



Carnegie Mellon University
Software Engineering Institute

(1)

AD-A219 295

Kernel Architecture Manual

Judy Bamberger
Timothy Coddington
Currie Colket
Robert Firth
Daniel Klein
David Stinchcomb
Roger Van Scoy

December 1989

DTIC
ELECTE
MAR 16 1990
S D

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

90 08 15 057

Technical Report

CMU/SEI-89-TR-19

ESD-89-TR-27

December 1989

Kernel Architecture Manual



Judy Bamberger
Timothy Coddington
Currie Colket
Robert Firth
Daniel Klein
David Stinchcomb
Roger Van Scoy

Distributed Ada Real-Time Kernel Project

Accession For	
NTIS CRASH	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution	
Availability Codes	
Dist.	Availability Codes
A-1	

Approved for public release.
Distribution unlimited.



Software Engineering Institute

Carnegie Mellon University
Pittsburgh Pennsylvania 15213

This report was prepared for the

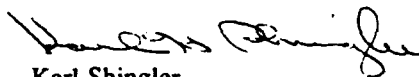
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report 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 report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl Shingler
SEI Joint Program Office

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

Copyright © 1989 Carnegie Mellon University

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.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

I. Overview	1
1. Introduction	1
1.1. Definition of Terms	2
2. Software Architecture Overview	4
3. Hardware Architecture Overview	9
3.1. The DARK Testbed	9
3.1.1. Target Processor	9
3.1.2. VME Bus	10
3.1.3. Host Computers	10
3.1.4. Distributed Network	10
3.2. Network Processor	10
3.2.1. Nproc-to-Nproc Interface	13
3.2.1.1. Parallel Interface	16
3.3. Kernel Processor	16
3.3.1. Kproc-to-Nproc Interface	16
3.3.2. Time Synchronization Bus Interface	18
3.3.3. Non-Kernel Device	18
II. Application Interface	21
1. Alarm Management	22
2. Communication Management	25
3. Hardware Interface	36
4. Interrupt Management	37
5. Process Attribute Modifiers	41
6. Process Attribute Readers	46
7. Process Managers	49
8. Processor Management	54
9. Semaphore Management	63
10. Time Globals	66
11. Time Management	73
12. Timeslice Management	76
13. Tool Interface	79
III. Core Kernel	85
1. Bus I/O	86
2. Clock	94
3. Context Switcher	97
4. Exception_Raiser	99
5. Internal Process Management	101
6. Kernel Interrupt Management	104
7. Kernel Time	107

8. Network Configuration	113
9. Process_Encapsulation	115
10. Process Index Table	117
11. Process Table	121
12. Scheduler	123
13. Time Keeper	138
14. Tool Logger	149
IV. Communication Subsystem	153
1. Communication Overview	154
1.1. Design Decisions	155
2. Data Structures	157
2.1. Datagram Data Structures	157
2.2. Datagram_Pointer	157
2.3. Datagram_Class	157
2.4. Datagram	158
3. Semaphores and Atomic Regions	163
4. Datagram Management	165
5. Packet Communication	171
6. Kproc/Nproc Interface	172
6.1. Shared Memory	172
6.2. Enqueueing Messages for Transmission	172
6.3. Receiving Incoming Messages	173
7. Nproc Communication Routines	174
8. Message Transfer Thread Examples	182
8.1. Detailed Thread Description	182
8.2. Graphic Representation of Thread	183
V. General Utilities	191
1. Low_level_storage_manager	192
2. Storage Manager	194
3. Queue Manager	196
VI. Target-Specific Utilities	197
1. Interrupt Names	198
2. Low_level_hardware	200
3. Memory Addresses	203
4. MVME133A Definitions	205
5. MZ8305 Definitions	207
6. SCC_porta	209
7. Timer_controller	218

VII. Debug Utilities	223
1. CSA_debug	224
2. Debug	226
3. dgg_debug	228
4. Make NCT	231
5. NCT_debug	233
6. PTB_debug	235
7. semaphore_debug	239
VIII. 68020 Hardware Configuration	241
1. Target Processor Board	242
1.1. MVME133A Board	242
1.1.1. Local Memory	242
1.1.2. Floating Point Coprocessor	243
1.1.3. Real-Time Clock	243
1.1.4. Serial Debug Port	243
1.1.5. Serial Ports A and B	243
1.1.6. Timers	243
1.1.7. Interrupts	244
1.1.8. ROM, PROