usage

The NCT is a read-only data structure to the application. Once initialized by the application, the application should treat the NCT as a constant. If the application modifies any fields within the NCT during execution, correct execution cannot be ensured, as modification would violate the integrity of the Kernel.

5.1.2. Semaphores

The Kernel provides the traditional Boolean ("Dykstra") semaphore facility, slightly revised to be consistent with the overall philosophy of the Kernel primitives.

5.1.2.1. Exporting Package

Generic_process_table, process_table

5.1.2.2. Structure

A *semaphore* is a data object, and as such it can be a component of a larger data structure.

In abstract terms, a *semaphore* consists of two components: a count (*number_of_waiting_processes*) and a queue (*queue_head*). The count records the number of processes waiting to access the semaphore; the waiting processes themselves are enqueued in FIFO order on the queue maintained at *queue_head.* The *semaphore* type does not have any components whose size or characteristics need to be determined at execution time.

The initial state of a *semaphore* is *free*, which is represented by an empty queue and a count value of -1. When a semaphore has been claimed, the queue remains empty, but the count is zero. If another process tries to claim the *semaphore*, the count is incremented, and the process is enqueued.

If a process *claims* a *semaphore*, that process owns the *semaphore*, and any subsequent process is blocked on the *claim* request until the owning process *releases* the *semaphore*.

Figure 5-2 illustrates the actual concrete structure of a *semaphore* along with the scenario it represents.

```
--
-- the information maintained for each semaphore is:
--   number_of_waiting_processes
--     the number of processes in the waiting queue for the semaphore
--
--     default value:
--       -1 (there are no processes waiting and the semaphore is free;
--          the value 0 indicates that the semaphore is claimed and
--          there are no processes waiting for it; any positive value
--          indicates that the semaphore is currently claimed and there
--          are positive value number of processes waiting for the
--          semaphore
--
--     this value is incremented via a call to the Kernel primitive:
--     claim and decremented via a call to the Kernel primitive:  release;
--     otherwise, this value should never change
--
```

Scenario:
The Main Unit on **processor a** declares three semaphores: S1, S2, and S3

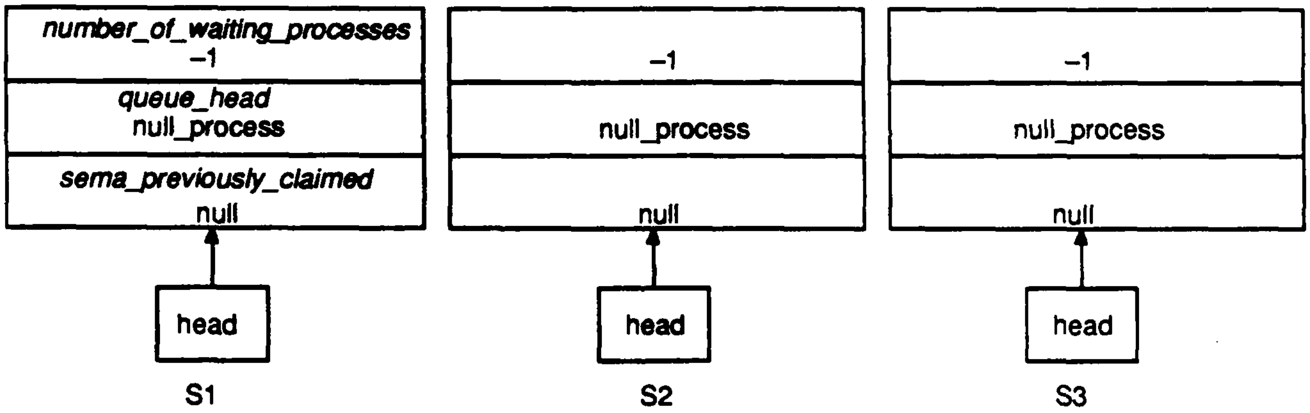


Figure 5-2: Semaphore Structure - Part 1 of 8

Scenario:
Process **Merlin** claims S1

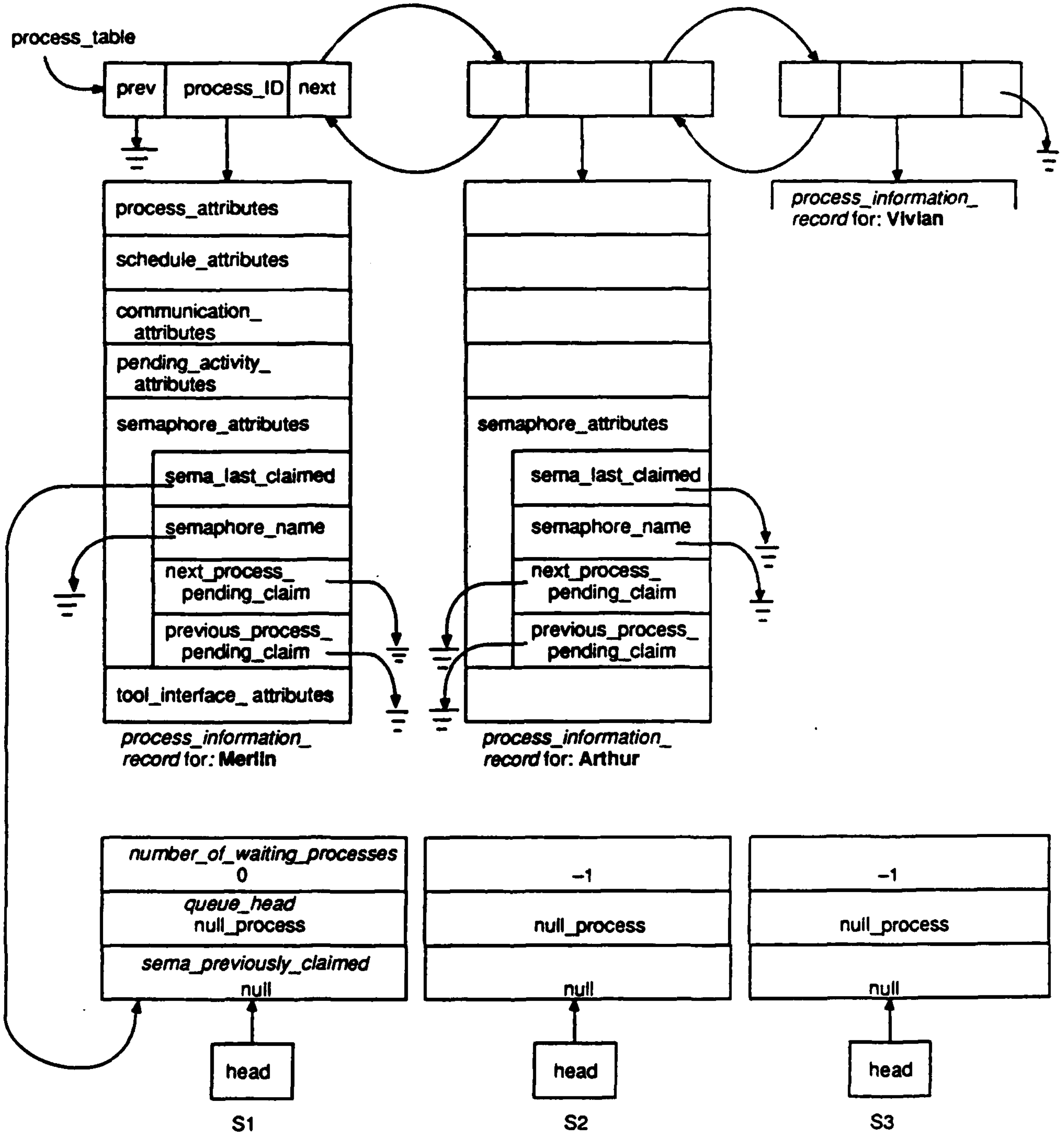

process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
tool_interface_ attributes
process_information_ record for: **Merlin**
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: **Arthur**
process_information_ record for: **Vivian**
number_of_waiting_processes
0
queue_head
null_process
sema_previously_claimed
null
head
S1
-1
null_process
null
head
S2
-1
null_process
null
head
S3

Figure 5-2: Semaphore Structure - Part 2 of 8

Scenario:
Process **Merlin** claims S2 after S1

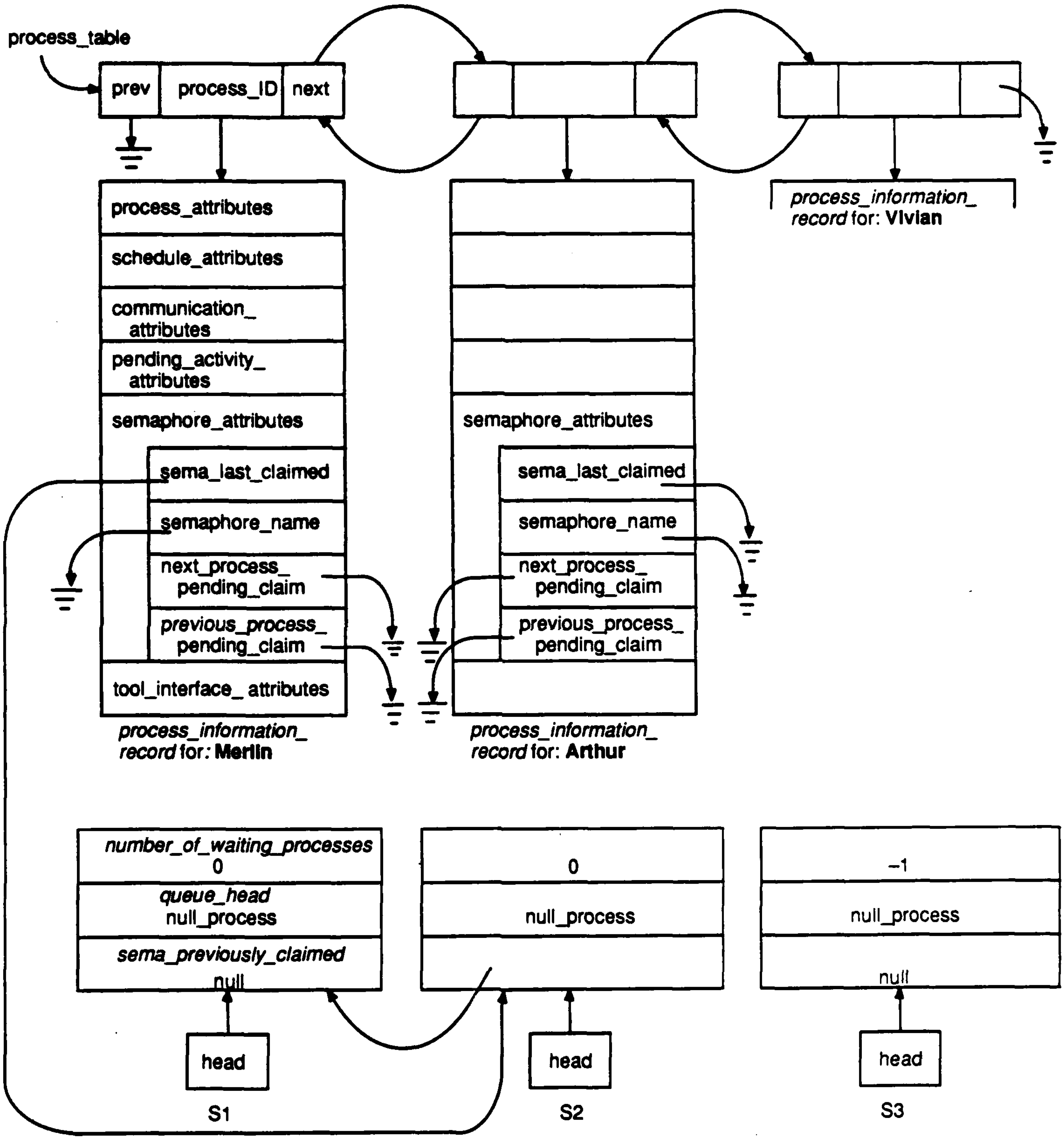

process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
tool_interface_ attributes
process_information_ record for: Merlin
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: Arthur
process_information_ record for: Vivian
number_of_waiting_processes
0
queue_head
null_process
sema_previously_claimed
null
head
S1
0
null_process
head
S2
–1
null_process
null
head
S3

Figure 5-2: Semaphore Structure - Part 3 of 8

Scenario:
Process **Merlin** claims S3 after S2 after S1

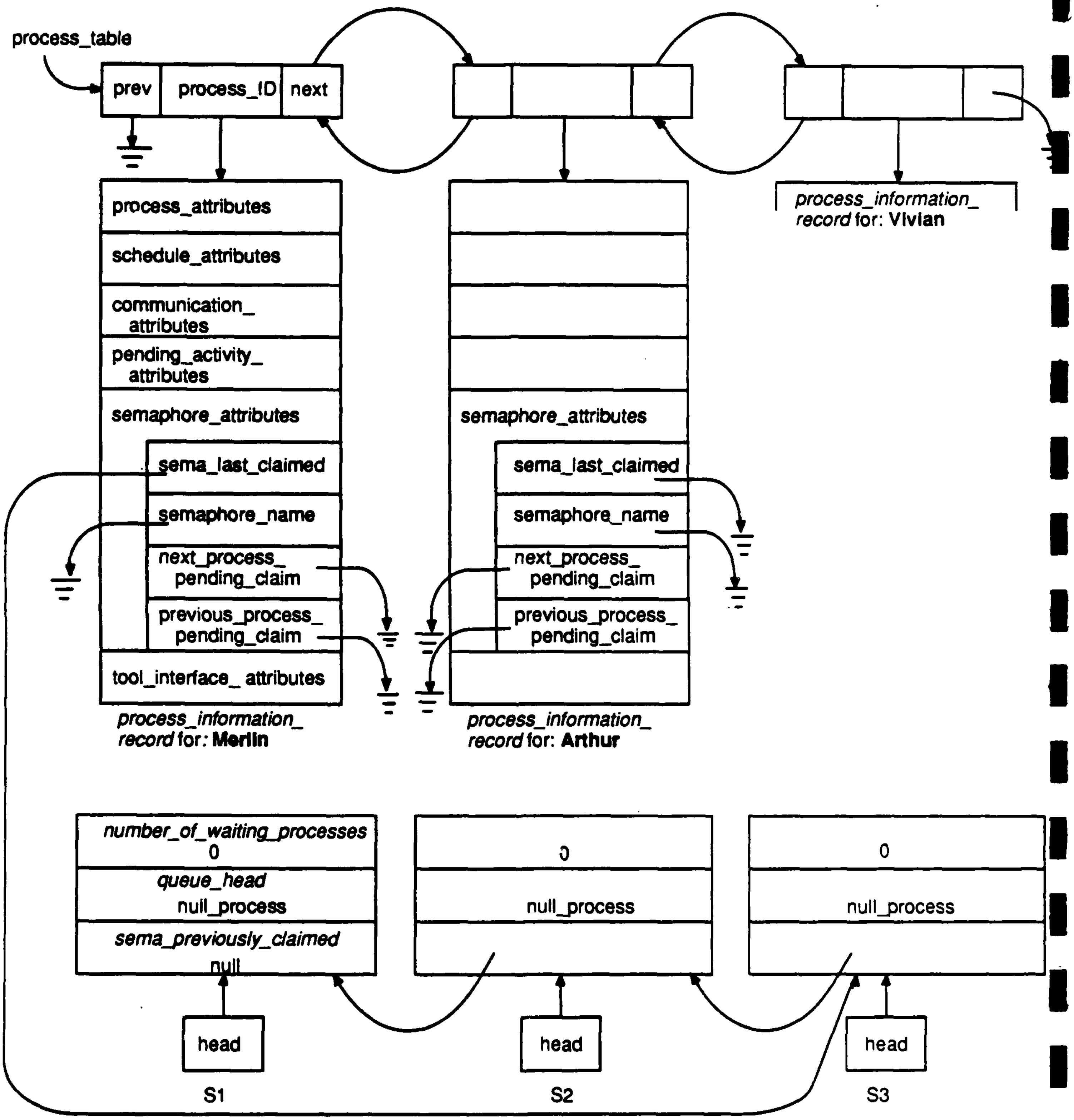
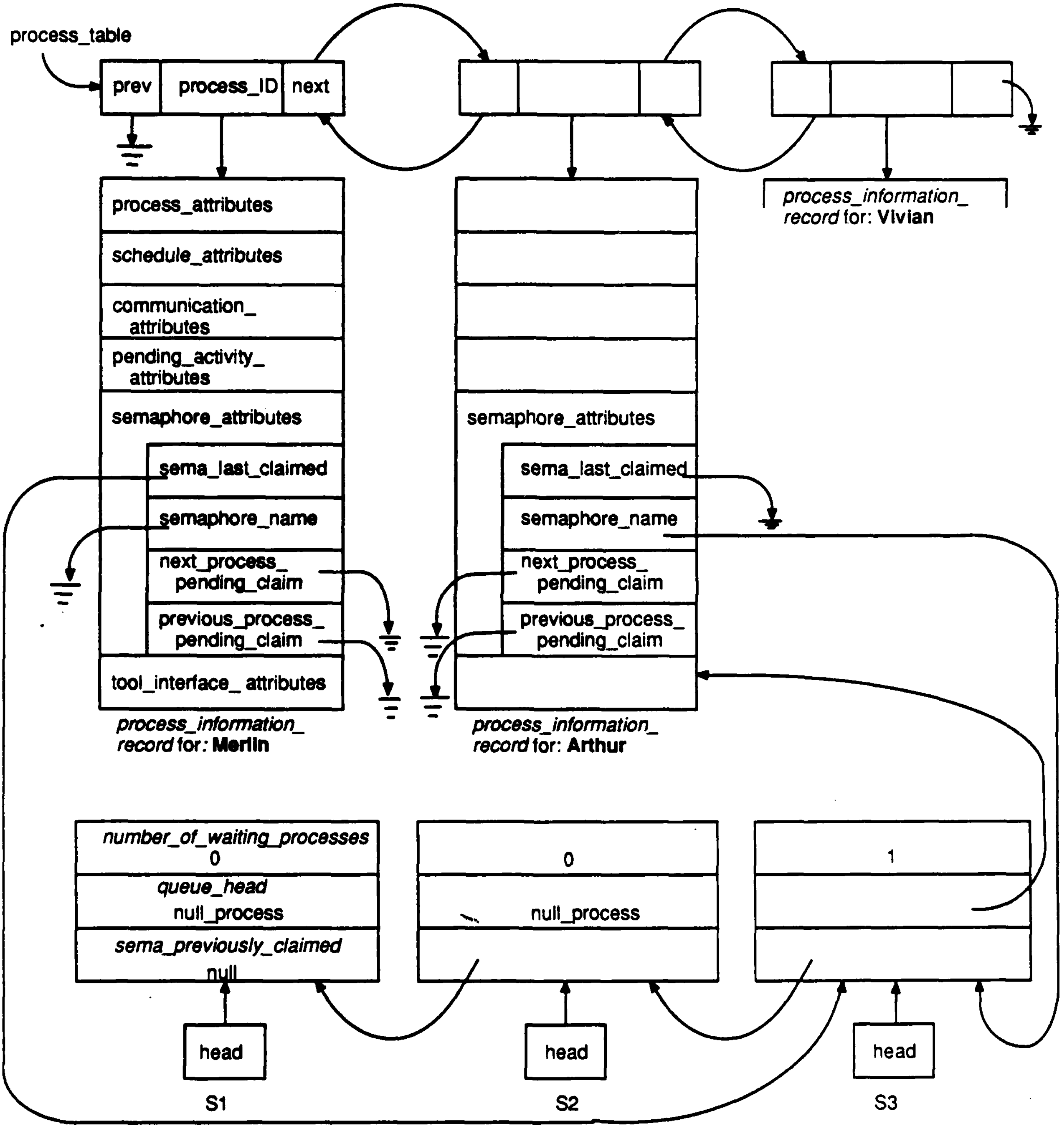

process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
tool_interface_ attributes
process_information_ record for: **Merlin**
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: **Arthur**
process_information_ record for: **Vivian**
number_of_waiting_processes
0
queue_head
null_process
sema_previously_claimed
null
0
null_process
0
null_process
head
S1
head
S2
head
S3

Figure 5-2: Semaphore Structure - Part 4 of 8

Scenario: Process **Arthur** requests to claim S3 after
Process **Merlin** claims S3 after S2 after S1



process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_attributes
pending_activity_attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_pending_claim
previous_process_pending_claim
tool_interface_ attributes
process_information_record for: **Merlin**
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_pending_claim
previous_process_pending_claim
process_information_record for: **Arthur**
process_information_record for: **Vivian**
number_of_waiting_processes
0
queue_head
null_process
sema_previously_claimed
null
0
null_process
1
head
head
head
S1
S2
S3

Figure 5-2: Semaphore Structure - Part 5 of 8

Scenario:
Processor a has three local processes: **Merlin** and **Arthur** (as in all the examples) and **Lancelot** (added to illustrate semaphore waiting queues). **Lancelot** claims S1.

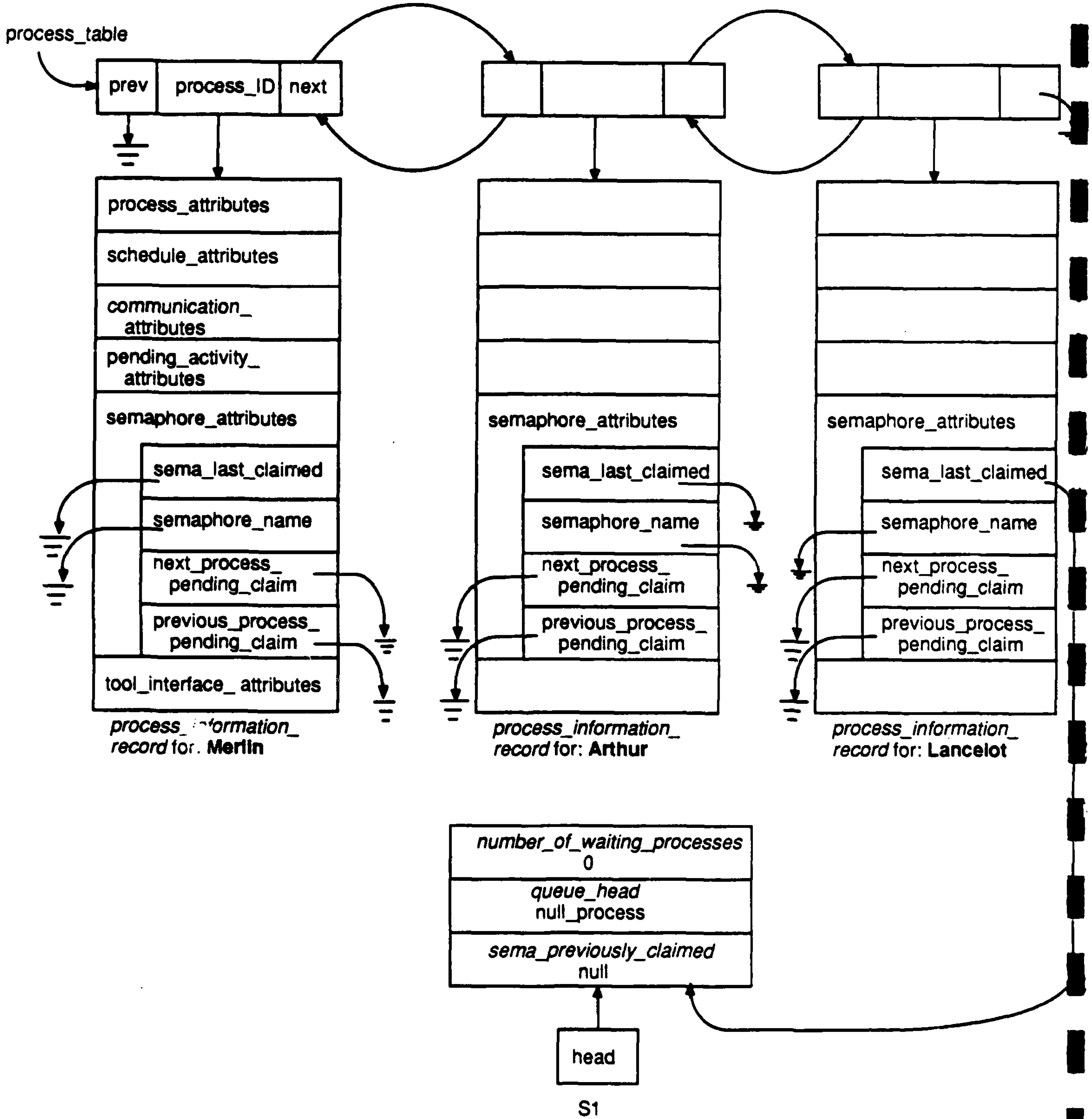

process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
tool_interface_ attributes
process_information_ record for. Merlin
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: Arthur
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: Lancelot
number_of_waiting_processes
0
queue_head
null_process
sema_previously_claimed
null
head
S1

Figure 5-2: Semaphore Structure - Part 6 of 8

Scenario:
Process **Arthur** requests to claim S1 after process **Lancelot** claims S1.

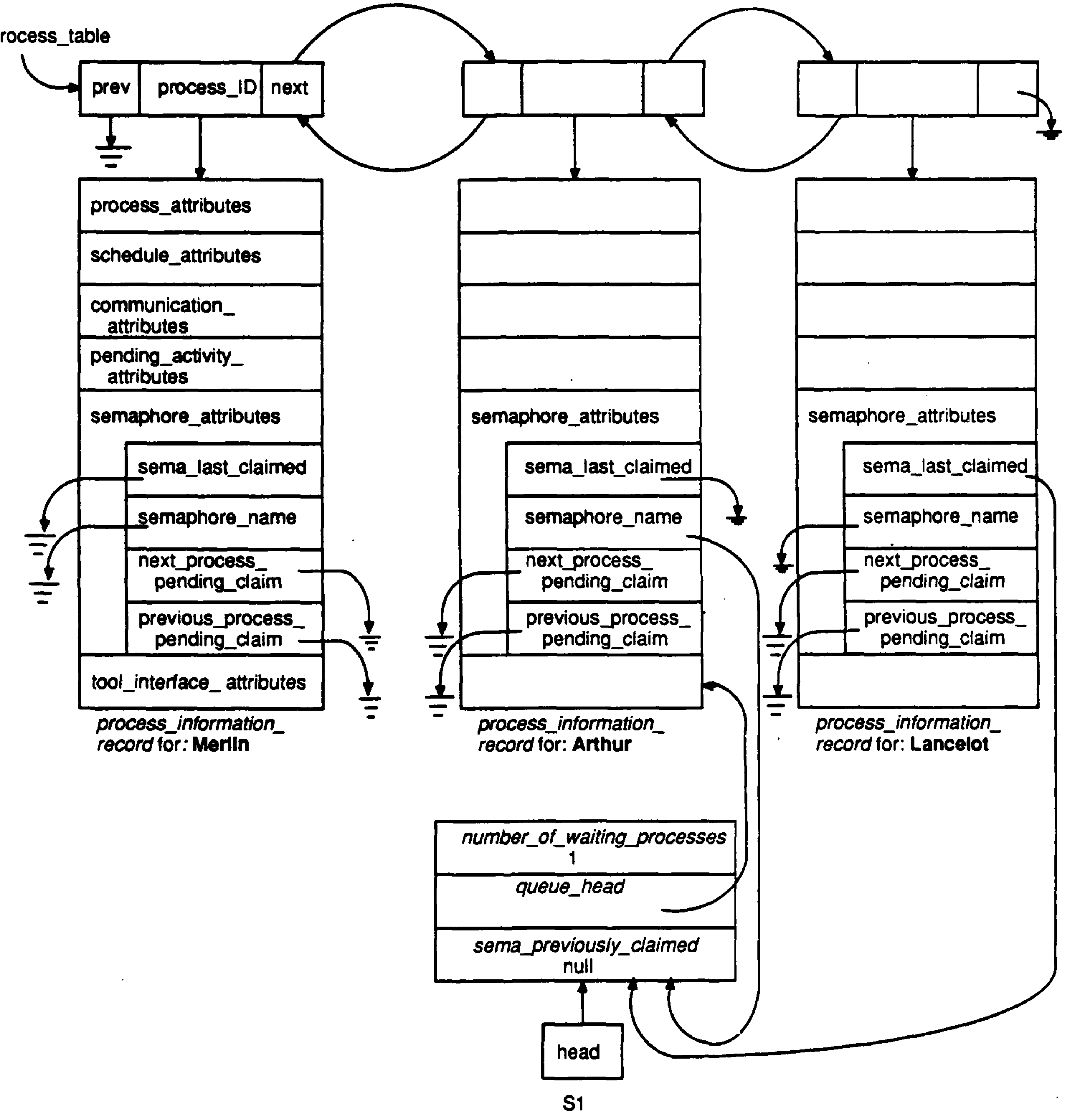

process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
tool_interface_ attributes
process_information_ record for: **Merlin**
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: **Arthur**
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: **Lancelot**
number_of_waiting_processes
1
queue_head
sema_previously_claimed
null
head
S1

Figure 5-2: Semaphore Structure - Part 7 of 8

Scenario:
Process **Merlin** requests to claim S1 after process **Arthur** requests to claim S1 after process **Lancelot** claims S1.

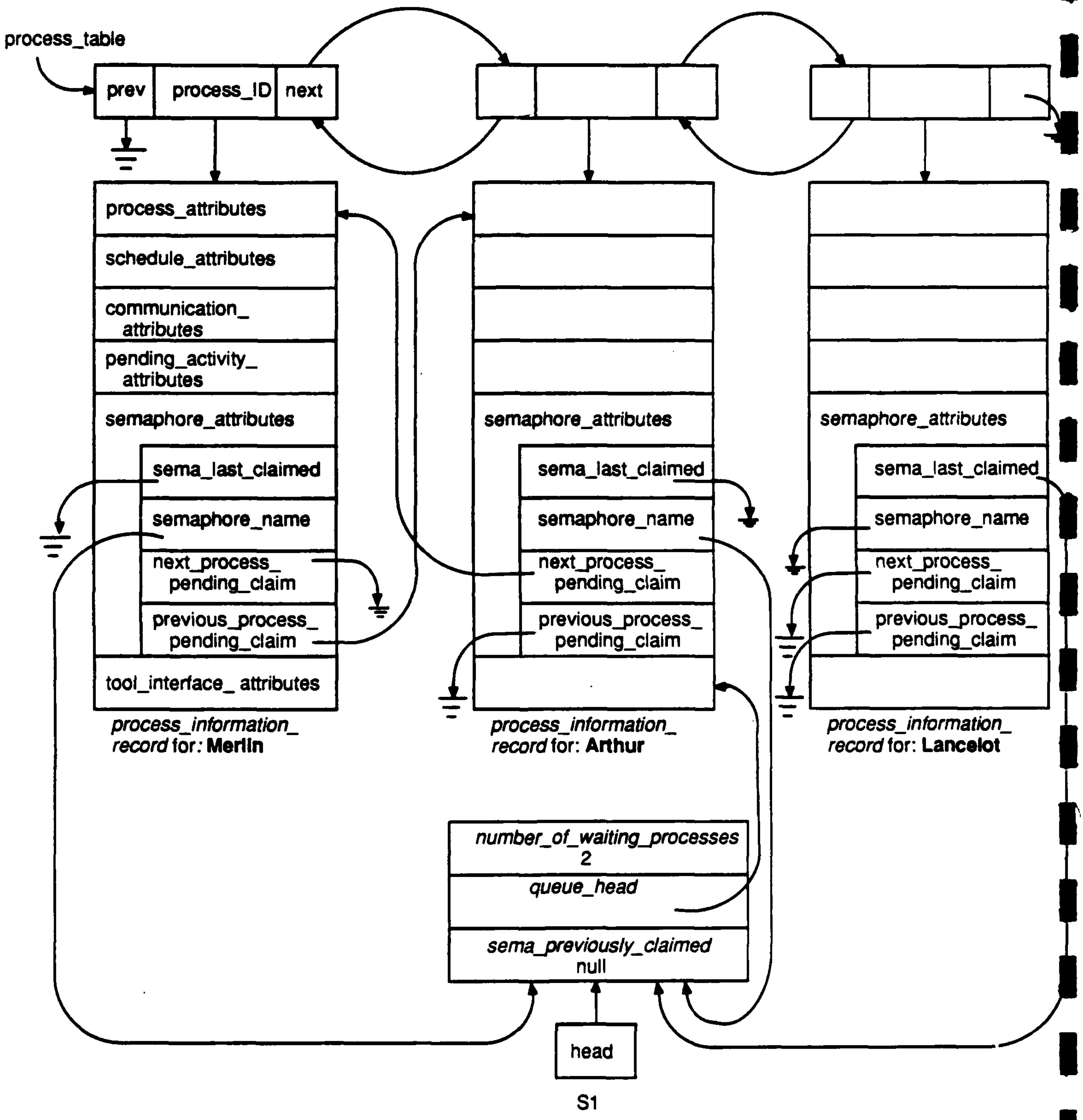

process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
tool_interface_ attributes
process_information_ record for: Merlin
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: Arthur
semaphore_attributes
sema_last_claimed
semaphore_name
next_process_ pending_claim
previous_process_ pending_claim
process_information_ record for: Lancelot
number_of_waiting_processes
2
queue_head
sema_previously_claimed
null
head
S1

Figure 5-2: Semaphore Structure - Part 8 of 8

```
--  queue_head
--    the first process waiting in the queue for this semaphore
--
--    default value:
--      null_process (there are no processes waiting in the queue for
--      this semaphore)
--
--    this value may be set via a call to the Kernel primitive:  claim
--    (if the call is the first process waiting for the semaphore)
--    and reset via a call to the Kernel primitive:  release (if the call
--    is for the last process waiting for the semaphore); otherwise,
--    this value should never change
--
--    sema_previously_claimed
--    the last semaphore claimed by the process that owns this semaphore
--    (i.e., if process P claims semas s1 and s2 in that order, then this
--    component of s2 head will designate s1
--
--    default value : null_semaphore
--
--    The purpose of this component is to chain together in LIFO order
--    all semas currently owned by a process.
--

   type semaphore
   is record
      head : semaphore_head_ptr := new semaphore_head;
   end record;

   type semaphore_head
   is record
      number_of_waiting_processes : hw_integer := -1;
      queue_head : process_identifier := null_process;
      sema_previously_claimed : semaphore_head_ptr := null;
   end record;

   type semaphore_head_ptr is access semaphore_head;
```

There are no representation specifications relevant to a semaphore.

The concrete data structure that implements the semaphore enforces mutually exclusive access to a *semaphore* object. The Kernel primitives that manipulate *semaphores claim* and *release* must therefore be passed their *semaphore* parameter by reference, which in Ada is accomplished by passing them an access value designating the semaphore.

The *semaphore* concrete data structure maintains two logical pieces of information:

1. The processes waiting for this *semaphore*. This information is provided via the *number_of_waiting_processes* and *queue_head* fields. When non-negative, the value of *number_of_waiting_processes* indicates the number of entries in the list of processes pointed to by *queue_head*. The *queue_head* points to the *process_information_record* for the first process waiting for this *semaphore*. The *process_information_record.semaphore_attributes.next_process_pending_claim* field of this first waiting process points to the next process waiting for this

semaphore, and so forth. The *process_information_record.semaphore_attributes. previous_process_pending_claim* fields are linked in the reverse order. Thus, the list of processes waiting for this *semaphore* is threaded through the Process Table data structure (via the *process_information_records*). Figure 5-2 Part 8 illustrates this.

2. Other *semaphores* currently claimed by the process claiming this *semaphore*. The head of this list (i.e., the most recently claimed *semaphore*) is the *process_information_record.semaphore_attributes.sema_last_claimed* field, which points to the most recently claimed *semaphore*, and *sema_previously_claimed* field of that *semaphore* points to the next most recently claimed *semaphore*, which, in turn, points to the next most recently claimed *semaphore*, and so forth. Thus, the list of *semaphores* claimed by this p;rocess is threaded through the *semaphore* data structure. Figure 5-2 Part 4 illustrates this.

In addition, all *semaphores* must be correctly initialized. This can be done automatically in Ada by declaring the *semaphore* object to be a record type with initialized components. These two requirements give rise to the *semaphore* and *semaphore_head* data types.

Immediately after its declaration has been elaborated, a *semaphore* is in a correct initial state and ready for use.

5.1.2.3. Initialization

The initial value of an application-declared semaphore is *free* (i.e., the default value is: -1 indicating that not only are there no processes waiting for the semaphore, there are also no *claims* registered for it; *null_process*, indicating that there are also no *claims* registered for it), and *null*, indicating that this semaphore is not yet claimed). Initialization occurs automatically when the *semaphore* is declared.

5.1.2.4. Additional Allocation Requirements

After declaration of a *semaphore*, no additional allocation is required. To manipulate the list of processes waiting for any semaphore, pointers (i.e., *process_identifiers*) are assigned and unassigned; no dynamic storage allocation is required.

The maximum size of any semaphore is simply the size of its three components.

There is no way to destroy a *semaphore* or to reclaim its storage.

5.1.2.5. Constraints on Usage

Even though the *semaphore* is potentially a visible data structure, it should be treated as an "abstract data type" by the application program. Knowledge of its internal structure should not be exploited in the application program, as this may violate the integrity of the Kernel and the application program.

5.1.3. Process Table

The Process Table is the central repository for information relating to all processes executing on the Kernel. Each node has a Process Table. The structure of the Process Table is identical across the network; much of the information is identical as well. One difference between instances of Process Tables at different nodes is that more information is maintained about local processes than about remote processes. All common information held in Process Tables across nodes is identical.

The application indirectly creates new entries in the Process Table by invoking the Kernel primitive *declare_process*. Additional information is added to the corresponding Process Table entry by invoking the Kernel primitive *create_process.* The Process Table is "pruned" of unnecessary entries during the execution of *initialization_complete*. After this point, the structure of the Process Table is static. Information within the Process Table is read and modified by the Kernel during the execution of the application.

The Process Table is a collection of *process_information_records*, built dynamically; the *process_identifier* points to a *process_information_record*, which was allocated and initialized during process initialization time (e.g., via calls to the Kernel primitives *declare_process* and *create_process*).

Each *process_table_entry* contains a *process_identifier*, and the list of entries is maintained by an instantiation of the *generic_queue_manager*.

Each logical entry in the Process Table comprises two pieces: a *process_table_entry*, which maintains the list of Process Table entries and points to the "real" process information, and *process_information_record* structures, which contain the "real" process information. The details of these structures are presented in the following paragraphs.

The Process Table should *not* be set or read directly by the user, but may be accessed by the application via the *tool_interface* package (see Section 6.1).

5.1.3.1. Exporting Package

Generic_process_table, process_table

5.1.3.2. Structure

Entries in the Process Table have two parts:

1. *Process_table_entry* - which points to a *process_information_record* for a specific process and chains together Process Table entries; and
2. *Process_information_record* - which contains the actual information that comprises the Process Table. This information includes: *process_attributes, schedule_attributes, communication_attributes, pending_activity_attributes, send_w_ACK_attributes, semaphore_attributes*, and *tool_interface_attributes.*

Figure 5-3 illustrates the structure of the Process Table along with the scenario it represents.

```
--
-- entries in the process table comprise:
--   process_ID
--     this is the real reference to the information specific to this
--     process; this is the value, cast as a
--     process_types.process_identifier, that the application uses
--     when referencing a process anywhere in the application program
--
--     default value:
```

Scenario: Contents of the Process Table after execution of *processor_a_Main_Unit* - the Main Unit on processor a.

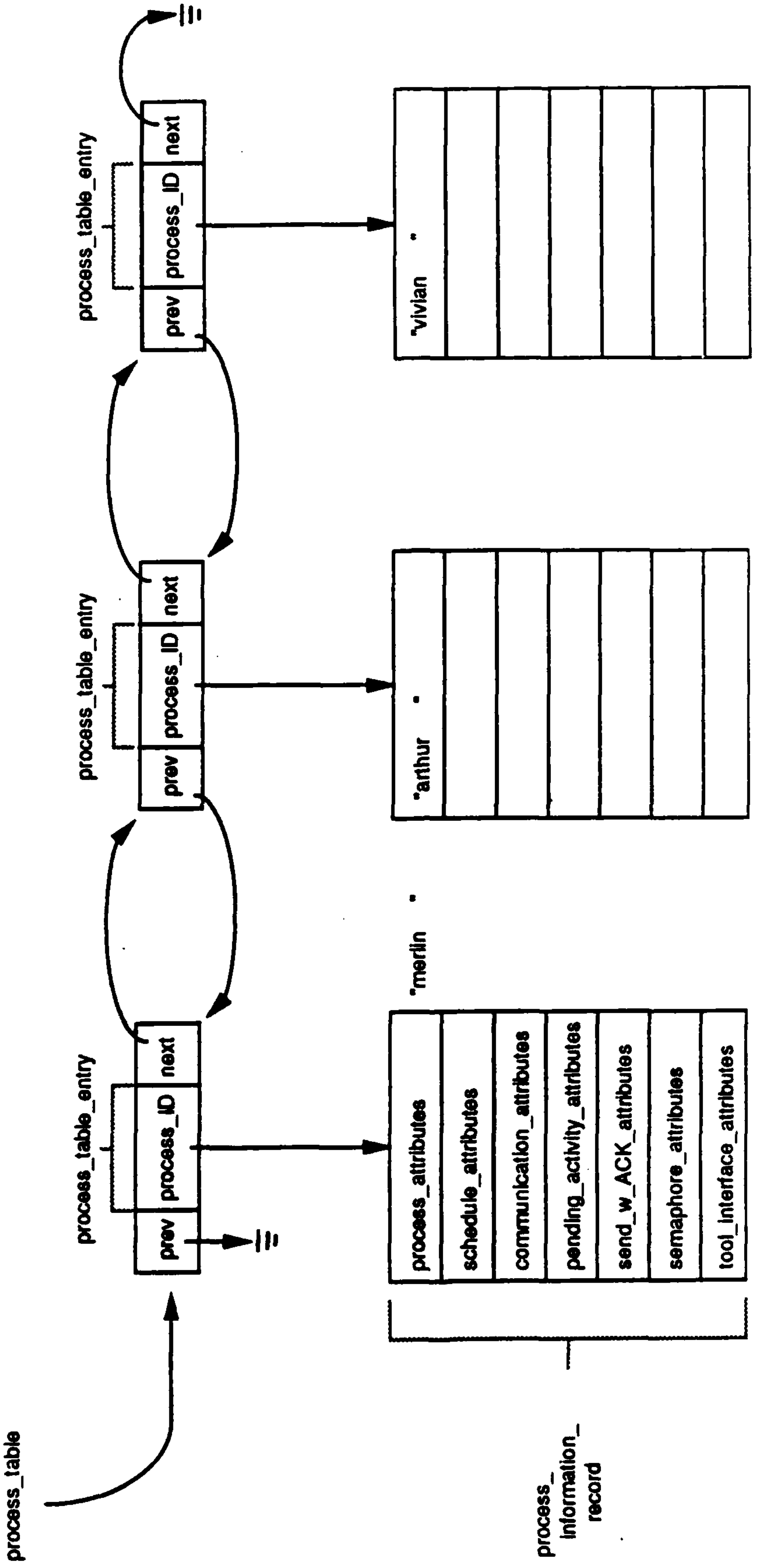


Figure 5-3: Process Table Structure

```
--        none
--
--      this value is set via a call to the Kernel primitive:
--      declare_process; this value should never change after the call
--
--   next and previous pointers, which are maintained by the instantiation
--   of the generic_queue_manager
--

   type process_table_entry
   is record
      process_ID : process_identifier := null_process;
   end record;
```

There are no representation specifications relevant to a *process_table_entry.*

```
--
-- each process information record comprises the following:
--   process_attributes
--      the collection of process attribute information defined below
--
--   schedule_attributes
--      the collection of schedule attribute information defined below
--
--   communication_attributes
--      the collection of communication attribute information defined below
--
--   pending_activity_attributes
--      the collection of pending activity attribute information defined
--      below
--
--   send_w_ACK_attributes
--      the collection of attributes relating to the sending process of
--      a message sent via the Kernel primitive:  send_message_and_wait
--      defined below
--
--   semaphore_attributes
--      the collection of semaphore attribute information defined below
--
--   tool_interface_attributes
--      the collection of tool interface attribute information defined below
--

   type process_information_record
   is record
      process_attributes : process_attributes_information;
      schedule_attributes : schedule_attributes_information;
      communication_attributes : communication_attributes_information;
      pending_activity_attributes : pending_activity_attributes_information;
      send_w_ack_attributes : acknowledged_message_information;
      semaphore_attributes : semaphore_attributes_information;
      tool_interface_attributes : tool_interface_attributes_information;
   end record;
```

There are no representation specifications relevant to a *process_information_record* at this level.

Each component of a *process_information_record* has a detailed structure as well and is presented in turn.

Process Attributes

The *process_attributes_information* structure maintains general information about the process itself and the execution environment built for it by the Kernel. Process information includes: *logical_name*, *kind_of_process*, *process_initialization_status*, and *process_index*. Process Attributes are used by the Kernel for both Kernel processes, local and remote, and non-Kernel devices.

The *process_index* is an internal structure used for communication purposes. It is described in Section 5.2.3.

Execution environment information includes: *code_address*, *stack_low_address*, *stack_high_address*, *context_saved*, and *process_context_save_area*.

The *process_context_save_area* is hardware-specific, and is described in the documentation that will be provided with the code.

Figure 5-4 illustrates the structure of the Process Attributes component of the Process Table along with the scenario it represents.

```
--
-- process attributes include:
--   logical_name
--     the string valued name provided by the application for a process;
--     the length of the string is limited by the user-provided value
--     for the maximum length of a process name
--
--     default value:
--       none
--
--     this value is set via a call to the Kernel primitive:
--     declare_process; it should never change after that call
--
--   kind_of_process
--     indication of whether a process is running on a Kernel processor
--     (and thus follows all Kernel protocols) or a process is really
--     just a non-Kernel device (and thus follows none of the Kernel
--     protocols)
--
--     values include:
--       Kernel_process (follows Kernel protocols)
--       non_Kernel_device (does not follow Kernel protocols)
--
--     default value:
--       Kernel_process
--
--     this value is set via a call to the Kernel primitive:
--     declare_process; it should never change after that call
```

Scenario:Contents of the Process Attributes component of the Process Table entry for the Kernel process **Merlin** at various points of execution of *processor_a_Main_Unit.*

	initial contents	after call to *declare_process* for **Merlin** on **processor a**	after call to *create_process* for **Merlin** on **processor a**	after call to *Initialization_ complete* on **processor a** **
logical_name	all blank	"merlin *	"merlin *	"merlin *
kind_of_process	Kernel_process	Kernel_process	Kernel_process	Kernel_process

process_initialization_status	declared => F
	created => F
	remotely_created => F
		declared => T
		created => F
		remotely_created => F
			declared => T
			created => T
			remotely_created => F
				declared => T
				created => T
				remotely_created => F

process_index	node_number => bus_address'first
	process_number => hw_integer'first
		node_number => bus_address'first
		process_number => hw_integer'first
			node_number => 1
			process_number => -32_766
				node_number => 1
				process_number => -32_766

code_address			evaluation of *merlin_ process_code'address*	evaluation of *merlin_ process_code'address*
stack_low_address			0*	0*
stack_high_address			1_020	1_020
context_saved	via_call	via_call	via_call	via_call
process_context_save_area			dummy initial call frame and context save area created for **Merlin**	dummy initial call frame and context save area created for **Merlin**

*This is an arbitrary value used solely for example purposes.

** *Initialization_complete* makes no changes to any of the Process Attributes.

Figure 5-4: Process Table Process Attributes Component Structure - Part 1 of 2

Scenario: Contents of the Process Attributes component of the Process Table entry for the non-Kernel device **Device** at various points of execution of *processor_a_Main_Unit*, the Main Unit on **processor a**.

	initial contents	after call to *declare_process* for **Device** on **processor a**	there is no call to *create_process* for a non-Kernel device	after call to *initialization_complete* on **processor a** *
logical_name	all blank	"device "		"device "
kind_of_process	Kernel_process	Kernel_device		Kernel_device

process_initialization_status	declared => F
	created => F
	remotely_created => F
		declared => T
		created => F
		remotely_created => F
				declared => T
				created => F
				remotely_created => F

process_index	node_number => bus_address'first
	process_number => hw_integer'first
		node_number => 3 (*physical_address*)
		process_number => *null process number*
				node_number => 3 (*physical_address*)
				process_number => *null process number*

code_address				
stack_low_address				
stack_high_address				
context_saved	via_call	via_call		via_call
process_context_save_area				

* *Initialization_complete* makes no changes to any of the Process Attributes.

Figure 5-4: Process Table Process Attributes Component Structure - Part 2 of 2

```
--
--  process_initialization_status
--    indication of current status of process initialization protocol
--    (i.e., process declaration and creation)
--
--    default value:
--      declared => false (set by declare_process)
--      created => false (set by create_process)
--
--    this value is set via a call to the Kernel primitives:
--    declare_process (the declared component) and create_process (the
--    created component); it should never change after the calls
--
--  process_index
--    another way of referencing a process (in addition to a process
--    identifier) via its owning processor and an identifier unique with
--    respect to that processor
--
--    default value:
--      none
--
--    for a non-Kernel device, this value is set via a call to the Kernel
--    primitive:  declare_process; it should never change after that call;
--    for a Kernel device, this value is set via a call to the Kernel
--    primitive:  create_process; it should never change after that call
--
--  code_address
--    the address of the code that comprises this process
--
--    default value:
--      none
--
--    this value is set via a parameter to the Kernel primitive:
--    create_process; it should never change after that call
--
--  stack_low_address
--    the system low address (e.g., 16#000#) of the Kernel-created
--    process stack; this is the FIRST longword address (i.e., aligned
--    on a 32-bit boundary) at which the Kernel may safely store a
--    longword (i.e., 32 bits) of data in the Kernel-maintained
--    process stack
--
--    *** the Kernel-maintained process stack is always longword-aligned
--
--    default value:
--      none
--
--   this value is set via a call to the Kernel primitive:
--   create_process; it should never change after that call
--
--  stack_high_address
--    the system high address (e.g., 16#FFC#) of the Kernel-created
--    process stack; this is the LAST longword address (i.e., aligned
--    on a 32-bit boundary) at which the Kernel may safely store a
--    longword (i.e., 32 bits) of data in the Kernel-maintained
--     process stack
--
--     *** the Kernel-maintained process stack is always longword-aligned
--
--     default value:
--       none
--
--     this value is set via a call to the Kernel primitive:
--     create_process; it should never change after that call
--
--   context_saved
--     indication of whether or not the current context of this process
--     may be assumed to be saved (i.e., that the context_save_area has
--     contents that are currently valid)
--
--     values include:
--       via_call (context is saved and was saved via the
--         procedure/function calling protocol)
--       via_interrupt (context is saved as was saved via the
--         interrupt handling protocol)
--       not_saved (context must not be assumed to be saved)
--
--     default value:
--       via_call
--
--     this value is set initially via a call to the Kernel primitive:
--     create_process; it is modified by the Kernel as process context is
--     saved and restored (when a process context switch occurs as
--     directed by the Scheduler or when an interrupt occurs)
--
--   process_context_save_area
--     place where the context of a process is saved (e.g., registers,
--     program counter)
--
--     default value:
--       none
--
--     this value is modified by the Kernel as process context is
--     saved and restored (when a process context switch occurs as
--     directed by the Scheduler or when an interrupt occurs)
--

  type process_attributes_information
  is record
     logical_name : hw_string (1 ..
        positive
           (process_managers_globals.maximum_length_of_process_name) ) :=
              (others => ' ');
     kind_of_process : process_type := Kernel_process;
     process_initialization_status : process_initialization_status_type;
     process_index : network_globals.process_index_type;
     code_address : hw_address;
     stack_low_address : hw_address;
     stack_high_address : hw_address;
     context_saved : context_switcher_globals.context_saved_type :=
        context_switcher_globals.via_call;
```

```
      process_context_save_area :
         context_save_area.context_save_area_contents;
   end record;
```

There are no representation specifications relevant to a *process_attributes_information* record at this level. There are representation specifications that define the layout of the *process_context_save_area*; these are defined in the documentation that will be provided with the code.

The following information is used in the definition of the process attributes.

```
--
-- indication of current status of process initialization protocol
-- (i.e., process declaration and creation)
-- components include:
--   declared
--     indication of whether or not declare_process successfully completed
--
--     values include:
--       true (successful completion of declare_process)
--       false (unsuccessful completion of declare_process)
--
--     default value:
--       false (unsuccessful completion of declare_process)
--
--     this value is set via a call to the Kernel primitive:
--     declare_process; it should never change after this call
--
--   created
--     indication of whether or not create_process successfully completed
--
--     values include:
--       true (successful completion of create_process)
--       false (unsuccessful completion of create_process)
--
--     default value:
--       false (unsuccessful completion of create_process)
--
--     this value is set via a call to the Kernel primitive:
--     create_process; it should never change after this call
--
--   remotely_created
--     indication of whether or not the process was created on another
--     node
--
--     values included:
--       true (process was created on a remote node)
--       false (process was not created remotely)
--
--     default value:
--       false (no knowledge yet about where the process was created)
--
--   This value is set by the receive_datagram_interrupt_handler whenever
--   a process_created message arrives.

  type process_initialization_status_type
  is record
     declared : boolean := false;
     created : boolean := false;
     remotely_created : boolean := false;
  end record;


--
-- indication of whether or not a process is running on a Kernel processor
-- (and thus follows all Kernel protocols) or a process is really just a
-- non-Kernel device (and thus follows none of the Kernel protocols)
--
-- values include:
--   Kernel_process (follows Kernel protocols)
--   non_Kernel_device (does not follow Kernel protocols)
--

  type process_type is (
     Kernel_process, non_Kernel_device);
```

There are no representation specifications relevant to any of these types.

Schedule Attributes

The *schedule_attributes_information* structure maintains general information used by the Kernel Scheduler. This information provides a snapshot of *the status* in which the Scheduler considers this process. This information includes: process *state*, *priority*, *preemption*, *block_time*, and *unblock_time*. Schedule Attributes are used by the Kernel only for local Kernel processes.

Figure 5-5 illustrates the structure of the Schedule Attributes component of the Process Table along with the scenario it represents.

```
--
-- schedule attributes include:
--   state
--     the current state of this process; this is used by the Kernel's
--     Scheduler
--
--     values include:
--       running (this process controls the processor and is the
--        currently running process)
--       suspended (this process is able to run but another process is
--        currently running)
--       blocked (this process is unable to run)
--       dead (this process is no longer able to run)
--
--     default value:
--       suspended (this process is able to run but another process is
--        currently running)
--
--     this value is set by the Scheduler as the process state changes
--     (due to:  a call to a Kernel primitive, the passage of time, the
```

Scenario:Contents of the Schedule Attributes component of the Process Table entry for the Kernel process **Merlin** at various points of execution of *processor_a_Main_Unit.*

	initial contents	after call to *declare_process* for **Merlin** on **processor a** *	after call to *create_process* for **Merlin** on **processor a**	after call to *initialization_ complete* on **processor a** **
process_attributes.logical_name	"merlin "	"merlin "	"merlin "	"merlin "
state	suspended	suspended	suspended	suspended
priority			highest_priority (1)	highest_priority (1)
preemption			default_preemption (disabled)	default_preemption (disabled)
block_time	0, 0	0, 0	0, 0	0, 0
unblock_time			*get current time*	*get current time*

**Declare_process* makes no changes to any of the Schedule Attributes.

***Initialization_complete* makes no changes to any of the Schedule Attributes.

Figure 5-5: Process Table Schedule Attributes Component Structure

```
--     occurrence of an event)
--
--   priority
--     the current priority of this process; this is used by the Kernel's
--     Scheduler, and the primitive:
--     process_attribute_readers.get_process_priority, and all primitives
--     take a (resumption) priority as a parameter
--
--     default value:
--       none
--
--     this value is set initially via a parameter to the Kernel primitive:
--     create_process; it may be modified by the Scheduler as the process
--     priority changes
--     (due to:  a call to a Kernel primitive, the passage of time, the
--     occurrence of an event)
--
--   preemption
--     an indication of whether or not this process may be preempted; this
--     is used by the Kernel's Scheduler (for time slicing), and the
--     primitives:
--     process_attribute_readers.get_process_preemption
--     process_attribute_modifiers.set_process_preemption
--
--     values include:
--       true (this process may be preempted)
--       false (this process may not be preempted)
--
--     default value:
--       none; provided by initial call to create_process
--
--     this value is set initially via a parameter to the Kernel primitive:
--     create_process; it may be modified via a call to the Kernel
--     primitive:  set_process_preemption
--
--   block_time
--     the Kernel_time at which the state of this process became blocked;
--     this is used by the Kernel's Scheduler
--
--     default value:
--       zero_Kernel_time
--
--     this value is set by the Scheduler when a process calls a blocking
--     Kernel primitive; it may be modified by the Scheduler when the
--     process becomes blocked again (note that
--     this value is not strictly needed by the Kernel, as the Scheduler
--     maintains a time-ordered queue for its processing; it is included
--     for debugging purposes)
--
--   unblock_time
--     the Kernel_time at which the state of this process became
--     unblocked; this is used by the Kernel's Scheduler
--
--     default value:
--       none
```

```
 --
 --    this value is initially set via a call to the Kernel primitive:
 --    create_process; it may be modified by the Scheduler once the
 --    process becomes unblocked (note that this value is not strictly
 --    needed by the Kernel, as the Scheduler doesn't require knowledge of
 --    a process's time once it is unblocked; it is included for
 --    debugging purposes)
 --

   type schedule_attributes_information
   is record
      state : schedule_types.process_state := schedule_types.suspended;
      priority : schedule_types.priority;
      preemption : schedule_types.preemption;
      block_time : Kernel_time.Kernel_time := Kernel_time.zero_Kernel_time;
      unblock_time : Kernel_time.Kernel_time;
   end record;
```

There are no representation specifications relevant to a *schedule_attributes_information* record.

Communication Attributes

The *communication_attributes_information* structure maintains information about this process's communication requirements. This information includes: *next_available_message_ID*, *maximum_message_queue_size*, *message_queue*, *current_send_buffer*, *queue_overwrite_rule*, and *message_queue_overflow*. Communication Attributes are used by the Kernel only for local Kernel processes.

Figure 5-6 illustrates the structure of the Communication Attributes component of the Process Table along with the scenario it represents.

```
 --
 -- communication attributes include:
 --   next_available_message_ID
 --     the message ID that may be used for the next message sent by this
 --     process via send_message_and_wait; this value is constantly
 --     increasing
 --
 --     default value:
 --       first (lowest) message identifier available
 --
 --     this value is modified only via a call to the Kernel primitive:
 --     send_message_and_wait
 --
 --   maximum_message_queue_size
 --     the maximum number of messges that may be queued awaiting receipt
 --     for this process
 --
 --     default value:
 --       none
 --
 --     this value is set by a parameter to the Kernel primitive:
 --     create_process; it should never be modified after that call
 --
```

Scenario:Contents of the Communication Attributes component of the Process Table entry for the Kernel process **Merlin** at various points of execution of *processor_a_Main_Unit*.

	initial contents	after call to *declare_process* for **Merlin** on **processor a** *	after call to *create_process* for **Merlin** on **processor a**	after call to *initialization_ complete* on **processor a** **
process_attributes.logical_name	"merlin "	"merlin "	"merlin "	"merlin "
next_available_message_ID	0	0	0	0
maximum_message_queue_size		0	100	100
message_queue	null	null	pointer to incoming message queue head	pointer to incoming message queue head
current_send_buffer	null	null	null	null
queue_overwrite_rule			drop_newest_message	drop_newest_message
message_queue_overflow	F	F	F	F

Figure 5-6: Process Table Communication Attributes Component Structure

**Declare_process* makes no changes to any of the Communication Attributes.

***Initialization_complete* makes no changes to any of the Communication Attributes.

```
--  message_queue
--    pointer to the first message in the message queue for this process;
--    this is used by the Kernel primitives to send and receive messages
--
--    default value:
--      null
--
--   this value is set via a call to the Kernel primitive:
--   create_process; it should never be modified after that call
--
--  current_send_buffer
--    pointer to the current buffer being used to send a message, via
--    an application call to send_message*
--
--    default value:
--      null
--
--    this value is set via a call to the Kernel primitives:
--    send_message and send_message_and_wait; it is used by the Kernel
--    primitives:  die and kill; it should never be modified outside
---   these calls
--
--  queue_overwrite_rule
--    indication of how this process is to handle incoming message queue
--    overflow
--
--    values include:
--      drop_newest_message (the most recently received message is lost)
--
--    default value:
--      none
--
--    this value is set by a parameter to the Kernel primitive:
--    create_process; it should never change after that call
--
--  message_queue_overflow
--    indication of whether or not the incoming message queue for this
--    process is currently full and messages are being lost or in danger
--    of being lost; this is used by the Kernel primitive:
--    receive_message
--
--    values include:
--      true (at least one message has been lost already)
--      false (no messages have been lost since last call to
--       receive_message)
--
--    default value:
--      false (no messages have been lost since last call to
--      receive_message)
--
--    this value may be set by the Kernel as messages are received; its
--    value may be reset via a call to the Kernel primitive:
--    receive_message
--
```

```
type communication_attributes_information
is record
   next_available_message_ID : datagram_globals.message_identifier := 0;
   maximum_message_queue_size : hw_long_natural;
   message_queue : datagram_globals.datagram_pointer := null;
   current_send_buffer : datagram_globals.datagram_pointer := null;
   queue_overwrite_rule :
      process_managers_globals.how_to_handle_message_queue_overflow;
   message_queue_overflow : boolean := false;
end record;
```

There are no representation specifications relevant to a *communication_attributes_information* record.

Pending Activity Attributes

The *pending_activity_attributes_information* structure maintains information about activities that are currently pending for this process. The value of the *pending_activity* component determines which of the remaining fields contains relevant information about those activities that are mutually exclusive (i.e., via a call to one of the Kernel primitives: *receive_message*, *claim*, *send_message_and_wait*, *wait*). In addition to one of those events, a process may also have an alarm enabled and/or may be ready to raise a Kernel exception. *Pending_activity_attributes_information* includes: the *pending_activity* itself, *pending_event_ID*, *current_pending_message*, *alarm_event_ID*, *alarm_resumption_priority*, *exception_name*. Pending Activity Attributes are used by the Kernel only for local Kernel processes.

Figure 5-7 illustrates the structure of the Pending Activity Attributes component of the Process Table along with the scenario it represents.

```
--
-- pending activity attributes includes:
--   pending_activity
--     indication of what kind of event has caused the process to block
--
--     values include:
--       see pending_activity_type just above
--
--     default value:
--       nothing_pending
--
--     this value is set via a call to any blocking Kernel primitive:
--     receive_message, claim, send_message_and_wait, wait; it is
--     reset by the Kernel upon expiry of of the timeout, occurrence of
--     the event awaited (e.g., receipt of message or ACK/NAK, availability
--     of the semaphore)
--
--   pending_event_ID
--     an index into the time keeper's time event queue indicating the
--     event entry corresponding to the value of pending_activity for this
--     process;
--     used by Kernel internals as a link into the Kernel's time_keeper
--
```

Scenario:Contents of the Pending Activity Attributes component of the Process Table entry for the Kernel process **Merlin** at various points of execution of *processor_a_Main_Unit.*

	initial contents	after call to *declare_process* for **Merlin** on **processor a** *	after call to *create_process* for **Merlin** on **processor a** **	after call to *initialization_complete* on **processor a** ***
process_attributes.logical_name	"merlin "	"merlin "	"merlin "	"merlin "
pending_activity	nothing_pending	nothing_pending	nothing_pending	nothing_pending
pending_event_ID	null_event	null_event	null_event	null_event
current_pending_message				
alarm_event_ID	null_event	null_event	null_event	null_event
alarm_resumption_priority				
exception_name	no_exception	no_exception	no_exception	no_exception

Figure 5-7: Process Table Pending Activity Attributes Component Structure

**Declare_process* makes no changes to any of the Pending Activity Attributes.

***Create_process* makes no changes to any of the Pending Activity Attributes.

****Initialization_complete* makes no changes to any of the Pending Activity Attributes.

```
--    default value:
--      null event
--
--    this value is set via a call to any blocking Kernel primitive (as
--    enumerated above); it is reset by the Kernel as described above
--
--  current_pending_message
--    if pending_activity indicates send_with_ACK_pending, this is the
--    message identifier for which an ACK or a NAK is expected
--
--    default value:
--      none
--
--    this value is set via a call to the Kernel primitive:
--    send_message_and_wait; it is reset by the Kernel upon receipt of
--    the ACK/NAK for the identified message; it is valid if and only
--    if pending_activity indicates send_with_ACK_pending
--
--  alarm_event_ID
--    an index into the time keeper's time event queue indicating the
--    alarm expiration event for this process
--
--    default value:
--      null event
--
--    this value is set via a call to the Kernel primitive:  set_alarm;
--    it may be reset either via a call to the Kernel primitive:
--    cancel_alarm or by the Kernel upon the expiry of the alarm
--
--  alarm_resumption_priority
--    if alarm_event_ID is not the null event, the priority at which
--    this process is to be resumed upon the expiration of the alarm
--
--    default value:
--      none
--
--    this value is set via a call to the Kernel primitive:  set_alarm;
--    it should never change otherwise; it is valid if and only if
--    alarm_event_ID is not the null_event
--
--  exception_name
--    indication of whether or not the Kernel is raising an exception
--    for this process; if not no_exception, then also an indication of
--    which exception is to be raised
--
--    values include:
--      there is an enumeration literal corresponding to each exception
--      the Kernel may raise; see package Kernel_exceptions
--
--    default value:
--      no_exception
--
--    this value is set whenever the Kernel internals detect a Kernel
--    exception that is to be raised and reset to no_exception upon
--    completion of internal exception processing
```

```
--

   type pending_activity_attributes_information
   is record
      pending_activity : pending_activity_type := nothing_pending;
      pending_event_ID : event_identifier := null_event;
      current_pending_message : datagram_globals.message_identifier;
      alarm_event_ID : event_identifier := null_event;
      alarm_resumption_priority : schedule_types.priority;
      exception_name : Kernel_exceptions.Kernel_exceptions :=
         Kernel_exceptions.no_exception;
   end record;
```

There are no representation specifications relevant to a *pending_activity_attributes_information* record.

```
--
-- the kinds of mutually exclusive activities that can be pending for a
-- single process are:
--   receive_pending (the application called the Kernel primitive:
--      receive_message and is blocked until a message is received or until
--      the timeout expires)
--   semaphore_pending (the application called the Kernel primitive:
--      claim and is blocked until the requested semaphore is free or until
--      the timeout expires)
--   send_with_ACK_pending (the application called the Kernel primitive:
--      send_message_and_wait and is blocked until an ACK or a NAK is
--      returned or until the timeout expires)
--   wait_pending (the application called the Kernel primitive:  wait and is
--      blocked until the timeout expires)
--   nothing_pending (there is no activity on which this process is
--      currently pending)
--

   type pending_activity_type is (
      receive_pending,
      semaphore_pending,
      send_with_ACK_pending,
      wait_pending,
      nothing_pending);
```

There are no representation specifications relevant to any of these types.

Acknowledged Message Information

The *acknowledged_message_information* structure maintains information about the outstanding acknowledged send request (i.e., call to the Kernel primitive *send_message_and_wait*) that may have been issued by this process. This information includes: *event_ID*, the *message* sent by this process, and the receiver's incoming message *queue*. Acknowledged Message Information is used by the Kernel only for local Kernel processes.

Figure 5-8 illustrates the structure of the Acknowledged Message Information component of the Process Table along with the scenario it represents.

Scenario:Contents of the Acknowledged Message Information Attributes component of the Process Table entry for the Kernel process **Merlin** at various points of execution of *processor_a_Main_Unit*.

	initial contents	after call to *declare_process* for **Merlin** on **processor a** *	after call to *create_process* for **Merlin** on **processor a** **	after call to *initialization_ complete* on **processor a** ***
process_attributes.logical_name	"merlin "	"merlin "	"merlin "	"merlin "
event_ID	null_event	null_event	null_event	null_event
message	null	null	null	null
queue	null	null	null	null

**Declare_process* makes no changes to any of the Acknowledged Message Information Attributes.

***Create_process* makes no changes to any of the Acknowledged Message Information Attributes.

****Initialization_complete* makes no changes to any of the Acknowledged Message Information Attributes.

Figure 5-8: Process Table Acknowledged Message Information Component Structure

```
--
-- acknowledged message information is maintained in the process table
-- entry corresponding to the SENDING process; this information refers to
-- data about the RECEIVING process's incoming message queue; this is done
-- to facilitate ready access to message queue information to process
-- timeout expiration efficiently
--
-- acknowledged message information includes:
--   event_ID
--     indication that this process is the SENDING process and
--     sent a message via the Kernel
--     primitive:  send_message_and_wait; thus an ACK is required to be
--     returned to THIS process upon receipt of the corresponding message
--
--     default value:
--       null_event
--
--     this value is set by the Kernel when it receives a message that
--     was sent via a call to the Kernel primitive:  send_message_and_wait;
--     it is reset via a call to the Kernel primitive:  receive_message
--     or by the Kernel when the corresponding timeout expires
--
--   message
--     an index into the RECEIVING process's incoming message queue
--     indicating the message that this process sent via the Kernel
--     primitive:  send_message_and_wait
--
--     default value:
--       null
--
--     this value is set by the Kernel when it receives a message that
--     was sent via a call to the Kernel primitive:  send_message_and_wait;
--     it is reset via a call to the Kernel primitive:  receive_message
--     or by the Kernel when the corresponding timeout expires
--
--   queue
--     a pointer to the head of the RECEIVING process's message queue -
--     i.e., the message queue that contains the message field just above
--
--     default value:
--       null
--
--     this value is set by the Kernel when it receives a message that
--     was sent via a call to the Kernel primitive:  send_message_and_wait;
--     it is reset via a call to the Kernel primitive:  receive_message
--     or by the Kernel when the corresponding timeout expires
--

  type acknowledged_message_information
  is record
     event_ID : event_identifier := null_event;
     message : datagram_globals.datagram_pointer := null;
     queue : datagram_globals.datagram_pointer := null;
  end record;
```

There are no representation specifications relevant to an *acknowledged_message_information* record.

Semaphore Attributes

The *semaphore_attributes_information* structure maintains information about any semaphore that is requested or actually claimed by this process. This information includes: *semaphore_name*, *next_process_pending_claim*, and *previous_process_pending_claim*. Semaphore Attributes are used by the Kernel only for local Kernel processes.

Figure 5-9 illustrates the structure of the Semaphore Attributes component of the Process Table along with the scenario it represents.

```
--
-- semaphore attributes include:
--   sema_last_claimed
--     the identity of the semaphore most recently claimed by
--     the process, and still owned by it.
--
--     default value:
--       null_semaphore
--
--   semaphore_name
--     the identity of the semaphore on which this process is currently
--     waiting
--
--     default value:
--       none
--
--     this value is set via a call to the Kernel primitive:  claim;
--     reset by release
--
--   next_process_pending_claim
--     the process identifier for the process that called the Kernel
--     primitive claim after this process did
--
--     default value:
--       null_process_ID
--
--     this value is set by the next call of the Kernel primitive:  claim;
--     it may be reset by the Kernel if the timeout of that claim expires
--
--   previous_process_pending_claim
--     the process identifier for the process that called the Kernel
--     primitive claim before this process did
--
--     default value:
--       null_process_ID
--
--     this value is set via the current call to the Kernel primitive:
--     claim; it is reset by the Kernel if the timeout of the claim
--     expires or once the previous process releases the semaphore
--

  type semaphore_attributes_information
```

Scenario:Contents of the Semaphore Attributes component of the Process Table entry for the Kernel process **Merlin** at various points of execution of *processor_a_Main_Unit*.

	initial contents	after call to *declare_process* for **Merlin** on **processor a** *	after call to *create_process* for **Merlin** on **processor a** **	after call to *initialization_ complete* on **processor a** ***
process_attributes.logical_name	"merlin "	"merlin "	"merlin "	"merlin "
sema_last_claimed	null_semaphore	null_semaphore	null_semaphore	null_semaphore
semaphore_name	null_semaphore	null_semaphore	null_semaphore	null_semaphore
next_process_pending_claim	null_process	null_process	null_process	null_process
previous_process_pending claim	null_process	null_process	null_process	null_process

**Declare_process* makes no changes to any of the Semaphore Attributes.

***Create_process* makes no changes to any of the Semaphore Attributes.

****Initialization_complete* makes no changes to any of the Semaphore Attributes.

Figure 5-9: Process Table Semaphore Attributes Component Structure

```
is record
   sema_last_claimed : semaphore := null_semaphore;
   semaphore_name : semaphore := null_semaphore;
   next_process_pending_claim : process_identifier := null_process;
   previous_process_pending_claim : process_identifier := null_process;
end record;
```

There are no representation specifications relevant to an *semaphore_attributes_information* record.

Additional information about the function of these fields is provided in Section 5.1.2.

Tool Interface Attributes
This information will be provided in the next version of this document.

5.1.3.3. Initialization

The Kernel initializes the Process Table with a single *process_table_entry* and *process_information_record* structure. Table 5-1 shows the fields of the Process Table that have defined default va!ues.

The number of entries in the Process Table may vary from node to node. The maximum size of the Process Table after initialization is complete may be limited by the tailoring parameter *maximum_number_of_processes_value*. See Section C.2.5 for more details.

5.1.3.4. Additional Allocation Requirements

The Process Table is a dynamic data structure during the initialization process (i.e., until the Kernel primitive *initialization_complete* finishes executing).

When the Kernel primitive *declare_process* is invoked the first time, it uses the initial Process Table entry. Each invocation of *declare_process* causes a new entry to be made into the Process Table consisting of a *process_table_entry* and *process_information_record* pair. This storage is allocated dynamically.

The call to the Kernel primitive *declare_process* for Kernel processes initializes the following fields:

Initialization Via Call to *Declare_process* for Kernel Process	
Field Name	Value
process_attributes.logical_process	< input parameter >
process_attributes.kind_of_process	Kernel_process
process_attributes.process_initialization_status.declared	true

Table 5-1: Process Table Defined Default Values

Process Table Defined Default Values	
Field Name	Value
communication_attributes.current_send_buffer	null
communication_attributes.message_queue	null
communication_attributes.message_queue_overflow	false
communication_attributes.next_available_message_ID	0
pending_activity_attributes.alarm_event_ID	null_event
pending_activity_attributes.exception_name	no_exception
pending_activity_attributes.pending_activity	nothing_pending
pending_activity_attributes.pending_event_ID	null_event
process_attributes.context_saved	via_call
process_attributes.kind_of_process	Kernel_process
process_attributes.process_index.node_number	bus_address'first
process_attributes.process_index.process_number	hw_integer'first
process_attributes.process_initialization_status.created	false
process_attributes.process_initialization_status.declared	false
process_attributes.process_initialization_status.remotely_created	false
schedule_attributes.block_time	zero_Kernel_time
schedule_attributes.state	suspended
semaphore_attributes.next_process_pending_claim	null_process
semaphore_attributes.previous_process_pending_claim	null_process
semaphore_attributes.sema_last_claimed	null_semaphore
semaphore_attributes.semaphore_name	null_semaphore
send_w_ACK_attributes.event_ID	null_event
send_w_ACK_attributes.message	null
send_w_ACK_attributes.queue	null
tool_attributes*	provided in the next version of this document

The call to the Kernel primitive *declare_process* for non-Kernel devices initializes the following fields:

Initialization Via Call to *Declare_process* for Non-Kernel Device	
Field Name	Value
process_attributes.logical_process	< input parameter >
process_attributes.kind_of_process	non_Kernel_device
process_attributes.process_initialization_status.declared	true
process_attributes.process_index.node_number	< from NCT >
process_attributes.process_index.process_number	0

The call to the Kernel primitive *create_process* initializes the fields in Table 5-2.

Table 5-2: Initialization Via Call to *Create_process*

1. Additional Allocation Requirements

Initialization via Call to *Create_process*	
Field Name	Value
process_attributes.process_index.node_number	< hardware-specific value >
process_attributes.process_index.process_number	< next available number >
process_attributes.code_address	< input parameter >
process_attributes.stack_low_address	< next available longword >
process_attributes.stack_high_address	< computed using input parameter >
process_attributes.context_saved	via_call
process_attributes.process_context_save_area	< appropriate setup >
process_attributes.process_initialization_status.created	true
schedule_attributes.priority	< input parameter >
schedule_attributes.preemption	< input parameter >
schedule_attributes.unblock_time	< current clock reading >
communication_attributes.maximum_message_queue_size	< input parameter >
communication_attributes.queue_overwrite_rule	< input parameter >
communication_attributes.message_queue	< next available space >

The Process Table is pruned during the execution of the Kernel primitive *initialization_complete*. During its execution, the following Process Table entries are eliminated from the Process Table (and the storage is optionally reclaimed):

1. Kernel processes that are created on remote processors but are not declared locally.

The exception *network_failure* is raised and the node is killed (i.e., it does not initialize) if any of the following inconsistencies are detected in the Process Table during pruning and consistency checking:

1. Kernel processes that are declared locally but are not created anywere in the network, or
2. Kernel processes that are created both locally and remotely.

When the Kernel primitive *initialization_complete* terminates, there may be at most *maximum_number_of_processes_value* entries in the Process Table. The size of each entry is the size of the *process_table_entry* record and the *process_information_record*. The size of the *process_information_record* is constrained by the tailoring parameter: *maximum_length_of_process_name_value.*

5.1.3.5. Constraints on Usage

The Process Table should never be accessed directly by the application. It should only be read via the *tool_interface* subprograms. See Section 6.1 for more information.

Even though the Process Table is potentially a visible data structure, it should be treated as an "abstract data type" by the application program. Knowledge of its internal structure should not be exploited in the application program, as this may violate the integrity of the Kernel and the application program.

5.2. Internal Data Structures

5.2.1. Datagram Queues

Datagrams are used by the Kernel to manage messages. A *datagram* is never referenced directly, but is instead always accessed through a *datagram pointer.* For consistancy, the *datagram* that is sent from one process is identical to the *datagram* that is received by another. This section describes the *datagram* contents; more details about *datagrams* and the lower layers of the Kernel communication protocol are described in the documentation that will be provided with the code.

5.2.1.1. Exporting Package

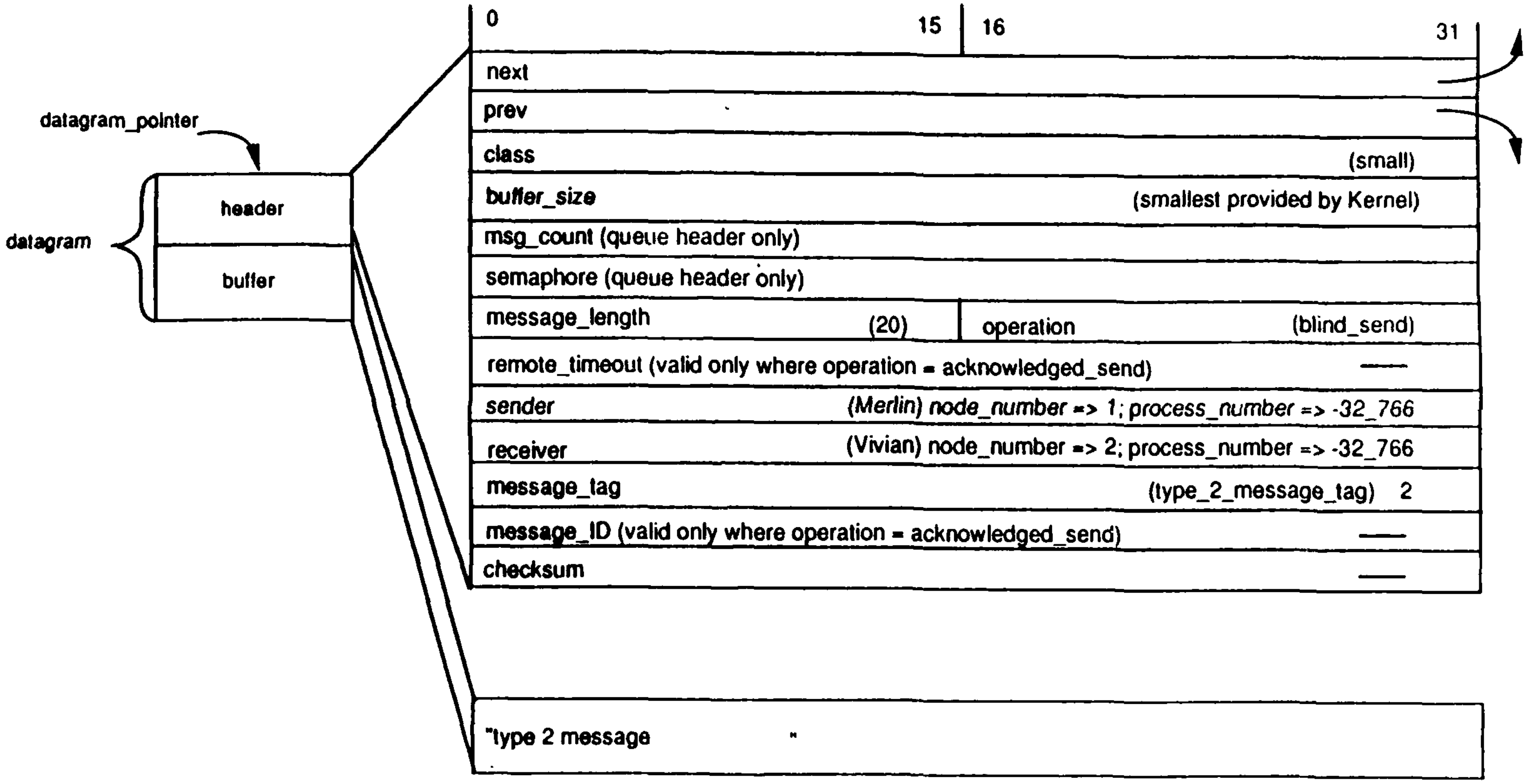
Datagram_globals, datagram_management

5.2.1.2. Structure

A *datagram* is a static data structure; it is fully defined at compile time. A *datagram* comprises two pieces: a *datagram_header*, which contains protocol information, and a *buffer*, which contains the actual text of the message being sent.

Figure 5-10 illustrates the structure of the *datagram* along with the scenario it represents.

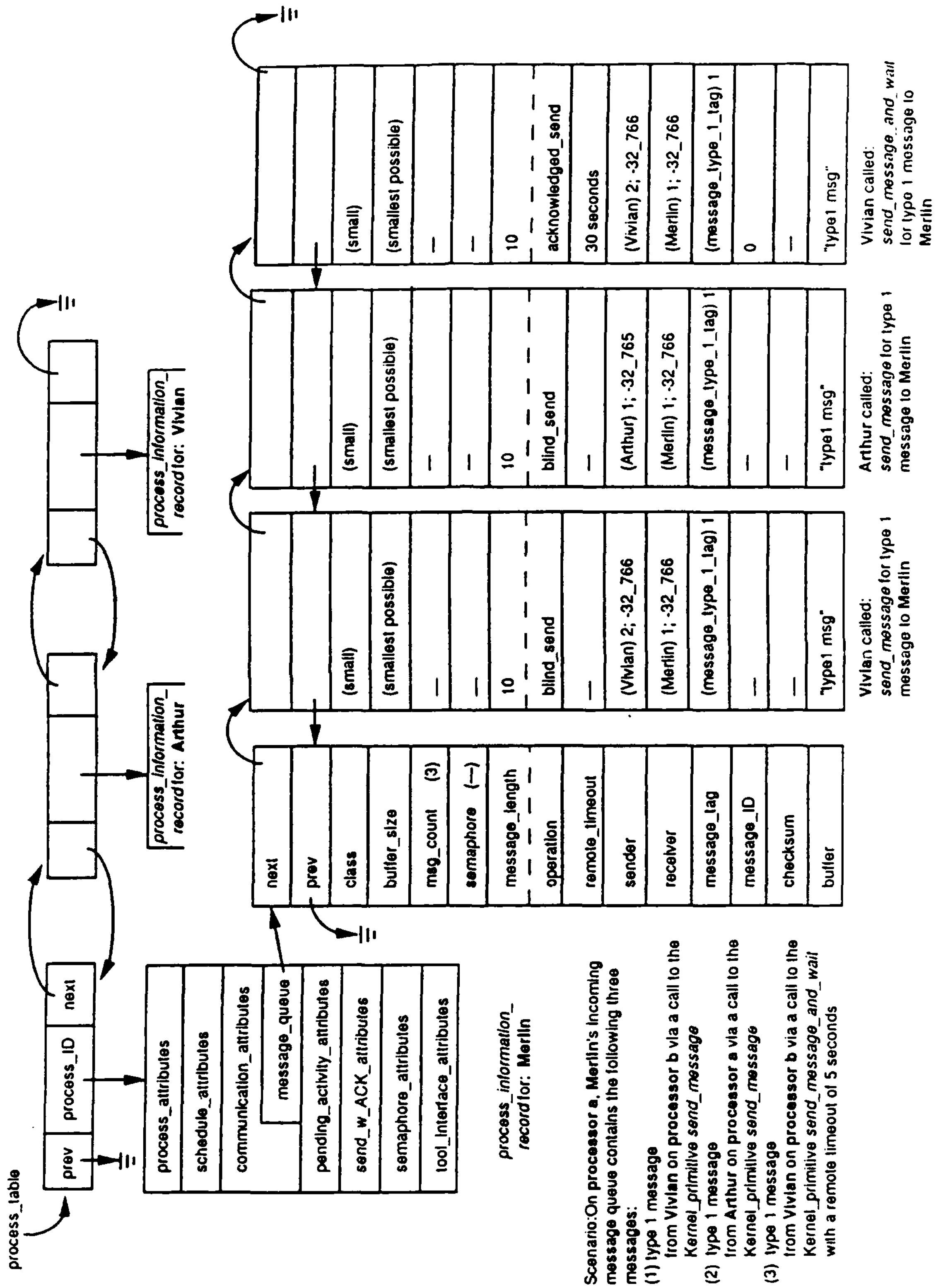
Scenario:On **processor a**, **Merlin** calls the Kernel primitive *send_message* to send a type 2 message to **Vivian** on **processor b**.



next			
prev			
class			(small)
buffer_size			(smallest provided by Kernel)
msg_count (queue header only)			
semaphore (queue header only)			
message_length	(20)	operation	(blind_send)
remote_timeout (valid only where operation = acknowledged_send)			——
sender			*(Merlin) node_number => 1; process_number => -32_766*
receiver			(Vivian) node_number => 2; process_number => -32_766
message_tag			(type_2_message_tag) 2
message_ID (valid only where operation = acknowledged_send)			——
checksum			——

"type 2 message "

Figure 5-10: Datagram Structure - Part 1 of 2



next			
prev			
class	(small)	(small)	(small)
buffer_size	(smallest possible)	(smallest possible)	(smallest possible)
msg_count (3)	—	—	—
semaphore (—)	—	—	—
message_length	10	10	10
operation	blind_send	blind_send	acknowledged_send
remote_timeout	—	—	30 seconds
sender	(Vivian) 2; -32_766	(Arthur) 1; -32_765	(Vivian) 2; -32_766
receiver	(Merlin) 1; -32_766	(Merlin) 1; -32_766	(Merlin) 1; -32_766
message_tag	(message_type_1_tag) 1	(message_type_1_tag) 1	(message_type_1_tag) 1
message_ID	—	—	0
checksum	—	—	—
buffer	"type1 msg"	"type1 msg"	"type1 msg"

Vivian called: *send_message* for type 1 message to Merlin

Arthur called: *send_message* for type 1 message to Merlin

Vivian called: *send_message_and_wait* for type 1 message to Merlin

Scenario:On processor a, Merlin's incoming message queue contains the following three messages:

(1) type 1 message
from Vivian on processor b via a call to the Kernel_primitive *send_message*

(2) type 1 message
from Arthur on processor a via a call to the Kernel_primitive *send_message*

(3) type 1 message
from Vivian on processor b via a call to the Kernel_primitive *send_message_and_wait* with a remote timeout of 5 seconds

Figure 5-10: Datagram Structure - Part 2 of 2

There are no representation specifications relevant to a *datagram* at this level.

```
--
-- The actual datagram comprises the header field and a data buffer. The
-- size of the data buffer depends on the class.
--

type datagram is
    record
        header :        datagram_header;
        buffer :        data_buffer(1 .. CG.message_length_type'last);
    end record;


--
-- Each datagram header comprises the following:
--
-- next
--   A pointer to the next datagram in a linked list. Both next and
--   prev define a doubly linked circular list. This field and a number
--   of fields following are not transmitted across the network. They are
--   used only on the node which actually "owns" the datagram, and are not
--   considered part of the message that is shipped between processors.
--
--   Default value:
--      none
--
--   Modified by:
--      The routines in datagram_management. This value should not be
--      touched by any other routines.
--
-- prev
--   A pointer to the previous datagram in a linked list. Both next and
--   prev define a doubly linked circular list. This field is not
--   transmitted across the network, and is maintained locally by each
--   processor node.
--
--   Default value:
--      none
--
--   Modified by:
--      The routines in datagram_management. This value should not be
--      touched by any other routines.
--
-- class
--   The type of datagram. There are four different types of datagram
--   buffers that are used (we simulate a variant record). The class field
--   denotes the type of datagram buffer that is attached to the header.
--   This field is not transmitted across the network, and is maintained
--   locally by each processor node.
--
--   Possible values:
--      small (a small sized user datagram buffer)
--      large (a large sized user datagram buffer)
--      kernel (a datagram buffer guaranteed to hold the largest
--            kernel message)
--      queue_head (datagram contains no buffer - used only as the
--          head of a queue)
--
--    Default value:
--      none
--
--    Modified by:
--      No one.  This value is set at initialization time and should not be
--      touched by any other routines.
--
-- buffer_size
--    The size of the data buffer of this datagram.  The message in the
--    datagram (see "message_length") may be less than or equal to this
--    size field.  This field is not transmitted across the network, and
--    is maintained locally by each processor node.
--
--    Default value:
--      none
--
--    Modified by:
--      No one.  This value is set at initialization time and should not be
--      touched by any other routines.
--
-- msg_count
--    The number of messages present in a queue.  This field is used
--    only for queue headers.  In order to make type checking work, a queue
--    header is simply another type of datagram (which is never transmitted).
--    The msg_count keeps track of how many datagrams are in a queue.  This
--    field is not transmitted across the network, and is maintained
--    locally by each processor node.
--
--    Default value:
--      none
--
--    Modified by:
--      The routines in datagram_management.  This value should not be
--      touched by any other routines.
--
-- semaphore
--    A flag (or pair of flags, depending on the actions of LLH.P)
--    which locks a queue header against interprocessor access.  This field is
--    used only in queue headers, and is not transmitted across the network.
--
--    Default value:
--      none
--
--    Modified by:
--      The routines in datagram_management.  This value should not be
--      touched by any other routines.
--
-- message_length
--    The actual size (number of bytes) of the message that is in the
--    data_buffer.  This value will always be less than or equal to the
--    buffer_size field.  This field, and all following fields are transmitted
--    across the network, and are considered part of the message that is sent
--   and received.
--
--   Default value:
--      none
--
--   Modified by:
--      The routines in bus_io and communication_management.  May be freely
--      set (with care) by other routines implementing communications.
--
-- operation
--   The Kernel operation defined in the KAM 10.2.1
--
--   Possible values:
--      See the enumerated type kernel_operation defined above
--
--   Default value:
--      none
--
--   Modified by:
--      The routines in bus_io and communication_management.  May be freely
--      set (with care) by other routines implementing communications.
--
-- remote_timeout
--   The timeout time specified for a send_message_and_wait.  This
--   field is passed to the receiving processor, which enqueues the timeout
--   event remotely.
--
--   Default value:
--      none
--
--   Modified by:
--      The routines in bus_io and communication_management.  May be freely
--      set (with care) by other routines implementing communications.
--
-- sender
--   The process_index of the process which is sending the datagram.
--   Because the sender field contains both a process number and a processor
--   number, the sender field uniquely identifies a process within the DARK
--   network.
--
--   Default value:
--      none
--
--   Modified by:
--      The routines in bus_io and communication_management.  May be freely
--      set (with care) by other routines implementing communications.
--
-- receiver
--   The process_index of the process to which the datagram is being sent.
--   Because the sender field contains both a process number and a processor
--   number, the sender field uniquely identifies a process within the DARK
--   network.
--
--   Default value:
--      none
```

```
--
--  Modified by:
--     The routines in bus_io and communication_management. May be freely
--     set (with care) by other routines implementing communications.
--
-- message_tag
--  Overloaded to contain either the message tag supplied by the
--  user, or for a Kernel tag defined in KAM 10.3.1
--
--  Possible values:
--     When a datagram has an operation field of kernel_message,
--     init_protocol_message, or sync_protocol_message, the possible
--     values of this field are from the enumerated type kernel_tag, listed
--     above. When the datagram has an operation field of blind_send or
--     acknowledged_send, the possible values of this field are any value
--     from the type CG.message_tag_type.
--
--  Default value:
--     none
--
--  Modified by:
--     The routines in bus_io and communication_management. May be freely
--     set (with care) by other routines implementing communications.
--
-- message_id
--  A Kernel sequencing number used in send_message_and_wait to
--  insure that ACK and NAK messages are associated with the correct send.
--
--  Default value:
--     none
--
--  Modified by:
--     The routines in bus_io and communication_management. May be freely
--     set (with care) by other routines implementing communications.
--
-- checksum
--  A check value calculated by the sending Nproc and verified by
--  the receiving Nproc.
--
--  Default value:
--     none
--
--  Modified by:
--     The Nproc code. This value should not be touched by any other
--     routines.
--

type datagram_header is
   record
      next :            datagram_pointer;      -- Not transmitted
      prev :            datagram_pointer;      --  "        "
      class :           datagram_class;        --  "        "
      buffer_size :     buffer_range;          --  "        "
      msg_count :       hw_long_natural;       --  "        "
      semaphore :       hw_long_integer;       -- Not transmitted
```

```
        message_length :         buffer_range;
        operation :              kernel_operation;
        remote_timeout :         KT.kernel_time;
        sender :                 NG.process_index_type;
        receiver :               NG.process_index_type;
        message_tag :            CG.message_tag_type;
        message_id :             message_identifier;
        checksum :               hw_integer;
    end record;
```

The following representation specifications define the layout of the *datagram_header*. These are defined in a hardware-independent manner but used internally in a hardware-dependent manner. The representation specifications are used to ensure that the layout of a datagram is always the same, no matter what hardware or compilation system is used. See also the documentation that will be provided with the code for a discussion of the translation of *sender* and *receiver* addresses into a format that is used by the underlying hardware.

```
for datagram_header use
    record
        next                    at  0*longword range  0 .. 31;
        prev                    at  1*longword range  0 .. 31;
        class                   at  2*longword range  0 .. 15;  --16..31 unused
        buffer_size             at  3*longword range  0 .. 31;
        msg_count               at  4*longword range  0 .. 31;
        semaphore               at  5*longword range  0 .. 31;
        message_length          at  6*longword range  0 .. 15;
        operation               at  6*longword range 16 .. 31;
        remote_timeout          at  7*longword range  0 .. 63;
        sender                  at  9*longword range  0 .. 31;
        receiver                at 10*longword range  0 .. 31;
        message_tag             at 11*longword range  0 .. 31;
        message_id              at 12*longword range  0 .. 31;
        checksum                at 13*longword range  0 .. 31;
    end record;

for datagram_header'size use 14*longword*bits_per_byte;
```

The following data types are used for components of the *datagram_header*.

```
--
-- Enumerated type describing the different type of messages that can be
-- sent, either originating as a application message or as a Kernel message.
--
-- The values of this enumerated type are:
--
--   blind_send
--      The kernel operation which corresponds to the send_message routine.
--
--   acknowledged_send
--      The kernel operation which corresponds to the send_message_and_wait
--      routine.
--
--   kernel_message
--      Any message sent by the kernel to respond to send_message_and_wait
--      actions, or to kill (or announce the death of) processes.
--
--    init_protocol_message
--      Any message sent during the initialization procedure before the
--      application code is running.
--
--    sync_protocol_message
--      Any message sent during the global clock resynchronization protocol.
--

type kernel_operation is (
          blind_send,
          acknowledged_send,
          kernel_message,
          init_protocol_message,
          sync_protocol_message
          );


--
-- Enumerated type describing the various Kernel-to-Kernel messages.  This
-- value is written into the "message_tag" field of a datagram
--
-- The values of this enumerated type are:
--
--    ack
--      A kernel message acknowledging the timely receipt of an
--      acknowledged_send message (i.e., the message was received).
--
--    nak
--      A kernel message specifying that an acknowledged_send message
--      was not received in a timely manner (i.e. the message timed out
--      before it was received).
--
--    nak_process_dead
--      A kernel message specifying that an acknowledged_send message
--      was not received because the process to which it was addressed
--      died before receipt.
--
--    info_process_dead
--      A kernel message specifying that a process has died.  This message
--      is sent independently of an acknowledged_send.
--
--    kill_process
--      A message specifying that a remote process is to be terminated.
--
--    init_complete
--      A Kernel message indicating that the network initialization
--      procedure has completed, and that the Kernel may begin running
--      the application code.
--
--    network_failure
--      A initialization message indicating that a network failure has been
--      detected.
--
```

```
--  process_created
--     An initialization message announcing the creation of a process.  This
--     message is sent via the create_process kernel call.
--
--  master_ready
--     An initialization message indicating that the temporary network
--     master is ready to resume the initialization protocol.
--
--  nct_count
--     An initialization message announcing the size of the NCT table that
--     is to follow.
--
--  nct_entry
--     An initialization message containing an NCT entry.
--
--  go_enclosed
--     An initialization message containing the epoch time.
--
--  go_acknowledgement
--     An initialization message acknowledging the previous go_enclosed.
--
--  prepare_to_sync
--     A synchronization protocol message announcing the start of the
--     clock resynchronization sequence.
--
--  ready_for_sync
--     A synchronization protocol message replying to the prepare_to_sync
--     message, announcing that resynchronization is in progress.
--
--  time_is_now
--     A synchronization protocol message announcing the epoch time to
--     resynchronize to.
--
--  now_in_sync
--     A synchronization protocol message acknowledging the receipt of the
--     time_is_now message.
--
--  abort_sync
--     A synchronization protocol message which prematurely terminates
--     the resynchronization procedure.
--

type kernel_tag is (
        ack,
        nak,
        nak_process_dead,
        info_process_dead,
        kill_process,
        init_complete,
        network_failure,
        process_created,
        master_ready,
        nct_count,
        nct_entry,
        go_enclosed,
        go_acknowledgement,
        prepare_to_sync,
        ready_for_sync,
        time_is_now,
        now_in_sync,
        abort_sync
        );

--
--  definitions of data used internal to a datagram
--

subtype buffer_range is CG.message_length_type;

--
-- data buffer that can be indexed in byte sized quantities
--

type data_buffer is array (buffer_range range <>) of hw_byte;

--
-- insure that the bytes of the array are contiguous
--

pragma pack (data_buffer) ;

--
-- this type allows the Kernel to generate unique message identifiers
-- for those messages sent via send_message_and_wait
--

type message_identifier is new hw_long_integer;

--
-- Datagrams are referenced by pointers for effiecient access and
-- parameter passing
--


--
-- Datagrams are divided into classes.  While variant records were an
-- Ada language option, the implementation of these records was so
-- inefficient that a design decision was made to use unchecked_conversion
-- on different datagram classes (identified by the datagram_class type)
-- to simulate a single datagram type.
--

type datagram_class is (small, large, kernel, queue_head);
```

5.2.1.3. Initialization

The pool of *datagrams* is completely allocated during Kernel initialization. It is from this pool that individual *datagrams* are assigned to hold incoming messages destined to local processes and outgoing messages sent from a local process. The size of this pool is determined by the hardware and is described in the documentation that will be provided with the code.

5.2.1.4. Additional Allocation Requirements

The application creates the header for an incoming message queue for each Kernel process during the execution of the Kernel primitive *create_process*. *Datagrams* are removed from the *datagram* pool and assigned to a Kernel process as messages are sent by or received for that process. No additional allocation is performed.

The Kernel returns *datagrams* it no longer needs to the *datagram* pool. No deallocation is performed.

5.2.1.5. Constraints on Usage

Datagrams should never be accessed directly by the application. All use of *datagrams* should be through the Kernel primitives *send_message, send_message_and_wait, receive_message, kill, synchronize*, and the initialization primitives.

Even though the *datagrams* are potentially visible data structures, they should be treated as an "abstract data type" by the application program. Knowledge of the internal structure, other than the fact that all messages are stored in FIFO order, should not be exploited in the application program, as this may violate the integrity of the Kernel and the application program.

5.2.2. Time Event Queue

The Time Event Queue maintains the Kernel's internal view of time. To the Kernel, time is represented as an ordered sequence of events that will happen some time in the future. The event with the least amount of time until expiration (i.e., the nearest in the future) appears at the head of the Time Event Queue; events with longer amounts of time until expiration appear toward the tail.

Three events, at most, can be outstanding for a process as a direct result of its own actions:

1. Alarm
2. Timeslice
3. One of: wait, semaphore, or receive timeouts (via calls to one of the Kernel primitives: *wait, claim*, or *receive_message*).

Additionally, a process may have any number of messages queued for its receipt (limited by the size of the incoming message queue) that require an acknowledgement to be returned to the sender.

5.2.2.1. Exporting Package

Generic_process_table, process_table

5.2.2.2. Structure

The Time Event Queue is a semi-dynamic data structure. Each entry in the Time Event Queue fully describes the event and its timing aspects.

Figure 5-11 illustrates the structure of the Time Event Queue along with the scenario it represents.

Scenario:The Time Event Queue on **processor a** has three entries, based on the occurrence of the following Kernel primitive calls:

- **Merlin** calls *wait* to timeout after 5 seconds.
- **Arthur** calls *set_alarm* with a timeout for 9:30 am.
- A message was sent to **Merlin** via *send_message_and_wait* with a remote timeout of 30 seconds.

Regardless of the order in which these events occur, the Time Event Queue is always ordered the same way.

Figure 5-11 Part 1 shows just the contents of the Time Event Queue; Figures 5-11 Part 2, Part 3, and Part 4 show the relationship between events in the Time Event Queue and other Kernel data structures.

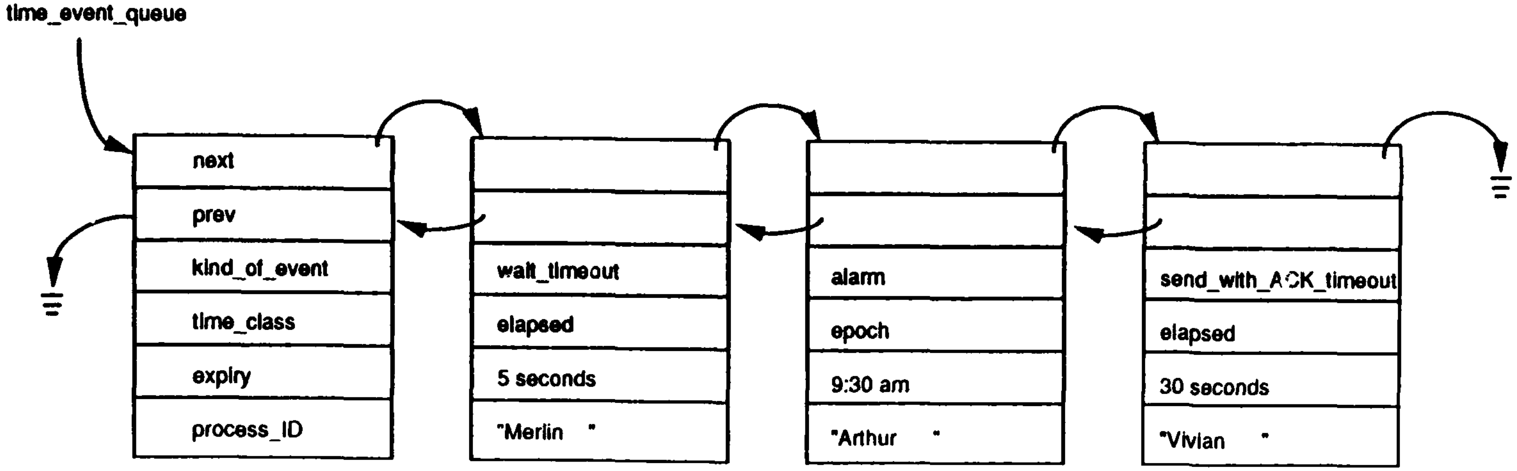


Figure 5-11: Time Event Queue Structure - Part 1 of 4

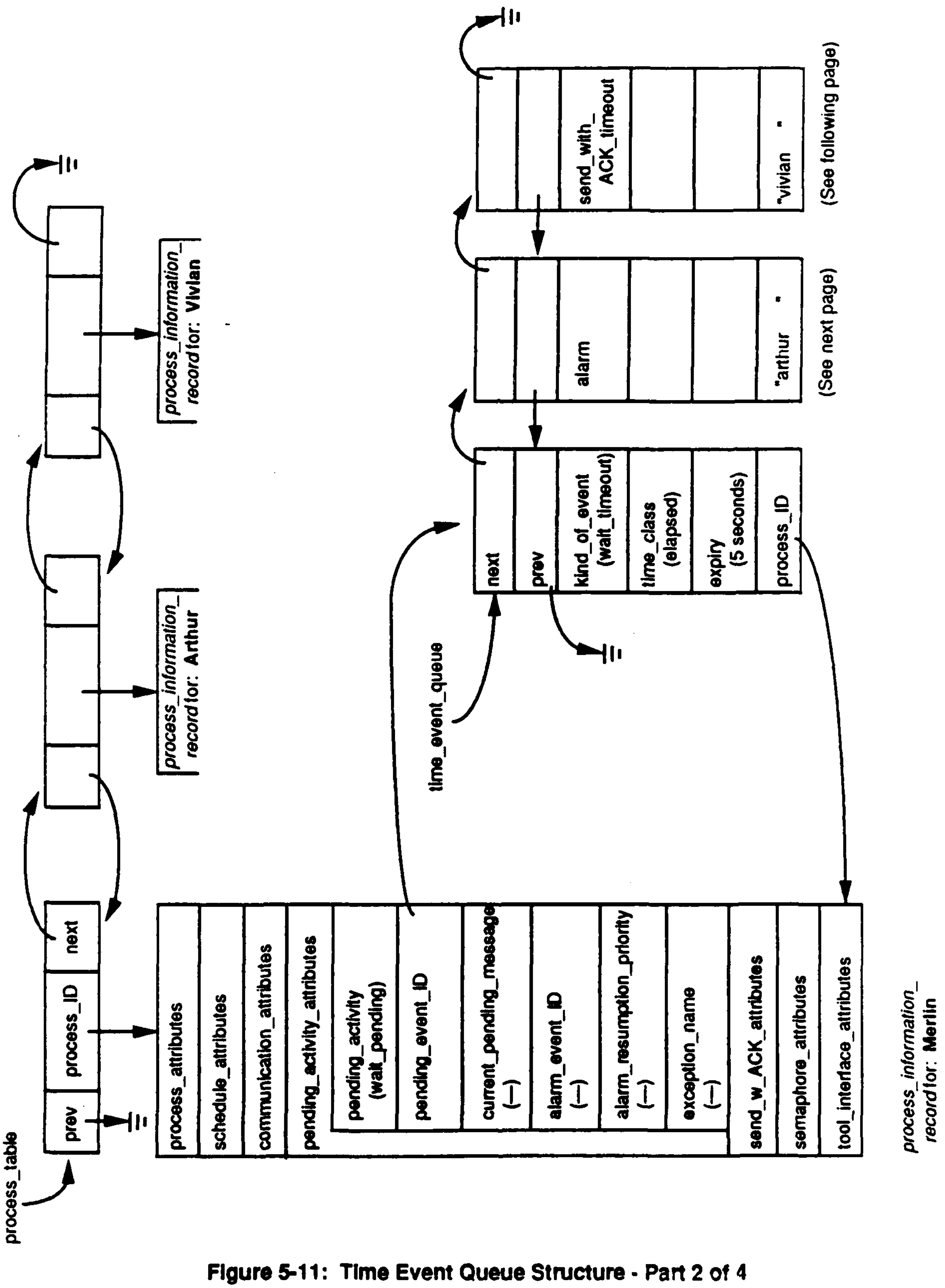


Figure 5-11: Time Event Queue Structure - Part 2 of 4

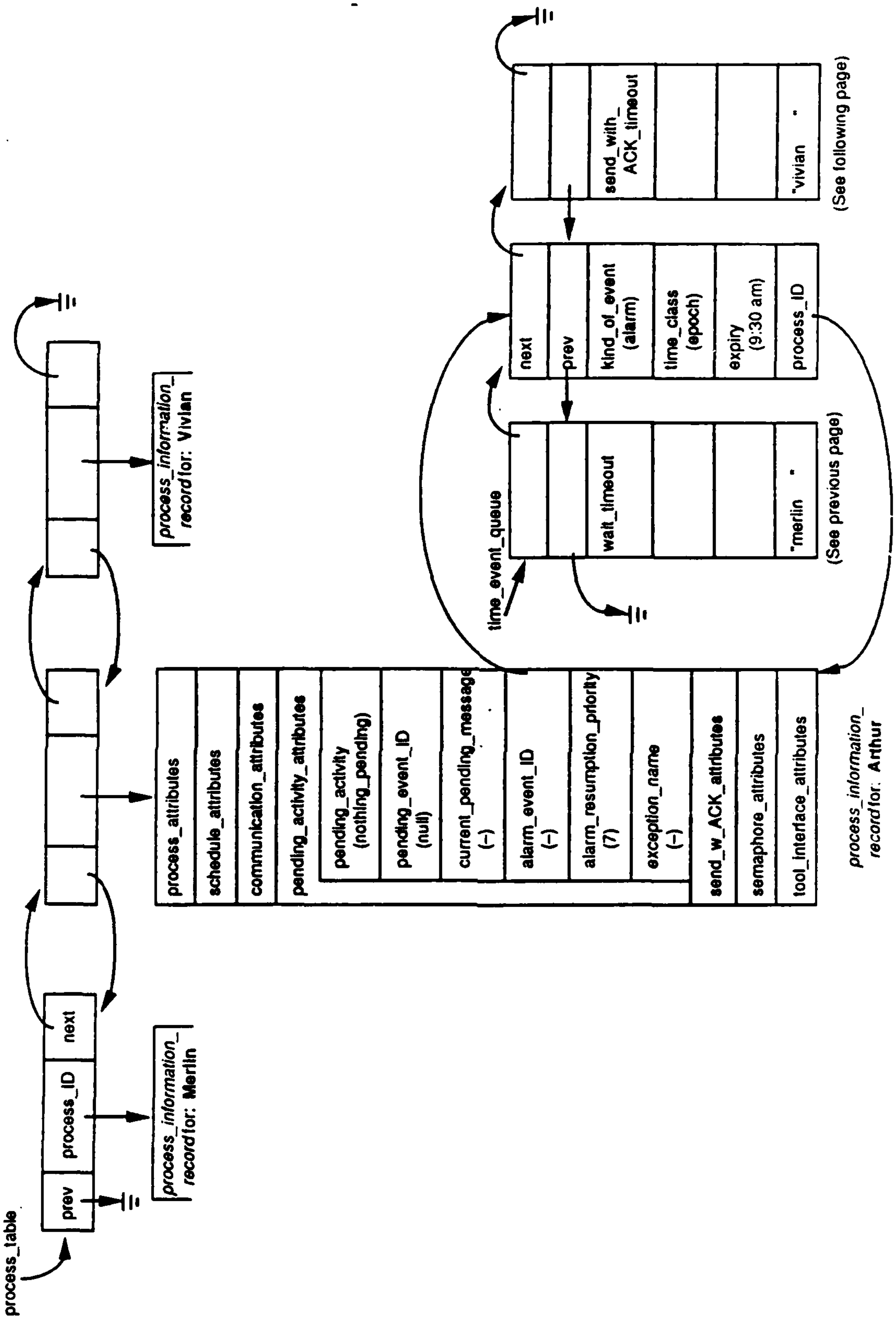
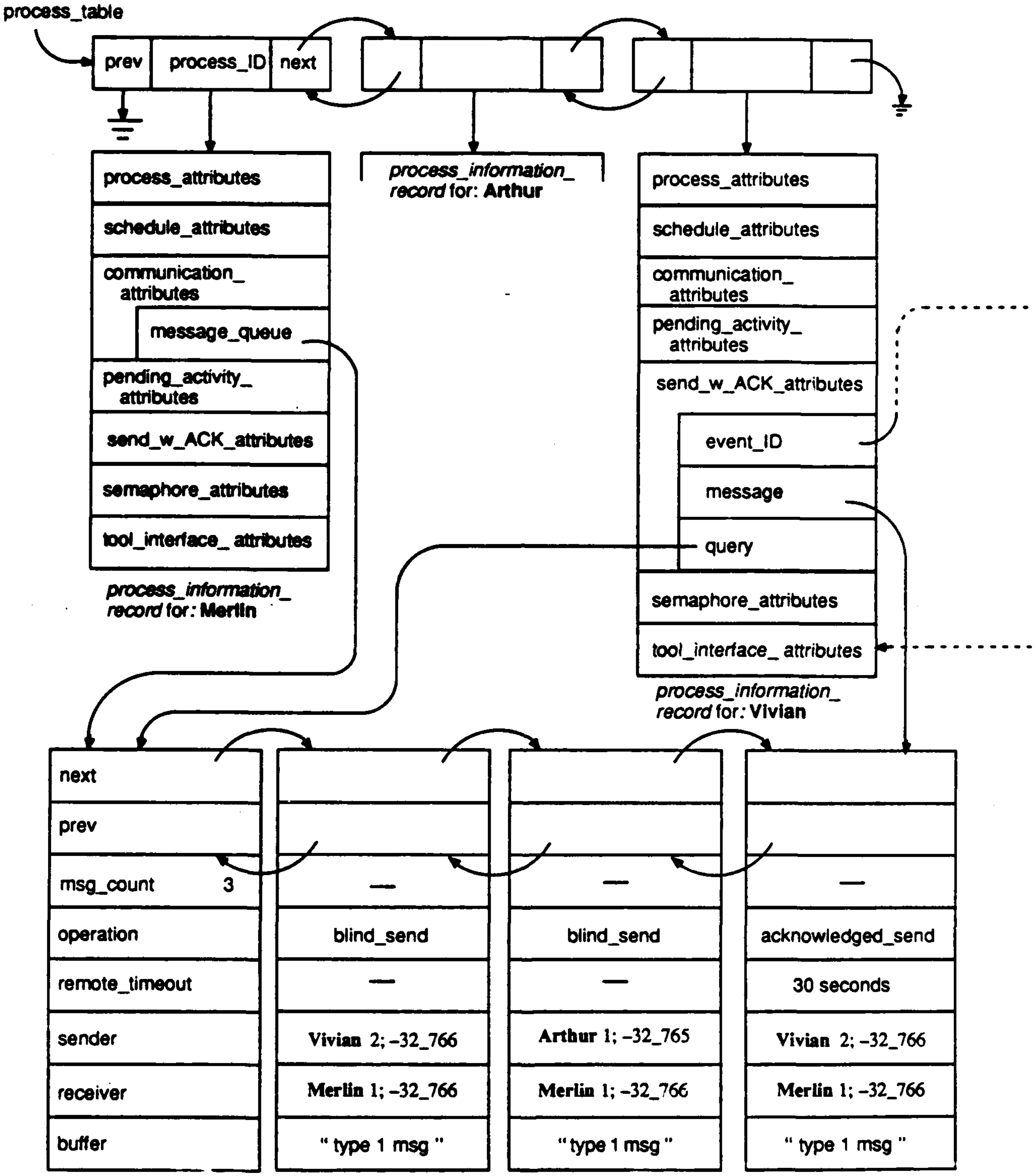


Figure 5-11: Time Event Queue Structure - Part 3 of 4


process_table
prev
process_ID
next
process_attributes
schedule_attributes
communication_ attributes
message_queue
pending_activity_ attributes
send_w_ACK_attributes
semaphore_attributes
tool_interface_ attributes
process_information_ record for: **Merlin**
process_information_ record for: **Arthur**
process_attributes
schedule_attributes
communication_ attributes
pending_activity_ attributes
send_w_ACK_attributes
event_ID
message
query
semaphore_attributes
tool_interface_ attributes
process_information_ record for: **Vivian**

next			
prev			
msg_count 3	—	—	—
operation	blind_send	blind_send	acknowledged_send
remote_timeout	—	—	30 seconds
sender	Vivian 2; -32_766	Arthur 1; -32_765	Vivian 2; -32_766
receiver	Merlin 1; -32_766	Merlin 1; -32_766	Merlin 1; -32_766
buffer	" type 1 msg "	"type 1 msg"	" type 1 msg "

Figure 5-11: Time Event Queue Structure - Part 4a of 4

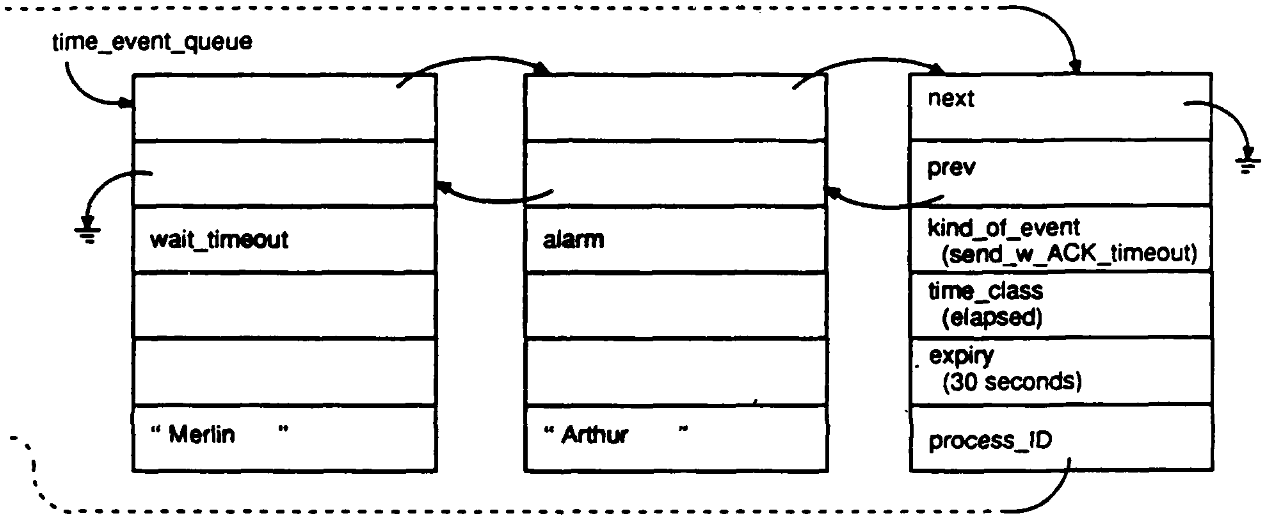


Figure 5-11: Time Event Queue Structure - Part 4b of 4

```
--
-- event_information is the template used to represent each of the sequence
-- of events that are understood by the Kernel to happen some time in the
-- future (i.e., the template for each entry in the time event queue)
--
-- the information maintained for each time event is:
--   kind_of_event
--     the event_type causing the creation of this time event
--
--     possible values:
--       alarm (via a call to set_alarm)
--       receive_timeout (via a call to receive_message)
--       semaphore_timeout (via a call to claim)
--       wait_timeout (via a call to wait)
--       send_with_ACK_timeout (by receiving a message indicating a
--         remote timeout is required)
--       slice_expiration (time slicing was previously enabled)
--
--     default value:
--       none
--
--     this value is assigned when an event_information record is added
--     to the time event queue and should never change as long as it is
--     still in the time event queue
--
--   time_class
--     the type of time specified by the application (via a call to any
--     Kernel primitive that provides a time parameter)
--
--     possible values:
--       elapsed (the application specified an elapsed_time)
--       epoch (the application specified an epoch_time)
--
--     default value:
--       none
--
--     this value is assigned when an event_information record is added
--     to the time event queue and should never change as long as it is
--     still in the time event queue
--
--   expiry
--     the Kernel_time at which the time event expires; the time in each
--     entry is that which was specified via the Kernel primitive called
--
--     default value:
--       none
--
--     this value is assigned when an event_information record is added
--     to the time event queue; it may change only via a call to one of
--     the Kernel primitives:  adjust_elapsed_time (for all entries in
--     the time event queue) or reset_epoch_time (only for those time
--     event queue entries with time_class = epoch)
--
--   process_ID
--     the concrete process identifier for which the time event is
--    maintained (i.e., a pointer into the process table for the
--    process "owning" this event)
--
--    default value:
--      none
--
--    this value is assigned when an event_information record is added
--    to the time event queue and should never change as long as it is
--    still in the time event queue
--

   type event_information
   is record
      kind_of_event : time_keeper_globals.event_type;
      time_class : time_keeper_globals.time_class_type;
      expiry : Kernel_time.Kernel_time;
      process_ID : process_identifier;
   end record;
```

5.2.2.3. Initialization

The initial allocation for the Time Event Queue is computed as the product of *maximum_number_of_processes_value* times two (to accommodate the alarm timeout and one of the other, mutually exclusive, timeouts) plus one (for the slice event). If no messages requiring acknowledgements (i.e., sent via the Kernel primitive *send_message_and_wait*) are ever received by this processor, the Time Event Queue will never grow.

5.2.2.4. Additional Allocation Requirements

With the exception of processing messages that require acknowledgements, no additional allocation is required. To manipulate the entries in the Time Event Queue, pointers are assigned and unassigned; no dynamic storage allocation is required.

Each time a message that requires an acknowledgement is received, there is the potential for allocating a new entry in the Time Event Queue. Once allocated, this entry is maintained in the Time Event Queue so that, once the timeout it represents expires, it may be reused by the Kernel. This avoids repeated calls for dynamic allocation.

The maximum size to which the Time Event Queue could ever grow is:

((number of locally created processes * 2) + 1) +

$$\sum_{i=1}^{number\_of\_Kernel\_nodes} \textit{number of locally created processes on node } i$$

-1

The first term is the maximum number of events that could be pending for each process as described in Section 5.2.2.3. The second term is the maximum number of processes that could conceivably send a message via the Kernel primitive *send_message_and_wait*. The third term excludes the process itself.

5.2.2.5. Constraints on Usage

The Time Event Queue should never be accessed directly by the application. It should only be manipulated via calls to the Kernel primitives (see Chapter 4 and Appendix A).

Even though the Time Event Queue is potentially a visible data structure, it should be treated as an "abstract data type" by the application program. Knowledge of its internal structure should not be exploited in the application program, as this may violate the integrity of the Kernel and the application program.

5.2.3. Process Index Table

The Process Index Table maintains the Kernel's mapping from the application-visible *process identifier* assigned to a process (and maintained in the Process Table) to an internal notation used for network communication. This internal notation is called the *process index*, and it comprises two fields that, together, uniquely identify each process declared to the Kernel:

1. *Node_number*: This corresponds to the *physical_address* field in the NCT entry corresponding to the processor on which the process is sited.
2. *Process_number*: This value, unique within each Kernel node, identifies each Kernel process declared on the processor designated by the *node_number* field.

The *process_index* for a non-Kernel device is:

1. *Node_number*: The *physical_address* field in the NCT entry corresponding to the non-Kernel device.
2. *Process_number*: A well-known null value (this is never used by the Kernel).

The Kernel initializes the Process Index Table during Kernel and network initialization time, and the Kernel uses that information for inter-processor communication.

5.2.3.1. Exporting Package

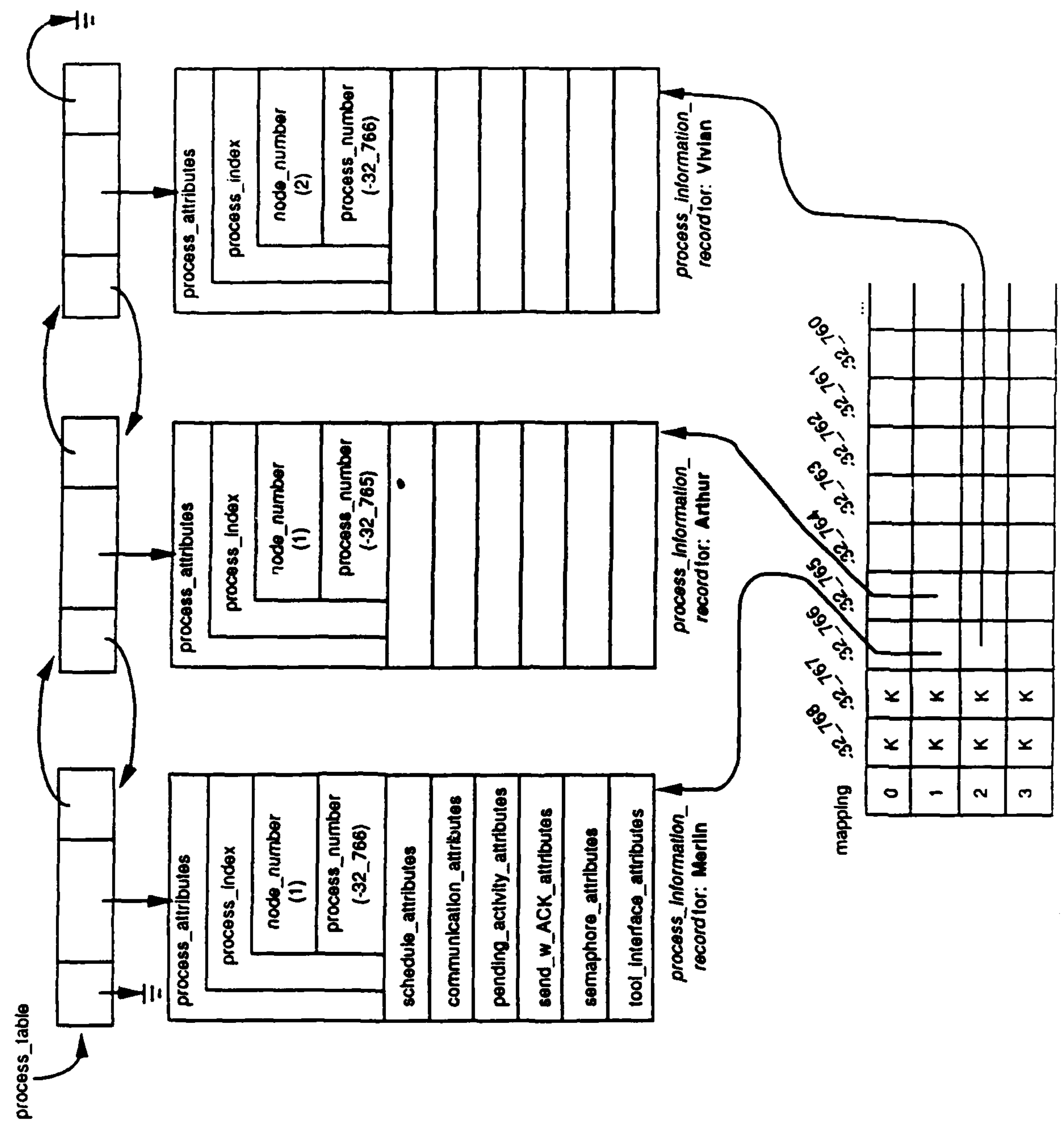
Process_index_table

5.2.3.2. Structure

The Process Index Table is a static data structure; it is fully defined at compile time. The Process Index Table is a two-dimensional array: the first dimension represents the physical address of the nodes in the network; the second dimension represents the number of the process on a particular node. This index into the Process Index Table identifies a unique *process identifier* component. The *process_information_record* pointed to by the *process identifier* identified by the Process Index Table indices contains a component—*process_attributes.process_index*—that corresponds to the two-dimensional index into the Process Index Table. If there is no process at the component identified by an index into the Process Index Table, that value is null (i.e., no such process exists).

Figure 5-12 illustrates the structure of the Process Index Table along with the scenario it represents.

```
type mapping_array_type is array
(
```



Scenario:
Contents of the
Process Table and
Process Index Table
after execution of
processor_a_Main_Unit -
the Main Unit on processor a.

Figure 5-12: Process Index Table Structure

```
   NG.bus_address range <>, hw_integer range <>
) of PTB.process_identifier;

--
-- Compute the upper bounds on the mapping table so that it can be properly
-- sized.
--
-- largest_node_number:   this is be the number of nodes in the network.
-- largest_process_index: this the maximum number of processes that the
--                        application can create (as defined by the
--                        application)

largest_node_number : constant NG.bus_address :=
   NG.bus_address (NC.number_of_nodes - 1);

largest_process_index : constant hw_integer :=
   hw_integer'first + hw_integer (PTB.maximum_number_of_processes) + 5;

mapping : mapping_array_type
(
   0 .. largest_node_number, hw_integer'first .. largest_process_index
) := (others => (others => null));
```

There are no representation specifications relevant to the Process Index Table.

5.2.3.3. Initialization

All components of the Process Index Table are initialized to null when the Process Index Table is declared. Components are set to the appropriate *process identifier* value when a process is created locally or remotely (for Kernel processes) or when a process is associated with a non-Kernel device via the Kernel primitive *allocate_device_receiver.*

5.2.3.4. Additional Allocation Requirements

No additional allocation is required. The maximum size of the Process Index Table is constrained by the tailoring parameters: *maximum_number_o,_processes_value* (the maximum number of processes that are known to the Kernel on this node) and *number_of_nodes_value* (the number of entries in the NCT).

The size of the Process Index Table may vary from node to node. The maximum size of the Process Index Table after intitialization is complete may be limited by the tailoring parameter *maximum_number_of_processes_value.* See Section C.1.4 for more details.

5.2.3.5. Constraints on Usage

The Process Index Table should never be accessed directly by the application. All use of the Process Index Table should be through Kernel primitives.

Even though the Process Index Table is a potentially visible data structure, it should be treated as an "abstract data type" by the application program. Knowledge of the internal structure should not be exploited in the application program, as this may violate the integrity of the Kernel and the application program.

5.2.4. Interrupt Table

Interrupt management relies primarily on one data structure called the Interrupt Table. It is a fixed size, statically allocated structure with entries for each interrupt that could possibly occur. Otherwise, no other data structure is referenced.

5.2.4.1. Exporting Package

Generic_interrupt_globals, *interrupt_globals*

5.2.4.2. Structure

The Interrupt Table contains all information needed for interrupt management. The table consists of a number of entries, one for each legal interrupt name. See Appendix H for the 68020-specific definition of type *interrupt_name*, which is used to determine the size of the Interrupt Table on the 68020. Each entry contains information about the interrupt itself (its state, type, and source), whether or not it is reserved for use by the hardware or Ada runtime system, Kernel, or used by the application, and information about the handler associated with the interrupt (handler code, stack, indication of whether bound or not).

Figure 5-13 illustrates the structure of the Interrupt Table along with the scenario it represents.

```
-- the information maintained about interrupts includes:
--
--  owner
--    the owner of the interrupt
--
--    values include:
--      absent (not assigned to any interrupting device)
--      reserved (reserved to Ada runtime or other non-Kernel system code
--      Kernel (owned by Kernel)
--      application (owned by application)
--
--    default value:
--      application (owned by application)
--
--  condition
--    the condition of the interrupt
--
--    values include:
--      bound (a handler has been associated with the named interrupt
--           via a Kernel call)
--      unbound (a handler has not been associated with the named
--           interrupt via a Kernel call)
--
--    default value:
--      unbound (a handler has not been associated with the named
--           interrupt via a Kernel call)
--
--  state
--    the state of the interrupt
--
--    values include:
--      enabled (the interrupt is enabled)
--      disabled (the interrupt is disabled)
```

Scenario: Contents of the Interrupt Table as instantiated by package *interrupt_globals*.

	0	...	12	...	72	73	...	253	254	255
owner	application		application		application	application		application	application	application
condition	unbound		unbound		unbound	unbound		unbound	unbound	unbound
state	disabled		disabled		disabled	disabled		disabled	disabled	disabled
source	external		external		external	external		external	external	external
can_preempt	T		T		T	T		T	T	T
handler	null_handler		null_handler		null_handler	null_handler		null_handler	null_handler	null_handler

*Example range and initialization are specific for the DARK testbed as described in Appendix I.

In this example, *interrupt_names* 0, 12, 72, 73, and 255 are all not available to the application; only *interrupt_names* 253 and 254 are available to the application.

Figure 5-13: Interrupt Table - Part 1 of 3

Scenario: Contents of the Interrupt Table after the Kernel initializes itself.

	0	...	12	...	72	73	...	253	254	255
owner	reserved		reserved		Kernel	Kernel		application	application	Kernel
condition	bound		bound		bound	bound		unbound	unbound	bound
state	enabled		enabled		enabled	enabled		disabled	disabled	enabled
source	external		internal		external	external		external	internal	external
can_preempt	T		F		F	T		T	T	T
handler	used by hardware		Ada runtime		clock. clock_I/h' address	time_keeper. event_I/h' address		null_handler	null_handler	bus_io. receive_ datagram_ I/h'address

*Example range and initialization are specific for the DARK testbed as described in Appendix I.

In this example, *interrupt_names* 0, 12, 72, 73, and 255 are all not available to the application; only *interrupt_names* 253 and 254 are available to the application.

Figure 5-13: Interrupt Table - Part 2 of 3

Scenario: Contents of the Interrupt Table after the application has executed the following sequence of Kernel primitive calls:

```
interrupt_management.bind_interrupt_handler
(
 interrupt => 253,
 handler_code =>
   hardware_interface'hw_address
     (IH253'address),
 can preempt => false
);
interrupt_management.bind_interrupt_handler
(
 interrupt =>254,
 handler_code =>
   hardware_interface.hw_address
     (IH254'address),
);
interrupt_management.enable
(
 interrupt => 254
);
interrupt_management.simulate_interrupt
(
 interrupt => 254
);
```

	0	...	12	...	72	73	...	253	254	255
owner	reserved		reserved		Kernel	Kernel		application	application	Kernel
condition	bound		bound		bound	bound		bound	bound	bound
state	enabled		enabled		enabled	enabled		disabled	enabled	enabled
source	external		internal		external	external		external	external	external
can_preempt	T		F		F	T		F	T	T
handler	used by hardware		Ada runtime		clock. clock_i/h' address	time_keeper. event_i/h' address		IH253' address	IH254' address	bus_io. receive_ datagram_ i/h'address

*Example range and initialization are specific for the DARK testbed as described in Appendix I.

In this example, *interrupt_names* 0, 12, 72, 73, and 255 are all not available to the application; only *interrupt_names* 253 and 254 are available to the application.

Figure 5-13: Interrupt Table - Part 3 of 3

```
--
--     default value:
--       disabled (the interrupt is disabled)
--
--   interrupt source
--     indication of from where the interrupt sequence originated
--
--     values include:
--       internal (the interrupt was generated by a call of the Kernel
--              primitive simulate_interrupt)
--       external (the interrupt was generated external to the processor
--              as some hardware interrupt)
--
--     default value:
--       external (the interrupt was generated external to the processor
--              as some hardware interrupt)
--
--   can_preempt
--     indication of whether or not the interrupt can cause the currently
--     running process to be preempted, letting another process execute
--     after interrupt processing
--
--     values include:
--       true  (the interrupt is preemptive; the Kernel Scheduler may
--            select a different process to continue)
--       false (the interrupt is non-preemptive; the Kernel Scheduler may
--            not select a different process to continue)
--
--     default value:
--       true  (the interrupt is preemptive; the Kernel Scheduler may
--            select a different process to continue)
--
--   interrupt_handler
--     interrupt service routine for the interrupt
--
--     default value:
--       null_handler (a well-known null address)
--
--   tool_interface
--     indication of whether or not the tool interface has been enabled
--
--     values include:
--       true (tool interface enabled)
--       false (tool interface not enabled)
--
--     default value:
--       false (tool interface not enabled)
--
--   monitoring_process_ID
--     the process identifier of the Kernel process monitoring interrupt
--     activity via the tool interface
--
--     default value:
--       null_process (a null process)
--
```

```
type interrupt_table_entry
is record
   owner                    : interrupt_owner := application;
   condition                : interrupt_condition := unbound;
   state                    : interrupt_state := disabled;
   source                   : interrupt_source := external;
   can_preempt              : Boolean := true;
   handler                  : hw_address := null_handler;
   tool_interface_enabled   : Boolean := false;
   monitoring_process_ID    : process_table.process_identifier :=
                                process_table.null_process;
end record;

--
-- the template for declaring the interrupt table is indexed
-- by the range of interrupt names
--

type interrupt_table_type
   is array (interrupt_name) of interrupt_table_entry;

interrupt_table : interrupt_table_type;
```

The following types are used in the definition of the Interrupt Table.

```
--
-- interrupt_name is an integer value in a hardware-dependent range;
-- see Appendix H for its definition for the 68020
--
-- these values correspond to the hardware vector assignments
-- of the target, and are generally sparse
--

type interrupt_name is (...);

--
-- an interrupt vector can be one of four kinds:
--   absent (not assigned to any interrupting device)
--   reserved (reserved to Ada runtime or other non-Kernel system
--             code)
--   Kernel (owned by Kernel)
--   application (owned by application)
--

type interrupt_owner is (absent, reserved, Kernel, application);

--
-- any interrupt may be in one of the following conditions:
--    bound (a handler has been associated with the named interrupt
--           via a Kernel call)
--    unbound (a handler has not been associated with the named
--             interrupt via a Kernel call)
--

type interrupt_condition is (bound, unbound);
```

```
--
-- this type indicates the state of an interrupt
--
-- values include:
--    enabled (the interrupt is enabled)
--    disabled (the interrupt is disabled)
--
--REQ:  11.1.4; 11.1.5
--PRIM:  20.1.1; 20.1.2; 20.1.3
--

type interrupt_state is (enabled, disabled);

--
-- this type indicates the source of an interrupt
--
-- values include:
--    internal - interrupt was generated by a call to the Kernel
--               primitive simulate_interrupt
--    external - interrupt was generated by hardware and was
--               processed by the hardware interrupt handling
--               mechanism
--

type interrupt_source is (internal, external);

--
-- this constant designates a null interrupt handler
--

null_handler : constant hw_address := null_hw_address;
```

There are no representation specifications relevant to the *interrupt_table_type*.

5.2.4.3. Initialization

The Kernel automatically initializes those entries in the Interrupt Table that are reserved for use by the hardware, the Kernel, or the Ada runtime environment.

No explicit initialization is required by the application, other than binding, enabling, or disabling interrupts as required by the application.

5.2.4.4. Additional Allocation Requirements

None; all data structures are static.

5.2.4.5. Constraints on Usage

The target interrupt mechanism should not be accessed directly through assembly routines or any Ada language feature without understanding the implications. Otherwise, certain assumptions made by the Kernel might be invalidated.

Because the Kernel primitive *simulate_interrupt* does not rely on the target interrupt mechanism but rather goes directly to the Interrupt Table to locate the appropriate ISR, the hardware is not aware that interrupt handling is in progress. Therefore, the ISR may exhibit a slightly different behavior when simulated if it depends on the internal mode of the target hardware.

5.2.5. Kernel Time

The model of time exported by the Kernel to the application was introduced in Section 2.5. This model of time is derived from the Kernel's internal representation of time, which is not visible to the application program except via *elapsed_time* and *epoch_time* abstractions. This Section presents the Kernel's internal representation of time.

It is possible that the Kernel's representation of time is insufficient to support a specific application. Appendix C.1.2 and Appendix C.2.3 each describe the analysis that must occur prior to using the Kernel for an application. This analysis assists the application engineers in determining a suitable value for the length of a slice—that is, the smallest interval of time required by an application—and the capability for the Kernel's representation of time to support it.

5.2.5.1. Exporting Package

Generic_Kernel_time, Kernel_time

5.2.5.2. Structure

Values of type *Kernel_time* and the application visible abstractions based on Kernel time, *elapsed_time* and *epoch_time*, are fully defined at compile time. They comprise two 32-bit parts: a *low* part and a *high* part that, combined, support representations of time beginning at zero and continuing for some 150_000 years (i.e., 2 ** 63 microseconds). The Kernel does perform error checking to ensure that values of type *Kernel_time*, *elapsed_time*, and *epoch_time* do not overflow. If an overflow is detected (during an arithmetic operation), the predefined exception *numeric_error* is raised.

```
--
-- type Kernel_time; the time on which elapsed time and epoch time
-- abstractions are built
--
-- this time representation allows applications beginning at time
-- zero to execute for some 150_000 years (i.e., 2 ** 63 microseconds)
--
-- ********
-- the user should ensure that adjusting any times does not approach
-- the limit of this time representation; proper Kernel functioning
-- is NOT guaranteed if time overflows
-- ********
--
-- Kernel_Time is represented as a signed 64-bit binary integer,
-- representing a count of microseconds.  Hence, a kernel time
-- of 1_000_000 corresponds to one second.
--
-- For the purposes of Ada definition, kernel_time is a record of
-- two components, being respectively the low-order and high-order
-- 32 bits.  The high-order component can be correctly declared to
-- be a signed 32-bit integer, of type hw_long_integer.  The low-order
-- component, however, is properly an UNSIGNED 32-BIT INTEGER, which
-- this Ada compiler will not accept.  Accordingly, it must be declared
-- to be SIGNED, which is wrong.
-- The consequences are these
--
-- (a) if a kernel time value is printed in the "obvious" manner, i.e., by
--     printing each component.  The low-order part may be printed as a
--     negative number when in fact it is a large positive number.
--
-- (b) if a kernel time value is constructed "by hand," i.e., as an
--     aggregate of two integers, the person writing the aggregate
--     must perform the necessary conversion from signed to unsigned
--     form For example, a kernel time of "-1" is represented as
--     16#ffffffff_ffffffff#, i.e., as (-1,-1) in Ada terms.

type Kernel_time is record
   low : hw_long_integer := 0;
   high: hw_long_integer := 0;
end record;

--
-- the value of zero for the Kernel_time abstraction
--

zero_Kernel_time : constant Kernel_time := (0, 0);

--
-- the range of durations represented as integral values
--

type integral_duration is new hw_long_integer;
```

There are representation specifications relevant to type *Kernel_time*. The representation specification is used to ensure that each portion of *Kernel_time* is aligned on a longword boundary and that it occupies a complete longword (i.e., 32 bits).

```
for Kernel_time use record
   low  at 0 range 0..31;
   high at 4 range 0..31;
end record;
```

5.2.5.3. Initialization

Objects of type *Kernel_time* or of those types derived from it, *elapsed_time* and *epoch_time* are initialized to zero. The size of those objects is completely determined at compilation time.

5.2.5.4. Additional Allocation Requirements

No additional allocation is required.

5.2.5.5. Constraints on Usage

The components of Kernel_time or of those types derived from it, elapsed_time and epoch_time, should never be accessed directly by the application. Kernel_time should never be referenced by the application; only the abstractions based on *Kernel_time, elapsed_time* and *epoch_time* should be used. See Section C.1.2 for more information about tailoring the representation of *Kernel_time* to the hardware and to the application.

Even though the representation of *Kemel_time*, *elapsed_time* and *epoch_time* are potentially visible, they should be treated as "abstract data types" by the application program. Knowledge of the internal structure of any of these types should not be exploited in the application program, as this may violate the integrity of the Kernel and the application program.

6. Application Evaluation

The detailed information in this chapter will be provided in the next version of this document.

6.1. Tool Interface

6.1.1. Concept of Operations

The Kernel is a utility intended to support the building of distributed Ada applications. As such, it is important that the Kernel be able to work in harmony with user-developed support tools. To provide that support, the Kernel must provide a window into its internal workings. It is envisioned that such a tool is simply another Kernel process executing on one or more of the processors in the network. As such, the tool has access to all the Kernel primitives. Using these primitives along with the Kernel-provided Tool Interface described below, a number of potential tools could be built, such as:

- Process Performance Monitor: To compile statistics about the runtime performance of a Kernel process(es).
- Processor performance monitor: To compile processor-level statistics.
- Network performance monitor: To compile network-level statistics.
- Interrupt activity monitor: To compile statistics on the frequency of interrupts and the amount of time spent in various interrupt handlers.
- Message performance monitor: To compile statistics about the frequency of messages, average message length, peak bus usage, etc.

Given the above motivation for the Tool Interface, the actual form of the Tool Interface is driven by the following principles:

- A user-developed tool must have easy access to all the information of the Kernel. Whether or not that tool makes use of the information is not the Kernel's concern. The key is that the Kernel must provide visibility into everything it knows intrinsically, without expending resources to combine that intrinsic knowledge in any way.
- The extraction of information based on what the Kernel knows is left to the user-developed tool (and indeed, it is deemed to be the function of that tool). It is in the domain of the tool where the intrinsic Kernel information is combined and presented in some context-specific manner.
- The internal Kernel information must be provided in a manner that does not compromise the integrity of the Kernel; this implies read-only access to the Kernel's internal data structures.
- The performance impact of using the Tool Interface must be predictable. Obviously, the performance impact will not be entirely predictable given the non-determinism inherent in the activities being monitored. But the Tool Interface bounds the impact in a way that provides insight into the potential performance impact of a user-developed tool (of course, the tool itself is a Kèrnel process that may be monitored like any other Kernel process in the system, so its performance may be determined empirically). The tool should consume predictable resources generally (not just clock cycles), e.g., storage, message bandwidth.

- Application code should *never* have to be modified simply to use a user-developed tool (while this may not always be possible, it is nevertheless a desirable goal). Therefore, while some of the information made available via the Tool Interface could be acquired by having the tool communicate directly with an application process, this approach is rejected as bad tool design and a distinct detriment to the application software of an embedded system. (The Ada Main Unit is used solely to configure a Kernel processor and to establish the initial process topology and, as such, is not considered "application code." To achieve the requisite separation of concerns, i.e., separation of the application from its monitoring, enabling or disabling Tool Interface functionality should be defined in the body of the Ada Main Unit.)

In general there are two classes of Kernel information that may be of interest to a user-developed tool: process information and interrupt information. The sections below describe the information available via the Tool Interface and the Kernel primitives provided to access this information.

6.2. Subprograms

Complete information will be provided in the next version of this document.

6.2.1. Begin_collection

Invocation

Resource Consumption

Conditions for Blocking

6.2.2. Cease_collection

Invocation

Resource Consumption

Conditions for Blocking

6.2.3. Read_process_table

Invocation

Resource Consumption

Conditions for Blocking

6.2.4. Read_interrupt_table

Invocation

Resource Consumption

Conditions for Blocking

6.3. Related Information

Complete information will be provided in the next version of this document.

6.3.1. Exported Constants

6.3.2. Exported Data Structures

6.3.3. Referenced Constants

6.3.4. Referenced Types

6.3.5. Relevant Generic Parameters

6.4. Monitoring Performance

Complete information will be provided in the next version of this document.

7. Notes

Section 7.1 is a general project glossary and, as such, may contain certain acronyms and terms that do not appear in this document specifically.

7.1. Glossary of Terms

Absolute (time):
A synonym for *epoch* time.

Ada: ANSI/MIL-STD-1815A.

ADT: Actual Delta Time. The actual delta time achieved when measuring a performance requirement.

AEGIS:
A class of Navy ships with a C^3 function.

AIT: Application Integrity Testing.

AITS: Application Integrity Test Suite.

Alarm:
A single timer associated with a process that may expire during process execution. If it does expire, a change of process state occurs, and the exception **alarm_expired** is raised.

ARTEWG:
Ada Runtime Environment Working Group.

Asynchronous (event):
An event that occurs while the *affected* process is performing other work or is waiting for the event.

Blocked (process state):
A process that is (temporarily) unable to run. All process states are described in the KFD.

Blocking (primitive):
A Kernel primitive that causes the process state to become blocked. The "blocked" process state is described in the KFD.

C^3: Command, control, and communications.

CPU: Central processing unit.

DARK:
Acronym for the SEI Distributed Ada Real-Time Kernel Project.

Dead (process state):
A process that is unable to run again. All process states are described in the KFD.

Device:
A hardware entity that can interrupt a processor or that can communicate over the system bus.

Distributed:
Executing on more than one processor in support of a single application.

DoD: U.S. Department of Defense.

Duration:
The Ada type *duration*; used to measure *elapsed* time. Related information can be found in the Ada Reference Manual [9.6].

EC: External Computer.

EDT: Expected Delta Time - The delta time specified in the performance requirements of the KFD.

Elaboration:
The elaboration of a *declaration* is the process by which the declaration achieves its effect (such as creating an *object*); this process occurs during program execution.

Elapsed (time):
The number of TICKs since the end of the application initialization process.

Epoch (time):
The value representing the moment at which the processors began to compute elapsed time.

Event:
Something that happens to a process (e.g., the expiry of a timer, the arrival of a message, the arrival of an acknowledgment, being killed by another process).

Exception:
An error situation which may arise during program execution.

FAR: Final Acceptance Review.

FIFO:
First in, first out.

GCD:
Greatest common divisor.

Hedgehog:
Echinus Europaeus L.

HM: Hardware Monitor.

Interrupt:
Suspension of a process caused by an event external to that process, and performed in such a way that the process can be resumed. (This external event is also called an interrupt.)

IDS: Interface Design Specification.

INS: Inertial Navigation System.

Interrupt handler:
Code automatically invoked by the Kernel in response to the occurrence of an interrupt.

ISO: International Standards Organization.

ISR: Interrupt service routine; interrupt handler.

Kernel:
Basic system software to provide facilities for a specific class of applications.

KAM: Kernel Architecture Model.

KFD: Kernel Facilities Definition.

KIT: Kernel Integrity Testing.

KITS: Kernel Integrity Test Suite.

KS: Kernel Specification.

KTC: Kernel Test Checklist

LIFO: Last in, first out.

MCCR:
Mission Critical Computer Resource.

MS: Motion Simulator (Part of the INS Simulation).

NAVSAT:
Navigation Satellite.

NTDS:
Navy Tactical Data System (A communications protocol).

Network:
Series of points (nodes, devices, processors) interconnected by communication channels.

NCT: Network Configuration Table.

NIT: Network Integrity Test.

Package:
A package specifies a group of logically related entities, such as *types*, *objects* of those types, and *subprograms* with *parameters* of those types. It is written as a *package declaration* and a *package body*. A package declaration is just a *package specification* followed by a semi-colon. A package is one of the kinds of *program unit*.

Package body:
Contains implementations of *subprograms* (and possibly *tasks* as other *packages* that have been specified in the package declaration.

Package Calendar:
The Ada Package Calendar [Ada Reference Manual 9.6, Appendix C].

Package specification:
Has a *visible part*, containing the *declarations* of all entities that can be explicitly used outside the package. It may also have a *private part* containing structural details that complete the specification of the visible entities, but which are irrelevant to the user of the package.

PITS: Processor Integrity Test Suite.

PM: Performance Monitor.

Postcondition:
An assertion that must be true after the execution of a statement or program component. Otherwise an exception is raised.

Pragma:
Conveys information to the Ada compiler. This definition is from the Ada Reference Manual.

Precondition:
An assertion that must be true before the execution of a statement or program component. Otherwise an exception is raised.

Primitive:
Basic Kernel action or datum.

Process (Kernel):
An object of concurrent execution managed by the Kernel outside the knowledge and control of the Ada runtime environment; a schedulable unit of parallel execution.

Process stack:
Built by the Kernel when creating a Kernel process. The process stack contains a stack plug (to prevent the propagation of unhandled exceptions), a dummy call frame (pointing to process termination code), and a place for process-local variables.

Processor:
Central processing unit (CPU); on the DARK prototype, a 68020.

RDAS:
Remote Data Acquisition System.

Real-time:
When it is done is as important as *what* is done.

RN: Requirement Number (as enumerated in the KFD).

RTE: Runtime Environment.

Runtime:
That fraction of elapsed time during which the processor is executing application code.

Running (process state):
A process that is executing on its processor.

SEI: Software Engineering Institute.

Semaphore:
A mechanism for controlling process synchronization, often used to implement a solution to the mutual exclusion problem.

Slice:
A schedulable interval of time.

SPM: Software Programmer's Manual.

Status code:
Generic term used to indicate the status of the execution of a Kernel primitive. A status code may correspond to an output parameter of some discrete type or to an exception.

STS: System Test Software.

Suspended (process state):
A process that is able to run, but cannot run because a process of higher or equal priority is running.

Synchronous (event):
An event that happens while a process is looking for that event.

System bus:
Communication medium connecting processors and devices.

Task:
An Ada language construct that represents an object of concurrent execution managed by the Ada Runtime Environment supplied as part of a compiler [Ada Reference Manual Chapter 9].

TC: Test Controller.

Tick:
The smallest resolvable interval of time used internally by the Kernel.

Time:
The Ada type *time*; see also epoch and elapsed.

UI: User Interface (part of the INS).

Appendix A: Kernel Packages

This appendix, which is bound separately, is a copy of the Kernel specification.

Appendix B: Kernel Exceptions

This appendix provides a set of indices to the exceptions that may be raised by executing Kernel primitives. The first index is sorted by exception name, the second index is sorted by package name, and the third index is sorted by raising subprogram. For all three indices, the following information is provided:

- The name of the exception,
- The name of the exporting package, and
- The Kernel subprogram that raised the exception under the conditions described in the Kernel specification, provided in Appendix A. For each subprogram in the Kernel specification, there is a section entitled "ERROR PROCESSING" in which the cause of the exception and the Kernel's ensuing actions are fully described.

Appendix B.1: Kernel Exceptions

Index of Kernel Exceptions		
Exception Name	Package Name	Raising Subprogram
alarm_expired	generic_alarm_management	raised by the Kernel when the alarm expires asynchronously
buffer_too_small_for_message	generic_communication_management	receive_message receive_message (waiting for) receive_message (waiting until)
bus_address_check	generic_network_globals	raised during elaboration if type *bus_address* is not within range
calling_unit_not_Main_Unit	generic_process_managers	declare_process (Kernel process) declare_process (non-Kernel device) create_process
calling_unit_not_Main_Unit	generic_processor_management	initialize_Master_processor initialize_subordinate_processor initialization_complete
change_results_in_negative_elapsed_time	generic_time_management	adjust_elapsed_time
change_results_in_negative_epoch_time	generic_time_management	adjust_epoch_time
claim_timed_out	generic_semaphore_management	claim (waiting for) claim (waiting until)
configuration_tables_inconsistent	generic_processor_management	initialize_Master_processor
final_sync_initialization_timeout_expired	generic_processor_management	initialization_complete
illegal_context_for_call	generic_alarm_management	cancel_alarm set_alarm (for) set_alarm (after)

Index of Kernel Exceptions

Exception Name	Package Name	Raising Subprogram
illegal_context_for_call	generic_communication_management	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until) receive_message receive_message (waiting for) receive_message (waiting until)
illegal_context_for_call	generic_process_attribute_modifiers	die set_process_preemption set_process_priority wait (for) wait (until)
illegal_context_for_call	generic_process_attribute_readers	get_process_preemption get_process_priority who_am_I
illegal_context_for_call	generic_semaphore_management	claim claim (waiting for) claim (waiting until) release
illegal_interrupt	generic_interrupt_management	bind_interrupt_handler disable enable enabled simulate_interrupt
illegal_interrupt_handler_address	generic_interrupt_management	bind_interrupt_handler
illegal_process_address	generic_process_managers	create_process
illegal_process_identifier	generic_process_managers	create_process
illegal_quantum	generic_timeslice_management	set_timeslice
insufficient_space	generic_process_managers	declare_process (Kernel process) declare_process (non-Kernel device) create_process

Index of Kernel Exceptions		
Exception Name	Package Name	Raising Subprogram
Master_initialization_timeout_expired	generic_processor_management	initialize_Master_processor
message_not_received	generic_communication_management	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
message_timed_out	generic_communication_management	send_message_and_wait (for) send_message_and_wait (until) receive_message (waiting for) receive_message (waiting until)
network_failure	generic_communication_management	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
network_failure	generic_processor_management	initialize_Master_processor initialize_subordinate_processor initialization_complete
network_failure	generic_time_management	synchronize
no_alarm_set	generic_alarm_management	cancel_alarm
no_interrupt_handler_bound	generic_interrupt_management	enable simulate_interrupt
no_Kernel_process_on_non_Kernel_device	generic_process_managers	create_process
no_message_available	generic_communication_management	receive_message (waiting for) receive_message (waiting until)
no_such_device_exists	generic_communication_management	allocate_device_receiver
not_my_semaphore	generic_semaphore_management	release
null_priority_range	generic_schedule_types	raised during elaboration if type *priority* has a null range
OK_but_time_already_passed	generic_time_management	adjust_epoch_time
process_already_created	generic_process_managers	create_process

Index of Kernel Exceptions		
Exception Name	Package Name	Raising Subprogram
process_already_exists	generic_process_managers	declare_process (Kernel process) declare_process (non-Kernel device)
process_initialization_failure	generic_processor_management	initialization_complete
process_maximum_exceeded	generic_processor_management	initialization_complete
processor_failed_to_ACK_go_message	generic_processor_management	initialize_Master_processor
processor_failed_to_transmit_NCT	generic_processor_management	initialize_Master_processor
receiver_dead	generic_communication_management	send_message send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
receiver_is_sender	generic_communication_management	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
receiver_never_existed	generic_communication_management	send_message send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
replacing_previous_allocation	generic_communication_management	allocate_device_receiver
replacing_previous_interrupt_handler	generic_interrupt_management	bind_interrupt_handler
reserved_interrupt	generic_interrupt_management	bind_interrupt_handler disable enable enabled simulate_interrupt
resetting_existing_alarm	generic_alarm_management	set_alarm (for) set_alarm (after)
subordinate_initialization_timeout_expired	generic_processor_management	initialize_subordinate_processor
synchronization_in_progress	generic_time_management	synchronize

Index of Kernel Exceptions		
Exception Name	Package Name	Raising Subprogram
synchronization_timeout_expired	generic_time_management	synchronize
unknown_non_Kernel_device	generic_process_managers	declare_process (non-Kernel device)

Index of Kernel Exceptions by Package

Package Name	Exception	Raising Subprogram
generic_alarm_management	alarm_expired	raised by the Kernel when the alarm expires asynchronously
generic_alarm_management	illegal_context_for_call	cancel_alarm set_alarm (for) set_alarm (after)
generic_alarm_management	no_alarm_set	cancel_alarm
generic_alarm_management	resetting_existing_alarm	set_alarm (for) set_alarm (after)
generic_communication_management	illegal_context_for_call	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until) receive_message receive_message (waiting for) receive_message (waiting until)
generic_communication_management	message_not_received	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
generic_communication_management	buffer_too_small_for_message	receive_message receive_message (waiting for) receive_message (waiting until)
generic_communication_management	message_timed_out	send_message_and_wait (for) send_message_and_wait (until) receive_message (waiting for) receive_message (waiting until)
generic_communication_management	network_failure	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
generic_communication_management	no_message_available	receive_message (waiting for) receive_message (waiting until)

Index of Kernel Exceptions by Package

Package Name	Exception	Raising Subprogram
generic_communication_management	no_such_device_exists	allocate_device_receiver
generic_communication_management	receiver_dead	send_message send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
generic_communication_management	receiver_is_sender	send_message_and_wait send_message_and_wait (for) send_message_and_wait (until)
generic_communication_management	receiver_never_existed	send_message send_message_ans_wait send_message_ans_wait (for) send_message_ans_wait (until)
generic_communication_management	replacing_previous_allocation	allocate_device_receiver
generic_interrupt_management	illegal_interrupt	bind_interrupt_handler disable enable enabled simulate_interrupt
generic_interrupt_management	illegal_interrupt_handler_address	bind_interrupt_handler
generic_interrupt_management	no_interrupt_handler_bound	enable simulate_interrupt
generic_interrupt_management	replacing_previous_interrupt_handler	bind_interrupt_handler
generic_interrupt_management	reserved_interrupt	bind_interrupt_handler disable enable enabled simulate_interrupt
generic_network_globals	bus_address_check	raised during elaboration if type *bus_address* is not within range

Index of Kernel Exceptions by Package

Package Name	Exception	Raising Subprogram
generic_process_attribute_modifiers	illegal_context_for_call	die set_process_preemption set_process_priority wait (for) wait (until)
generic_process_attribute_readers	illegal_context_for_call	get_process_preemption get_process_priority who_am_I
generic_process_managers	calling_unit_not_Main_Unit	declare_process (Kernel process) declare_process (non-Kernel device) create_process
generic_process_managers	illegal_process_address	create_process
generic_process_managers	illegal_process_identifier	create_process
generic_process_managers	insufficient_space	declare_process (Kernel process) declare_process (non-Kernel device) create_process
generic_process_managers	no_Kernel_process_on_non_Kernel_device	create_process
generic_process_managers	process_already_created	create_process
generic_process_managers	process_already_exists	declare_process (Kernel process) declare_process (non-Kernel device)
generic_process_managers	unknown_non_Kernel_device	declare_process (non-Kernel device)
generic_processor_management	calling_unit_not_Main_Unit	initialize_Master_processor initialize_subordinate_processor initialization_complete
generic_processor_management	network_failure	initialize_Master_processor initialize_subordinate_processor initialization_complete
generic_processor_management	Master_initialization_timeout_expired	initialize_Master_processor

Index of Kernel Exceptions by Package

Package Name	Exception	Raising Subprogram
generic_processor_management	configuration_tables_inconsistent	initialize_Master_processor
generic_processor_management	final_sync_initialization_timeout_expired	initialization_complete
generic_processor_management	process_initialization_failure	initialization_complete
generic_processor_management	process_maximum_exceeded	initialization_complete
generic_processor_management	processor_failed_to_ACK_go_message	initialize_Master_processor
generic_processor_management	processor_failed_to_transmit_NCT	initialize_Master_processor
generic_processor_management	subordinate_initialization_timeout_expired	initialize_subordinate_processor
generic_schedule_types	null_priority_range	raised during elaboration if type *priority* has a null range
generic_semaphore_management	illegal_context_for_call	claim claim (waiting for) claim (waiting until) release
generic_semaphore_management	claim_timed_out	claim (waiting for) claim (waiting until)
generic_semaphore_management	not_my_semaphore	release
generic_time_management	network_failure	synchronize
generic_time_management	OK_but_time_already_passed	adjust_epoch_time
generic_time_management	change_results_in_negative_elapsed_time	adjust_elapsed_time
generic_time_management	change_results_in_negative_epoch_time	adjust_epoch_time
generic_ time_management	synchronization_in_progress	synchronize
generic_time_management	synchronization_timeout_expired	synchronize
generic_timeslice_management	illegal_quantum	set_timeslice

Index of Kernel Names by Raising Subprogram		
Raising Subprogram	Exception Name	Package Name
adjust_elapsed_time	change_results_in_negative_elapsed_time	generic_time_management
adjust_epoch_time	OK_but_time_already_passed	generic_time_management
adjust_epoch_time	change_results_in_negative_epoch_time	generic_time_management
allocate_device_receiver	no_such_device_exists	generic_communication_management
allocate_device_receiver	replacing_previous_allocation	generic_communication_management
bind_interrupt_handler	illegal_interrupt	generic_interrupt_management
bind_interrupt_handler	illegal_interrupt_handler_address	generic_interrupt_management
bind_interrupt_handler	replacing_previous_interrupt_handler	generic_interrupt_management
bind_interrupt_handler	reserved_interrupt	generic_interrupt_management
cancel_alarm	illegal_context_for_call	generic_alarm_management
cancel_alarm	no_alarm_set	generic_alarm_management
claim	illegal_context_for_call	generic_semaphore_management
claim (waiting for)	illegal_context_for_call	generic_semaphore_management
claim (waiting for)	claim_timed_out	generic_semaphore_management
claim (waiting until)	illegal_context_for_call	generic_semaphore_management
claim (waiting until)	claim_timed_out	generic_semaphore_management
create_process	calling_unit_not_Main_Unit	generic_process_managers
create_process	illegal_process_address	generic_process_managers
create_process	illegal_process_identifier	generic_process_managers
create_process	insufficient_space	generic_process_managers
create_process	no_Kernel_process_on_non_Kernel_device	generic_process_managers
create_process	process_already_created	generic_process_managers

Index of Kernel Names by Raising Subprogram		
Raising Subprogram	Exception Name	Package Name
declare_process (Kernel process)	calling_unit_not_Main_Unit	generic_process_managers
declare_process (Kernel process)	insufficient_space	generic_process_managers
declare_process (Kernel process)	process_already_exists	generic_process_managers
declare_process (non-Kernel device)	calling_unit_not_Main_Unit	generic_process_managers
declare_process (non-Kernel device)	insufficient_space	generic_process_managers
declare_process (non-Kernel device)	process_already_exists	generic_process_managers
declare_process (non-Kernel device)	unknown_non_Kernel_device	generic_process_managers
die	illegal_context_for_call	generic_process_attribute_modifiers
disable	illegal_interrupt	generic_interrupt_management
disable	reserved_interrupt	generic_interrupt_management
enable	illegal_interrupt	generic_interrupt_management
enable	no_interrupt_handler_bound	generic_interrupt_management
enable	reserved_interrupt	generic_interrupt_management
enabled	illegal_interrupt	generic_interrupt_management
enabled	reserved_interrupt	generic_interrupt_management
get_process_preemption	illegal_context_for_call	generic_process_attribute_readers
get_process_priority	illegal_context_for_call	generic_process_attribute_readers
initialization_complete	calling_unit_not_Main_Unit	generic_processor_management
initialization_complete	network_failure	generic_processor_management
initialization_complete	final_sync_initialization_timeout_expired	generic_processor_management
initialization_complete	process_initialization_failure	generic_processor_management
initialization_complete	process_maximum_exceeded	generic_processor_management
initialize_Master_processor	calling_unit_not_Main_Unit	generic_processor_management

Index of Kernel Names by Raising Subprogram

Raising Subprogram	Exception Name	Package Name
initialize_Master_processor	network_failure	generic_processor_management
initialize_Master_processor	Master_initialization_timeout_expired	generic_processor_management
initialize_Master_processor	configuration_tables_inconsistent	generic_processor_management
initialize_Master_processor	processor_failed_to_ACK_go_message	generic_processor_management
initialize_Master_processor	processor_failed_to_transmit_NCT	generic_processor_management
initialize_subordinate_processor	calling_unit_not_Main_Unit	generic_processor_management
initialize_subordinate_processor	network_failure	generic_processor_management
initialize_subordinate_processor	subordinate_initialization_timeout_expired	generic_processor_management
receive_message	illegal_context_for_call	generic_communication_management
receive_message	buffer_too_small_for_message	generic_communication_management
receive_message (waiting for)	illegal_context_for_call	generic_communication_management
receive_message (waiting for)	buffer_too_small_for_message	generic_communication_management
receive_message (waiting for)	message_timed_out	generic_communication_management
receive_message (waiting for)	no_message_available	generic_communication_management
receive_message (waiting until)	illegal_context_for_call	generic_communication_management
receive_message (waiting until)	buffer_too_small_for_message	generic_communication_management
receive_message (waiting until)	message_timed_out	generic_communication_management
receive_message (waiting until)	no_message_available	generic_communication_management
release	illegal_context_for_call	generic_semaphore_management
release	not_my_semaphore	generic_semaphore_management
send_message	receiver_dead	generic_communication_management
send_message	receiver_never_existed	generic_communication_management
send_message_and_wait	illegal_context_for_call	generic_communication_management

Index of Kernel Names by Raising Subprogram		
Raising Subprogram	Exception Name	Package Name
send_message_and_wait	message_not_received	generic_communication_management
send_message_and_wait	network_failure	generic_communication_management
send_message_and_wait	receiver_dead	generic_communication_management
send_message_and_wait	receiver_is_sender	generic_communication_management
send_message_and_wait	receiver_never_existed	generic_communication_management
send_message_and_wait (for)	illegal_context_for_call	generic_communication_management
send_message_and_wait (for)	message_not_received	generic_communication_management
send_message_and_wait (for)	message_timed_out	generic_communication_management
send_message_and_wait (for)	network_failure	generic_communication_management
send_message_and_wait (for)	receiver_dead	generic_communication_management
send_message_and_wait (for)	receiver_is_sender	generic_communication_management
send_message_and_wait (for)	receiver_never_existed	generic_communication_management
send_message_and_wait (until)	illegal_context_for_call	generic_communication_management
send_message_and_wait (until)	message_not_received	generic_communication_management
send_message_and_wait (until)	message_timed_out	generic_communication_management
send_message_and_wait (until)	network_failure	generic_communication_management
send_message_and_wait (until)	receiver_dead	generic_communication_management
send_message_and_wait (until)	receiver_is_sender	generic_communication_management
send_message_and_wait (until)	receiver_never_existed	generic_communication_management
set_alarm (after)	illegal_context_for_call	generic_alarm_management
set_alarm (after)	resetting_existing_alarm	generic_alarm_management
set_alarm (for)	illegal_context_for_call	generic_alarm_management
set_alarm (for)	resetting_existing_alarm	generic_alarm_management

Index of Kernel Names by Raising Subprogram		
Raising Subprogram	Exception Name	Package Name
set_process_preemption	illegal_context_for_call	generic_process_attribute_modifiers
set_process_priority	illegal_context_for_call	generic_process_attribute_modifiers
set_timeslice	illegal_quantum	generic_timeslice_management
simulate_interrupt	illegal_interrupt	generic_interrupt_management
simulate_interrupt	no_interrupt_handler_bound	generic_interrupt_management
simulate_interrupt	reserved_interrupt	generic_interrupt_management
synchronize	network_failure	generic_time_management
synchronize	synchronization_in_progress	generic_time_management
synchronize	synchronization_timeout_expired	generic_time_management
wait (for)	illegal_context_for_call	generic_process_attribute_modifiers
wait (until)	illegal_context_for_call	generic_process_attribute_modifiers
who_am_I	illegal_context_for_call	generic_process_attribute_readers
raised by the Kernel when the alarm expires asynchronously	alarm_expired	generic_alarm_management
raised during elaboration if type *bus_address* is not within range	bus_address_check	generic_network_globals
raised during elaboration if type *priority* has a null range	null_priority_range	generic_schedule_types

Appendix C: Tailoring and Preparing the Kernel

This appendix presents a detailed discussion of the generic formal parameters used for tailoring the Kernel, what their settings mean, and how they interact with each other, with the hardware, and with the application. This appendix also enumerates all hard capacity and size limitations based on the declaration of types and values within the Kernel software.

C.1. Tailoring the Network

The following tailoring parameters require network-wide consistency and must be tailored to reflect the hardware network configuration, the real-time clock, communication limitations, and storage space considerations.

C.1.1. Tailoring the Hardware Network Configuration

Tailoring the hardware network configuration includes identifying the number of nodes, both Kernel processes and non-Kernel devices, on the network via the parameter *number_of_nodes_value*. This value then limits the number of entries in the NCT.

Tailoring the hardware network configuration also includes defining the following bus information to the Kernel. There are three values constituting this description:

1. The address with the lowest value recognized by the Kernel is given by *first_bus_address_value*.
2. The address with the highest value recognized by the Kernel is given by *last_bus_address_value*.
3. A null bus address is provided via the *null_address_value*.

These three values define the bounds of the *bus_address* type exported by *generic_network_globals*. Values of type *bus_address* are represented in the NCT in the *physical_address* field. Only values within the range *first_bus_address_value* .. *last_bus_address_value* are recognized as legal addresses by the communication management function of the Kernel.

C.1.2. Tailoring to the Real-Time Clock

The Kernel representation of time is described in Section 5.2.5. This representation is in terms of an integral number of microseconds, that is, a bit value of 16#00000001# represents one microsecond.

This assumes that the underlying real-time clock also measures time in units commensurable with one microsecond (i.e., that the clock counts time in units such as 1µsec, 2µsec, 10µsec, 0.25µsec, etc). If the clock counts in units that are not a multiple or fraction of a microsecond, such as 1/65536 second, then the Kernel representation will be inaccurate; there will be *jitter* in the representation of time.

In the case where a clock ticks at intervals of 1/65536 second, one tick is fractionally more than

15μsec, and 1024 ticks are exactly 15625 μsec. In terms of Kernel time, successive ticks will appear to be sometimes 15μsec apart and sometimes 16μsec apart, which is a jitter of 1 part in 15, or about 7%. If this is unacceptable, the Kernel representation of time must be changed.

It is assumed that the hardware timer can count a finite number of ticks before it overflows, and that the overflow causes an interrupt that resets the timer and continues accumulating ticks in software. For example, if the timer is 16 bits wide, then after every 65536 ticks, the timer causes an interrupt, and the handler must record that a further 65536 ticks have elapsed, probably by adding 16#00010000# to a data object somewhere.

Given that the underlying clock is suitable, the size of the tick then depends on two factors:

1. How fine a resolution is required, and
2. How much overhead is involved in resetting the timer.

In the above example, a tick of 1μsec would imply an interrupt every 65msec, which is probably satisfactory. If the handler takes 200μsec to execute (not an unreasonable figure), then about 0.4% of the CPU time is devoted to servicing the timer interrupt.

A tick larger than 1μsec is probably reasonable for many applications. Consider, for instance, that it may take 5 or 10 μsec simply to read the current value of the timer, and a further 20 μsec or more to convert that value into the Kernel representation and adjoin the high-order bits that the software is maintaining. If it takes 25μsec to read the current time, a tick of 8μsec or even more is not unreasonable.

The mechanics of tailoring to the real-time clock require three things:

1. The hardware clock driver and handler and the internal function *get_clock* should be adapted for the actual timer in use.
2. The customization parameter *ticks_per_second_value* in the internal package *generic_Kernel_time* should be set to the correct value, and the generic instantiations performed.
3. The application should use the value *ticks_per_second*, exported by the internal package *Kernel_time*, to determine the current granularity of representation of time. Time, as seen by the application, advances in ticks.

This does not change the representation of time values in the Kernel, which remains as described in Section 5.2.5. Rather, it determines the accuracy of any time value read from the clock. Thus, if the granularity is 8μsec, for a time value represented as integral microseconds, the bottom three bits will always be zero. The time perceived by the application advances instantaneously from 16#00000000# to 16#00000008# to 16#00000010# and so forth. In effect, the clock is making tiny jumps as it ticks.

C.1.3. Tailoring Communication Limitations

The application has the capability to limit the size of a single message that may be placed on the network. This is accomplished via *maximum_message_length_value*. By tailoring this with the application requirements in mind, the Kernel can do a more optimal job of message handling.

C.1.4. Tailoring Data Structure Storage

The application may limit the amount of string space consumed by the NCT by limiting the length of the processor name that is maintained in that data structure. This is done via *maximum_length_of_processor_name_value*.

C.1.5. Summary of Network-Wide Tailoring Parameters

Each of the tailoring parameters below identifies the exporting package in parentheses, describes the legal set of values for the parameter, indicates the DARK-provided default value (if any), and notes the value assigned to that parameter for execution on the DARK testbed at the SEI.

Network-Wide Tailoring Parameters			
Parameter Name	Package Name	Range	Default
first_bus_address_value	*generic_network_globals*	0 .. +32_767	0
last_bus_address_value	*generic_network_globals*	0 .. +32_767	255
maximum_length_of_processor_name_value	*generic_network_configuration*	0 .. +32_767	none
maximum_message_length_value	*generic_communication_globals*	0 .. +32_767	none
null_address_value	*generic_network_globals*	0 .. +32_767	none
number_of_nodes_value	*generic_network_configuration*	1 .. +32_767	none
ticks_per_second_value	*generic_Kernel_time*	1 .. +32_767	none

Currently, the only one of these values that is checked for network-wide consistency by the Kernel software is *number_of_nodes_value*, which is automatically checked as part of the network initialization protocol. It is up to the application engineer to ensure that all other tailoring parameters that require network-wide consistency are, in fact, consistent across the entire network.

All of these tailoring parameters are used to initialize visible constants of the same name less the "*_value*" suffix.

C.2. Tailoring Each Processor

The following tailoring parameters do not require network-wide consistency. They determine the characterists of the specific processor on which this version of the Kernel is to execute. These parameters are used to describe the process environment, limit the range of process priorities, define time constants available to the application, limit the number of interrupt names available to the application and the Kernel, and define storage space considerations.

C.2.1. Tailoring the Process Environment

The process environment is created by the Kernel for every process for which the Kernel primitive *create_process* is invoked. The application may provide the following maxima:

1. *Maximum_message_queue_size_value*, used as the default size for all process incoming message queues; and
2. *Maximum_process_stack_size_value*, used as the default size for all process stack storage.

Once these values are set, the Kernel uses these values as processor defaults, and performs error checking against these values with actual values provided by calling the Kernel primitive *create_process.*

C.2.2. Tailoring the Range of Process Priorities

The application may limit the range of priorities available within a single processor by specifying the lowest priority value (such a process would be among the very last to be selected for execution). This is accomplished via *lowest_priority_value*. This value determines the upper bound on type *priority*, which is available to the application program.

The highest priority recognized by the Kernel is hardwired as one (*priority'first* + 1). A priority of value zero indicates no change in priority (*priority'first* = 0).

C.2.3. Tailoring Time Constants

A slice is the smallest schedulable interval of time. The application uses this value when describing the duration of a timeslice quantum (in number of slices). All time values that imply scheduling action, such as delay times, are truncated to the nearest slice.

The *minimum_slice_time_value* sets the minimum amount of time that may be specified as a timeslice quantum. This value must take into account the minimum context switch time required by the Kernel, as well as the needs of the application and the requirements it places on the processing power of the hardware.

The *ticks_per_slice_value* establishes the smallest amount of elapsed time that may constitute a slice - a schedulable interval of time. To determine a reasonable setting for this value, the following must be taken into consideration:

To obtain a suitable value for the length of a slice, the following should be considered:

1. The smallest interval of time that the application needs to consider. For example, if a critical computation has to be performed in less than 200 µsec, the application will need to set an alarm for a time less than 200µsec in the future. The slice must therefore be no larger than this.
2. The greatest common divisor (GCD) of all application cycles. For example, if an application contains three cyclic processes of periods 1ms, 2.4ms and 4.6ms, the GCD is 200µsec. A slice of this value allows all three processes to run without having to round off any resumption time value.
3. The time necessary to set a timer, enable its interrupt, field the interrupt, and

suspend and resume a process. This is the cost of performing a time-based preemptive scheduling action. The slice must necessarily be larger than this.

Items (a) and (b) above will be derived from the intended application or set of applications. Item (c) will be derived from the performance statistics distributed with each version of the Kernel. If the times in (c) are substantially smaller than those derived from (a) and (b), then it is probably feasible to use the Kernel to support the applications, and a suitable slice value can probably be found. *If, however, the times in (c) are larger than those required by the application, then the Kernel performance is insufficient to support the application.*

C.2.4. Tailoring Interrupt Name Usage

To maintain Kernel data structures, two parameters are provided to limit the number of interrupt names used by the Kernel and by the application.

Number_of_interrupt_names_used_by_application defines the stated maximum. This value is used to compute the size of the interrupt data structures used by the Kernel. See Section 5.2.4 for more information about these data structures.

Number_of_interrupt_names_used_by_Kernel should **not** be adjusted once the Kernel is delivered. This value is provided as a tailoring parameter for Kernel developers.

C.2.5. Tailoring Data Structure Storage

The application may limit the amount of string space consumed by the Process Table by limiting the length of the process name that is maintained in that data structure. This is done via *maximum_length_of_process_name_value.*

In addition, the application may limit the final size of the Process Table and the Process Index Table. This is done via *maximum_number_of_processes_value.*

C.2.6. Summary of Processor-Specific Tailoring Parameters

Each of the tailoring parameters in Table C-1 identifies the exporting package in parentheses, describes the legal set of values for the parameter, indicates the DARK-provided default value (if any), and notes the value assigned to that parameter for execution on the DARK testbed at the SEI.

The tailoring parameters *lowest_priority_value* and *maximum_length_of_process_name_value* are used to initialize visible constants of the same name less the "*_value*" suffix.

C.3. Kernel Limitations

This section enumerates all hard limits imposed by the use of primitive data types in Kernel software. The absolute limit is as indicated in the list; the practical limit may be substantially less. These limits are presented in Table C-2.

See also limitations defined by package *hardware_interface* in Section 4.1.

Table C-1: Processor-Specific Tailoring Parameters

Processor-Specific Tailoring Parameters			
Parameter Name	Package Name	Range	Default
lowest_priority_value	*generic_schedule_types*	1 .. +32_767	none
maximum_length_of_process_name_value	*generic_process_managers_globals*	0 .. +32_767	none
maximum_message_queue_size_value	*generic_process_managers*	0 .. +2_147_483_647	none
maximum_process_stack_size_value	*generic_process_managers*	1 .. +2_147_483_647	none
maximum_number_of_processes_value	*generic_process_table*	1 .. +32_767	none
minimum_slice_time_value	*generic_timeslice_management*	77 µsec	none
*number_of_interrupt_names_used_by_application**	*generic_interrupt_globals*	1 .. +32_767	none
*number_of_interrupt_names_used_by_Kernel**	*generic_interrupt_globals*	1 .. +32_767	none

Table C-2: Kernel Limitations

Kernel Limitations		
Package Name	Parameter Name	Range
generic_communication_globals	*maximum_message_length_value*	0 .. +32_767 (tailorable)
generic_communication_globals	*message_length_type*	0 .. +32_767
generic_communication_globals	*message_tag_type*	32_768 .. +32_767
generic_interrupt_globals	*interrupt_name*	0 .. 255
generic_interrupt_globals	*interrupt_table_type*	indexed by *interrupt_name*; only 256 entries
generic_interrupt_globals	*number_of_interrupt_names_used_by_application*	1 .. +32_767 (tailorable)
generic_interrupt_globals	*number_of_interrupt_names_used_by_Kernel*	1 .. +32_767 (tailorable)
generic_Kernel_time	*integral_duration*	2_147_483_648 .. +2_147_483_647
generic_Kernel_time	*Kernel_time*	beginning at time zero .. +150_000 years
generic_Kernel_time	*ticks_per_second_value*	1 .. +32_767 (tailorable)
generic_network_configuration	*maximum_length_of_processor_name_value*	0 .. +32_767 (tailorable)
generic_network_configuration	*NCT*	index range is 1 .. *number_of_nodes*, which is set from *number_of_nodes_value*; only *number_of_nodes_value* number of entries
generic_network_configuration	*NCT_entry.initialization_order*	0 .. +32_767 (should have same upper bound as *number_of_nodes_value*)
generic_network_configuration	*number_of_nodes_value*	1 .. +32_767 (tailorable)
generic_network_globals	*bus_address*	0 .. +32_767 (actual bounds of this type determined by tailoring parameters *first_bus_address_value* and *last_bus_address_value*)
generic_network_globals	*first_bus_address_value*	0 .. +32_767 (tailorable)
generic_network_globals	*last_bus_address_value*	0 .. +32_767 (tailorable)
generic_network_globals	*process_index.node_number*	values of type *bus_address*

Kernel Limitations		
Package Name	Parameter Name	Range
generic_network_globals	*process_index.process_number*	-32_768 .. +32_767
generic_network_globals	*processor_identifier*	0 .. +32_767 (type of index into NCT; must be compatible with *number_of_nodes_value*)
generic_process_managers	*maximum_message_queue_size_value*	0 .. +2_147_483_647 (tailorable)
generic_process_managers	*maximum_process_stack_size_value*	1 .. +2_147_483_647 (tailorable)
generic_process_managers_globals	*device_name_type*	string indexed 1 .. *maximum_length_of_process_name_value*
generic_process_managers_globals	*maximum_length_of_process_name_value*	0 .. +32_767 (tailorable)
generic_process_managers_globals	*process_name_type*	string indexed 1 .. *maximum_length_of_process_name_value*
generic_process_table	*maximum_number_of_processes_value*	1 .. +32_767 (tailorable)
generic_process_table	*process_information_record. communication_attributes. maximum_message_queue_size*	0 .. +32_767
generic_process_table	*semaphore.number_of_waiting_processes*	0 .. +32_767
generic_schedule_types	*highest_priority*	*priority'first + 1*
generic_schedule_types	*lowest_priority*	*lowest_priority_value*
generic_schedule_types	*lowest_priority_value*	1 .. +32_767 (tailorable)
generic_schedule_types	*priority*	0 .. *lowest_priority_value* + 1(*priority'first* and *priority'last* are reserved for use by the Kernel only and should never be used by the application)

C.4. Tailoring Error Checking and Reporting

As described in Section 2.12, the Kernel provides the capability of selectively enabling and disabling error checking, processing, and reporting on a per-processor basis. Table C-3 enumerates each tailoring parameter that corresponds to an exception that falls into this category. In all cases, the exception name is identical to the tailoring parameter name less the "*_enabled*" suffix.

Each of the tailoring parameters below identifies the exporting package in parentheses. The DARK-provided default value of each error checking enabling parameter is true; the value assigned to that parameter for execution on the DARK testbed at the SEI is also true.

More information can be found in Appendix B and the "Error Conditions" and "Notes" portions of the Kernel Specification in Appendix A.

Table C-3: Error Checking Tailoring Parameters

Tailoring Error Checking and Reporting	
Package Name	Exception Name
generic_alarm_management	illegal_context_for_call_enabled no_alarm_set_enabled resetting_existing_alarm_enabled
generic_communication_management	buffer_too_small_for_message_enabled illegal_context_for_call_enabled message_not_received_enabled message_timed_out_enabled network_failure_enabled no_message_available_enabled no_such_device_exists_enabled receiver_dead_enabled receiver_is_sender_enabled receiver_never_existed_enabled replacing_previous_allocation_enabled
generic_interrupt_management	illegal_interrupt_enabled illegal_interrupt_handler_address_enabled no_interrupt_handler_bound replacing_previous_interrupt_handler_enabled reserved_interrupt_enabled
generic_network_globals	bus_address_check_enabled
generic_process_attribute_modifiers	illegal_context_for_call_enabled
generic_process_attribute_readers	illegal_context_for_call_enabled
generic_process_managers	calling_unit_not_main_unit_enabled illegal_process_address_enabled illegal_process_identifier_enabled insufficient_space_enabled no_kernel_process_on_non_kernel_device_enabled process_already_created_enabled process_already_exists_enabled unknown_non_kernel_device_enabled
generic_processor_management	calling_unit_not_main_unit_enabled
generic_schedule_types	null_priority_range_enabled
generic_semaphore_management	claim_timed_out_enabled illegal_context_for_call_enabled not_my_semaphore_enabled
generic_time_management	change_results_in_negative_elapsed_time_enabled change_results_in_negative_epoch_time_enabled network_failure_enabled OK_but_time_already_passed_enabled synchronization_in_progress_enabled synchronization_timeout_enabled
generic_timeslice_management	illegal_quantum_enabled

Appendix D: Scheduling Algorithms

This appendix presents the Kernel's scheduling algorithms.

The following Scheduler rules are universally applied:

1. Scheduler order does not change spontaneously.
2. Scheduler ordering is decided by:

   a. Higher priority before lower priority

   b. Prefer a process in an error state (to one in a normal state)

   c. First-in, first-out (FIFO) order otherwise

   In other words, in all Scheduler situations, where priorities are equal, a process in an error state is resumed preferentially; otherwise, the process first to become unblocked is resumed.

3. When two processes become unblocked simultaneously, the process that has been blocked longest is considered to become unblocked first.[4]

Scheduler Algorithm

```
Begin critical section
If the set of suspended processes is not empty
  Choose the process to resume according to the Scheduler rules above
  If timeslicing enabled =>
    Schedule slice event
  End if
  If chosen process = CURRENT RUNNING PROCESS =>
    Resume Process via the Context Switcher
  Else
    Switch Processes via the Context Switcher
  End if
Else
  Resume Process ("idle process")
End if
End critical section
```

[4] Two processes executing on the same processor cannot become `blocked` simultaneously.

Schedule Slice Event

```
If slice event ID /= no event =>
  If chosen process = CURRENT RUNNING PROCESS =>
    Null
  Else
    Remove Event (slice expiration)
    If chosen process is preemptable =>
      Set SLICE EVENT ID to Insert Event (slice expiration)
    End if
  End if
Else
  If chosen process is preemptable =>
    Set SLICE EVENT ID to Insert Event (slice expiration)
  End if
End if
```

Appendix E: Building Abstractions

This appendix provides example abstractions that can be built using the Kernel primitives. These examples include: building typed message passing, safe critical regions, cyclic and periodically scheduled processes, time-critical transactions, monitors, mutually self-scheduling processes, and a message router. The examples provided in this appendix can be used as a template for application builders who need to construct application-specific code that can be based on the paradigms herein. The examples include program design language (PDL) for one example solution.

E.1. Typed Message Passing

The Kernel communication primitives (see Section 4.7) transmit and receive messages that are untyped. Each of these primitives considers a message to consist of a length (in storage units) and an address designating the first storage unit occupied by the message.

An application written in Ada may require more security than this, exploiting Ada's compile-time type checking and using typed messages. This can be achieved by using a package such as the following:

```
generic

   type message_type is private;

package typed_communication_management is

   procedure send_message
   (
      receiver    : in process_identifier;
      message_tag : in message_tag_type;
      message     : in Message_Type
   );

   procedure receive_message
   (
      sender              : out process_identifier;
      message_tag         : out message_tag_type;
      message_buffer      : out Message_Type
      resumption_priority : in  priority {:=...};
      messages_lost       : out Boolean;
   );

end typed_communication_management;
```

This package exports *send_message* and *receive_message* primitives that expect typed values and objects. To use it, the application code instantiates the package for each actual message type. If two processes communicate via typed messages, the code of each imports the instantiated specification.

However, this is still not completely safe. Although good configuration management tools should prevent it from occurring, it is still possible for the sender to import one instantiation, and send a value of one type, and the receiver to import another instantiation, and so get a message that it thinks is of a different type. A further check, performed at execution time, could use the message tag as a validity check:

```
generic

   type message_type is private;
   tag_check_value : in message_tag_type;

package typed_communication_management is

   invalid_message_type : exception;

   procedure send_message
   (
      receiver    : in process_identifier;
      message     : in Message_Type
   );

   procedure receive_message
   (
      sender              : out process_identifier;
      message_buffer      : out message_type
      resumption_priority : in  priority {:=...};
      messages_lost       : out Boolean;
   );

end typed_communication_management;
```

The body of this package would look like this:

```
with communication_management;
with system;
package body typed_communication_management is

   procedure send_message
   (
      receiver    : in process_identifier;
      message     : in message_type
   ) is

   begin

      -- call the Kernel send_message primitive, passing the agreed
      -- message tag, the message length, the message address

      communication_management.send_message
      (
         receiver       => receiver,
         message_tag    => tag_check_value,
         message_length => message_type'size / system.storage_unit'size,
         message_text   => hw_address (message'address)
```

```
    );
  end send_message;

  procedure receive_message
  (
    sender              : out process_identifier;
    message_buffer      : out message_type
    resumption_priority : in  priority {:=...};
    messages_lost       : out Boolean;
  ) is

    rcvd_tag_value      : message_tag_type;
    rcvd message_length : message_length_type;
    buffer_size         : constant message_length_type :=
      message_type'size / system.storage_unit'size;

  begin

    -- call the Kernel receive_message primitive, telling it the
    -- buffer length and address, and receiving from it the actual
    -- received  message tag and length

    communication_management.receive_message
    (
      sender              => sender,
      message_tag         => rcvd_tag_value,
      message_length      => rcvd_message_length,
      message_buffer      => hw_address (message_buffer'address),
      buffer_size         => buffer_size,
      resumption_priority => resumption_priority,
      messages_lost       => messages_lost
    );

    -- if the message is of the correct type, the length and tag must
    -- be correct; if this is not so, a message of the wrong type
    -- was received and the exception must be raised

    if rcvd_message_length /= buffer_size
      or else rcvd_tag_value /= tag_check_value then
      raise invalid_message_type;
    end if;

  end receive_message;

end typed_communication_management;
```

Each instantiation must use a different *tag_check_value*.

Once set up, these packages can be used by application code with a high degree of reliability. However, a corresponding price must be paid in terms of code and execution overhead.

E.2. Safe Critical Regions

To build a safe critical region or protected data structure, mutually exclusive access to the region must be guaranteed. There are two possible sources of concurrent access against which to protect:

1. Processes, and
2. Interrupt handlers.

In addition, there are levels of exclusiveness in critical regions:

1. Providing mutually exclusive access to some object.
2. Providing mutually exclusive and uninterrupted access (by other processes) to some object.
3. Providing mutually exclusive and totally uninterrupted access (by other processes or interrupts) to some object.

Level 1 is easily achieved by convention within the application program, where all critical regions are guarded by a semaphore, and a process must have possession of that semaphore before accessing the critical region. Since the Kernel does not allow interrupt handlers to maintain state, an interrupt handler may not perform a blocking operation or claim a semaphore. If an interrupt handler needs to affect a protected object, it must do so via a process acting as its agent (i.e., by sending the process a message). The code to accomplish Level 1 exclusion might look like:

```
with semaphore_management;
package shared_data is

   lock : semaphore_management.semaphore;
   object : some_type;

end shared_data;

with semaphore_management;
with shared_data;
procedure sample_level_1_critical_region is
begin
      . . .

   semaphore_management.claim (semaphore_name => shared_data.lock);
   update (shared_data.object);
   semaphore_management.release (semaphore_name => shared_data.lock);

      . . .
end sample_level_1_critical_region;
```

Level 2 exclusiveness is also relatively easy to obtain by convention. It requires the semaphore convention of Level 1, to block out other processes that require the resource. It also requires that the process with the critical region must have the highest priority (a special priority level reserved by the application designer exclusively for this use) of any process in the system, to prevent other processes that don't require the resource from executing. The code to implement Level 2 exclusion might look like:

```
with schedule_types;
with semaphore_management;
package shared_data is

   lockout_priority : schedule_types.priority := 1;
   lock : semaphore_management.semaphore;
   object : some_type;

end shared_data;

with process_attribute_modifiers;
with process_attribute_readers;
with semaphore_management;
with shared_data;
procedure sample_level_2_critical_region is
begin
      . . .

   old_priority := process_attribute_readers.get_process_priority;
   semaphore_management.claim (
      semaphore_name => shared_data.lock,
      resumption_priority => shared_data.lockout_priority);
   update (shared_data.object);
   semaphore_management.release (semaphore_name => shared_data.lock);
   process_attribute_modifiers.set_process_priority
      (new_priority => old_priority);

      . . .
end sample_level_2_critical_region;
```

Level 3 is more difficult to obtain. At the process level, it requires the same conventions as Level 2 exclusion. The difficulty is locking out interrupts. Clearly, non-maskable interrupts may never be locked out (but since they represent truly disastrous system failures, this is not a problem). The only way to achieve Level 3 exclusion (minus non-maskable interrupts) is to effectively disable every interrupt, which may be done in two ways: by individually disabling every active device or by using an available hardware feature to mask interrupts. The example shown here uses an internal Kernel package that illustrates how to mask out interrupts on a 68020 using the Telesoft V3.22a cross-compiler:

```
with low_level_hardware;
with process_attribute_modifiers;
with process_attribute_readers;
with schedule_types;
with semaphore_management;
package shared_data is

   lockout_priority : schedule_types.priority := 1;
   lock : semaphore_management.semaphore;
   object : some_type;

end shared_data;

with semaphore_management;
with shared_data;
procedure sample_level_3_critical_region is
begin
      . . .

   old_priority := process_attribute_readers.get_process_priority;
   semaphore_management.claim (
      semaphore_name => shared_data.lock,
      resumption_priority => shared_data.lockout_priority);
   low_level_hardware.begin_atomic;
   update (shared_data.object);
   low_level_hardware.end_atomic;
   semaphore_management.release (semaphore_name => shared_data.lock);
   process_attribute_modifiers.set_process_priority
      (new_priority => old_priority);

      . . .
end sample_level_3_critical_region;
```

There are a number of risks associated with Level 3 exclusion. First, one runs the risk of missing important interrupts, which means the amount of time spent in a Level 3 critical section must always be minimized. Second, if the Kernel clock interrupt is disabled, the clock will drift and inaccuracies will creep into the Scheduler.

In critical regions, care must be taken to back out properly in the event of an exception. All critical regions should contain an exception handler of the form:

```
exception
   when others =>
      . . .
      low_level_hardware.end_atomic;   -- to back out of level 3
      semaphore_management.release
         (semaphore_name => shared_data.lock);
```

E.3. Cyclic Processes

A cyclic process is one that is scheduled to execute every *n* units of time and must complete it's execution within that time period. The solution presented here is a general one that allows the process to be implemented without knowing the cycle time and for that time to be varied as needed when the process begins execution. The code template to achieve this is:

```
with alarm_management;
with communication_management;
with process_attribute_modifiers;
with time_management;
procedure cyclic_process is

   --
   --  (1)  read the cycle_time from the message sent by the process
   --       controlling the system.  if this generality is not needed, then
   --       the cycle_time can be coded directly into the process or placed
--       in a global datum.
--
--  (2)  compute the next absolute time the process is to run
--
--  (3)  setup a frame overrun timeout, just a little shorter than the
--       actual cycle time, to allow time for cleanup and to get
--       back to the start of the loop for the next cycle (this
--       cleanup and recycle time is the value of delta)
--
--  (4)  if the processing is completed on time, then the timeout is
--       cancelled.  there is a potential race condition here in
--       that the timeout could expire just as the process
--       finishes its work. if this is a significant risk, then
--       the timeout handler must account for this possibility.
--
--  (5)  voluntarily deschedule the process until its next
--       scheduled wakeup time.
--
--  (6)  execution reaches this point if and only if the alarm has
--       expired, i.e., a frame overrun has occurred
--

begin

   -- (1) --
   communication_management.receive_message (..., cycle_time, ...);

   loop

      execution_frame:
      begin

         -- (2) --
         next_schedule_time := time_management.read_clock + cycle_time;

         -- (3) --
         alarm_management.set_alarm (cycle_time - delta);

         do_the_work;

         -- (4) --
         alarm_management.cancel_alarm;

         -- (5) --
         process_attribute_modifiers.wait (next_schedule_time);

      exception

         -- (6) --
         when alarm_management.alarm_expired =>
            -- back out whatever the process was doing but didn't finish

      end execution_frame;

   end loop;
   end cyclic_process;
```

This entire discussion is predicated on the assumption that the cyclic process gets enough cycles to execute the various steps. This is an application-level issue determined by the relative priorities of all the processes in the system.

E.4. Periodically Scheduled Processes

A periodically scheduled process is one that is scheduled to run every **n** slices after the last complete execution of itself. There are no constraints on how long the process should run – only on how much time should elapse between executions.

One example of such an application is a process that periodically updates a screen display. This process updates the screen every X seconds—it doesn't matter how long it takes to update the screen, or whether the process doing the screen update is preempted by a higher priority process. All that matters is that the information on the screen is periodically updated, and that (barring more important system functions) the screen information is no more than X seconds old.

The solution presented here is a general one that allows the process to be implemented without knowing the cycle time and for that time to be varied as needed. The code to achieve this is:

```
with hardware_interface;  use hardware_interface;
with communication_management;
with process_attribute_modifiers;
with time_globals;
procedure periodic_process is

   --
   -- read the interval_time from the message sent by the process
   -- controlling the system; if this generality is not needed,
   -- the cycle_time can be coded directly into the process or
   -- placed in a variable
   --
   -- do the work required
   --
   -- voluntarily block the process until its next scheduled wakeup
   -- time
   --

   procedure do_the_work;

   function to_interval_time (text : hw_string)
      return time_globals.elapsed_time;

begin

   communication_management.receive_message
   ( ...,
     interval_time_buffer, ...
   );
   loop
      do_the_work;
      process_attribute_modifiers.wait
      (
         for_elapsed_time => to_interval_time (interval_time_buffer)
      );
   end loop;

end cyclic_process;
```

The actual execution interval of the periodic process may vary, and can be longer (although never shorter) than the desired amount. This is a system-level issue determined by the relative priorities of all the processes in the system. If it is determined that the periodic process needs to be run with highly precise timing, a high priority should be assigned to the process.

E.5. Time-Critical Transactions

The Kernel alarm management facility (see Section 4.12) provides a means for a process to set a limit to the duration of any fragment of code.

For example, for a process to perform a computation *do_calculation*, but to abort it if it is not completed within 1 ms, the process can provide this guard:

```
guarded_fragment:
begin
   alarm_management.set_alarm (after_elapsed_time => milliseconds(1));
   do_calculation;
   alarm_management.cancel_alarm;
exception
   when alarm_management.alarm_expired =>
      null;
end guarded_fragment;
```

This ensures that the process may not consume more time than is allowed. However, it also causes the calculation to be abandoned when the alarm expires. This might be proper behavior in some cases, for instance in a real-time application where the result of the calculation is useless if not timely. But it is probably not adequate in other circumstances.

One example is a time-critical transaction that must either run to completion within a finite time or not be performed at all. If this transaction involves changing global state, such as a track table, then it is necessary for the process to be able to *back out* of the transaction when the alarm expires. In terms of a standard two-phase commit protocol, the skeleton looks like this:

```
guarded_fragment_with_backout:
begin
   POSIT;      -- prepare to perform transaction
   alarm_management.set_alarm (time_allowed_to_complete);
      perform_transaction;
      alarm_management.cancel_alarm;
   COMMIT;         -- transaction is now irreversible
exception
   when alarm_management.alarm_expired =>
      BACKOUT;        -- abort transaction and revert
end guarded_fragment_with_backout;
```

The protocol and the alarm management code must strictly nest in the manner shown. The postcondition of *perform_transaction* is that the transaction has succeeded; the postcondition of *cancel_alarm* is that the alarm has not expired (i.e., that the time taken is less than the time allowed). The joint postcondition is therefore that the transaction has succeeded within the time allowed, and so the transaction may be committed.

The above skeleton can be embedded in a generic procedure with a parameter *perform_transaction*:

```
with time_globals;
generic

   with procedure perform_transaction;

procedure time_critical_transaction
(
   time_allowed_to_complete : in time_globals.elapsed_time
);
```

This provides a safe encapsulation, allowing the global data administrator to build standard POSIT, COMMIT and BACKOUT protocols that all instantiations of the encapsulation may use in the correct manner.

E.6. Monitors

In many real-time applications, a resource needs to be shared by multiple processes. Usually, this resource sharing can be managed by using semaphores to lock and unlock access to the resource. The Kernel *semaphore_management* capability is described in Section 4.11. Sometimes, however, the mechanisms used to manipulate the resource must be uniformly and automatically enforced across an application or may need to be hidden, so that the processes using the resource do not know exactly how the resource is being used, or even that a shared resource is involved. In this case, a package to "monitor" the resource can be defined by the application.

E.6.1. Example Requirements and Justification

An application may need to update a group of related data when a single datum is changed. For example, the following physical quantities have the indicated relationships among them:

distance = speed * time
velocity = acceleration * time
bearing = angular velocity * time
height = rate of ascent * time

rate of ascent = speed * sin (glide angle)

In order for an application to obtain a consistent reading on these data, all must be modified as time progresses, as each is a function of time. On a uniprocessor system, this is relatively easy to accomplish; the *semaphore_management* capability may be used. However, on a system where processes may preempt each other, and where an access to global data may potentially be interrupted, a mechanism is needed to ensure that accesses are atomic (i.e., the data are modified in their entirety or not at all). For the purposes of the following simple example, the following assumptions are made:

1. Access to the data is a potentially blocking operation. A process needing access to any of the data (either for update or retrieval) requires fast access, but not immediate.
2. Access to the data is from a single processor. This simplifies the example by allowing the monitor abstraction to be built on the Kernel *semaphore_management* capability. *Semaphores* will be used to lock the database (*semaphores* are local to a processor).
3. The only operations on the data are: *update* and *retrieve*.

E.6.2. PDL of Example

The code for the procedures to monitor the data follows. External to the *data_monitor* package, the caller needs no knowledge as to how the monitor is implemented. All callers need to know is that the monitor ensures that modifications of the data are atomic and consistent access to the data is provided.

```
with time_globals;
package data_monitor is

   type position_data_record
   is record
      time : time_globals.elapsed_time;
      speed : {...}
      distance : {...}
      velocity : {...}
      height : {...}
      rate_of_ascent : {...}
   end record;

   procedure update
   (
      time : in time_globals.elapsed_time
   );

   procedure retrieve
   (
      data : in out position_data_record
   );

end data_monitor;

with hardware_interface;
with semaphore_management;
with time_globals;
package body data_monitor is

   type global_resources_information
   is record
      position_data : semaphore_management.semaphore;
      count : hw_integer;
   end record;

   global_resources : global_resources_information;

   position_information : position_data_record

   procedure update
   (
      time : in time_globals.elapsed_time
   ) is

   begin

      semaphore_management.claim
      (
         semaphore_name => global_resources.position_data
      );

      speed := read_speed_sensor;
      acceleration := read_accelerometer;
      glide_angle := read_attitude_sensor;
      rate_of_ascent :=
         compute_rate_of_ascent (speed, acceleration, glide_angle);
      new_height := compute_height (old_height, rate_of_ascent, time);

      old_copy_position_data := position_data;

      update_block:
      begin

         position_information.time := time;
         position_information.speed := speed;
         position_information.distance := speed * time;
         position_information.velocity_increment :=
            acceleration * time;
         position_information.height_increment :=
            rate_of_ascent * time;

      exception

         when others =>
            position_data := old_copy_position_data;

      end update_block;

      semaphore_management.release
      (
         semaphore_name => global_resources.position_data
      );

   end update;

   procedure retrieve
   (
      data : in out position_data_record
   ) is

   begin

      semaphore_management.claim
      (
         semaphore_name => global_resources.position_data
      );

      data := position_information;

      semaphore_management.release
      (
         semaphore_name => global_resources.position_data
      );

   end retrieve;

end data_monitor;
```

E.7. Mutually Self-Scheduling Processes

Kernel primitives may be used to specify points in the application program where processes voluntarily give up control of the processor in favor of other processes of the same or lower priority.

E.7.1. Example Requirements and Justification

One use of mutual self-scheduling is where calculations and control operations need to be performed in parallel on a single processor (that is that they need to be performed logically concurrently, but due to the distribution of the algorithm, true concurrency cannot be obtained). Thus, a time sharing system may be built, where the points at which the different parts of the algorithm are stopped and resumed are selected by the application designer, and not by the Kernel at the expiration of a timeslice. Another scenario in which a mutually self-scheduling paradigm is appropriate is one where algorithm processing is handled in a pipeline manner: a first process performs initial processing and suspends itself in favor of a second process, which continues processing and suspends itself in favor of a third process, and so forth. In this scenario, data would be transformed from an input state to an output state, where the output state of one process would correspond to an input state of the next process in the pipe-line.

One use of this type of scheduling is the sharing of data among processes, where partial update of data items is unacceptable to processing. Rather than use semaphores to control access to the data, scheduling is used to control access to the processor. When a process is able to give up control of the processor (i.e., it has finished writing data, reading data, or has executed a sequence of instructions that it deems a "fair" use of the processor), it explicitly calls the Kernel Scheduler via the Kernel primitive *wait*.

An example of mutually self-scheduling processes supporting a timeshare scenario is presented below.

E.7.2. PDL of Example

In this scenario, *process_1* does roughly one third of the time shared processing, as do *process_2* and *process_3*.

```
with process_attribute_modifiers;
procedure process_1 is

   --
   --  (1)  a wait of an elapsed time of 0 causes the Scheduler to
   --       choose the next process to run.  in the case of multiple
   --       processes with the same priority, the Scheduler chooses
   --       the process with the longest time of  not running (i.e.,
   --       one of process_2 or process_3).
   --

begin

   loop

      ... code that cannot be interrupted by process_2 or process_3

      -- (1) --
      process_attribute_modifiers.wait (
         for_elapsed_time => TG.elapsed_time(0));

      ... more code that cannot be interrupted by process_2 or process_3

      -- (1) --
      process_attribute_modifiers.wait (
         for_elapsed_time => TG.elapsed_time(0));

      ... more code that cannot be interrupted by process_2 or process_3

      -- (1) --
      process_attribute_modifiers.wait (
         for_elapsed_time => TG.elapsed_time(0));

   end loop;

end process_1;

--
-- process_2 and process_3 each do another third of the "time shared"
-- processing
--
with process_attribute_modifiers;
```

```
procedure process_2 is

   --
   --  (2)  allow either process_1 or process_3 to run by voluntarily
   --       descheduling by calling wait (0)
   --

begin

   loop

      ...code that cannot be interrupted by process_1 or process_3

      -- (2) --
      process_attribute_modifiers.wait(
         for_elapsed_time => TG.elapsed_time(0));

      ...

   end loop;

end process_2;

with process_attribute_modifiers;
procedure process_3 is

begin

    ... similar use of code and wait(0)

end process_3;

with process_managers;
with process_table;
with process_1;
with process_2;
with process_3;
procedure Main_Unit is

   proc_1 : process_table.process_identifier;
   proc_2 : process_table.process_identifier;
   proc_3 : process_table.process_identifier;

   --
   -- the Main Unit creates all three processes
   --

begin

   --
   -- network initialization and other initialization
   --

   proc_1 := process_managers.declare_process ("P1");
   proc_2 := process_managers.declare_process ("P2");
   proc_3 := process_managers.declare_process ("P3");
   process_managers.create_process (
      process_ID => proc_1,
      initial_priority => 4,
      address => process_1'address);

   process_managers.create_process (
      process_ID => proc_2,
      initial_priority => 4,
      address => process_2'address);

   process_managers.create_process (
      process_ID => proc_3,
      initial_priority => 4,
      address => process_3'address);

    --
    -- rest of initialization
    --

end Main_Unit;
```

E.8. Message Router

While in many instances, it is convenient for every process to know the exact destination of each message that it sends, at times it is more convenient for a process to know only the concept of a "service" (and not a specific "server"). In these cases, the notion of a service process (or message router) is useful, in which the message router partially decodes messages and sends them to the specific process responsible for acting on the service request.

E.8.1. Example Requirements and Justification

A specific, although simple, example of the need for this type of message passing can be shown in the following set of requirements:

- An application accesses data from numerous sensors and processes this data in a like manner irrespective of origin.
- The sensors are distributed, as are the processes which read the sensors.
- The frequency of acquisition of data from any given sensor is stochastic, although the computational overhead is uniform for any given data sample.
- To spread the system load, the processes that analyze the incoming data are also distributed. The number of data-gathering processes exceeds the number of analyzing processes.
- Sensor data is passed to the analyzing processes via messages. The analysis processes read the incoming data, compute some form of result, and communicate this result to the collector. One possible application of this form would be a multiple target tracking algorithm; another would be a print server spooling requests to multiple printers.

Given this set of constraints, two possible implementations are envisioned:

1. Each data collection process (associated with a single sensor) has associated with it a single analysis process (a many-to-one mapping). Because collection processes outnumber analysis processes, it is possible that a set of collection processes could be active that would heavily load a small set of analysis processes, leaving a different set of analysis processes relatively idle.

   If load sharing were to be implemented with this scheme, every collection process would need to know the location of every analysis process, and every collection process would also have to maintain a copy of the loading tables for the analysis processes, and would thus route messages to what it presumed to be the least loaded analysis process.

   This method would result in a great deal of data sharing, a large number of global variables, increased message traffic (since the collection and analysis processes are both distributed), and increased complexity of every collection process.

2. Each data collection process (associated with a single sensor) knows about a single, global service process (or message router). This service process would receive incoming requests from the collection processes, and based on the relative loading of the analysis processes, would choose a lightly loaded one and route the message to it for analysis. The analysis process would then be able to respond directly to the collection process (assuming the message router provided the address of the collect process to it).

   This scheme requires some increased message traffic, but substantially reduces the computational overhead of the collection processes. Each collection process need only be aware of a single, global message router, whose sole job it is to find unloaded analysis processes and ship sensor information messages off to them. Information about process load is localized, so no data sharing need be done.

E.8.2. PDL of Example

The code for the message router would look something like this:

```
with process_table;
package load_information is

   function find_least_loaded_analysis_process
      return process_table.process_identifier;

end load_information;

with communication_globals;
with process_table;
with load_information;
procedure message_router is

   unloaded : process_table.process_identifier;

   function to_tag (sender : in process_table.process_identifier)
      return communication_globals.message_tag_type;
      --
      -- encode a process_ID as a message_tag_type value so the ultimate
      -- receiver of the message may determine from which process it
      -- originated
      --

   --
   -- read an incoming message and assume that it comes from a collection
   -- process.  in more complicated cases, the message_tag or the
   -- message_buffer could indicate the type of service requested.
   --

begin

   loop
      communication_management.receive_message
      (
         sender, ... , length, buffer, ...
      );

      unloaded := load_information.find_least_loaded_analysis_process;

      communication_management.send_message
      (
         receiver => unloaded,
         message_tag => to_tag (sender),
         message_length => length,
         message_text => message_buffer
      );

    end loop;

end message_router;
```

E.9. Process Monitor (A Sample Tool)

This example will be provided in a future version of this document.

E.10. Network Integrity

This example will be provided in a future version of this document.

E.11. Prioritized Messages

This example will be provided in a future version of this document.

Appendix F: Application Example

This information will be provided in the next version of this document.

Appendix G: Relation to Standard Design Models

This appendix is currently in outline format.

G.1. Introduction

The Kernel presupposes certain models of real-time system development.

G.2. Basic Models

G.2.1. Process Model

Serial algorithms executing independently of each other, synchronizing by explicit signals or data flows.

G.2.2. Data Flow Model

Loosely-coupled producers, consumers and transducers, linked by persistent data-flow arcs along which typed messages are passed.

G.2.3. Time Model

Uniform linear flow of time common to all nodes, represented internally with a finite granularity.

G.2.4. Event Model

An event is the arrival of a signal or message; conceptually asynchronous. Processes await events that represent preconditions for continued execution; processes create postconditions that represent events being awaited by other processes.

G.2.5. Device Model

Device as an external source or sink of data; controlled by a driver/handler pair; the driver being synchronous with respect to the application and the handler asynchronous. Driver maps application requests into device commands; handler maps device responses into application events.

G.3. Corresponding Design Models

G.3.1. System Decomposition Models

From requirements to: processes, communication paths, sources and sinks.

G.3.2. Data Flow Models

From architecture to: data channels, data stores, message types, process roles (producer/consumer/transducer).

G.3.3. Transaction Models

From behavioral specification to: end-to-end transactions; pre and post conditions; invariants; internal states; state machines.

G.3.4. Temporal Models

From performance specifications to: event recognition and handling times; maximum latencies; resource requirements; throughput.

G.4. Suggested Standard Techniques

Decomposition:	Ward-Mellor; SA/SD
DFD:	MASCOT
Transaction:	Statecharts
Temporal:	Schedulability Analysis; Timelines

Appendix H: 68020 Specifics

The following information is specific to the Motorola 68020 target, as defined by the documents listed in Section 1.4:

1. There are 256 legal interrupts.
2. The range of legal interrupts is: 0 .. 255. This is the range for type *interrupt_name* in package *generic_interrupt_globals*. The declaration of type *interrupt_name* is:

```
type interrupt_name is new hardware_interface.hw_byte;
```

3. Interrupt names reserved by the Kernel and the hardware are:

Kernel Reserved Interrupts	
Interrupt Name	Meaning
0-63	Hardware-defined exceptions
66	PIT PIO "In" port #1
68	PIT Timer #1 (Timer A)
74	PIT PIO "In" port #2
76	PIT Timer #2 (Timer B)
82	PIT PIO "Out" port #1
84	PIT Timer #1 (Timer C)
90	PIT PIO "Out" port #2
92	PIT Timer #2 (Timer D)
100	MFP Timer D
101	MFP Timer C
104	MFP Timer B
109	MFP Timer A
120	SIOA TX Port A
122	SIOA RX Port A
124	SIOA SC Port A
126	SIOA SRC Port A
255	Interprocessor Interrupt

4. Information about the resources consumed by each Kernel primitive will be provided in the next version of this document.
5. The values used by the DARK development team for all tailoring parameters are presented in Table H-1.
6. The *context_save_area* embedded within the Process Table is target-specific. See Appendix A, package *context_save_area*, for a detailed description.
7. The following representation specification is relevant to type *hw_bits8* declared in package *hardware_interface*:

```
for hw_bits8, use record
   bit7 at 0..range 0..0;
   bit6 at 0..range 1..1;
   bit5 at 0..range 2..2;
   bit4 at 0..range 3..3;
   bit3 at 0..range 4..4;
   bit2 at 0..range 5..5;
   bit1 at 0..range 6..6;
   bit0 at 0..range 7..7;
end record;
```

This representation specification allows individual bit fields to be referenced by name in a manner compatible with the target hardware.

Default Values for Tailoring Parameters		
Package Name	Parameter Name	Value
communication_globals	*maximum_message_length_value*	1_024
interrupt_globals	*number_of_interrupt_names_used_by_application*	10
	number_of_interrupt_names_used_by_Kernel	4
Kernel_time	*ticks_per_second_value*	500_000
network_configuration	*maximum_length_of_processor_name_value*	16
	number_of_nodes_value	4
network_globals	*first_bus_address_value*	0*
	last_bus_address_value	255*
	null_bus_address_value	16#00#
process_managers	*maximum_message_queue_size_value*	1_024
	maximum_process_stack_size_value	4_096
process_managers_globals	*maximum_length_of_process_name_value*	32
process_table	*maximum_number_of_processes_value*	25
schedule_types	*lowest_priority_value*	10
timeslice_management	*minimum_slice_time_value*	77 µsec
all packages with error checking parameters	**_enabled*	true

Table H-1: Tailoring Parameters

*These are constants in the code that should be treated like generic formal parameters.

Appendix I: Index

This will be provided in the next version of this document.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
NONE

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-89-UG-1

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-89-TR-15

6a. NAME OF PERFORMING ORGANIZATION
SOFTWARE ENGINEERING INSTITUTE

6b. OFFICE SYMBOL *(If applicable)*
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

6c. ADDRESS *(City, State and ZIP Code)*
CARNEGIE MELLON UNIVERSITY
PITTSBURGH, PA 15213

7b. ADDRESS *(City, State and ZIP Code)*
ESD/XRS1
HANSCOM AIR FORCE BASE, MA 01731

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
SEI JOINT PROGRAM OFFICE

8b. OFFICE SYMBOL *(If applicable)*
SEI JPO

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962885C0003

8c. ADDRESS *(City, State and ZIP Code)*
CARNEGIE MELLON UNIVERSITY
SOFTWARE ENGINEERING INSTITUTE JPO
PITTSBURGH, PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
	N/A	N/A	N/A

11. TITLE *(Include Security Classification)*
Kernel User's Manual Version 1.0

12. PERSONAL AUTHOR(S)
Judy Bamberger, Tim Coddington, Robert Firth, Daniel Klein, David Stinchcomb, R. Van Scoy

13a. TYPE OF REPORT
FINAL

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Yr., Mo., Day)*
February, 1989

15. PAGE COUNT

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This manual describes the models underlying the Kernel and its concept of operations, presents the primitives available to the application program, and provides a number of abstractions that may readily be built on top of Kernel primitives.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT □ DTIC USERS ☒

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED, UNLIMITED

22a. NAME OF RESPONSIBLE INDIVIDUAL
KARL SHINGLER

22b. TELEPHONE NUMBER *(Include Area Code)*
(412) 268-7630

22c. OFFICE SYMBOL
SEI JPO

EDITION OF 1 JAN 73 IS OBSOLETE

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE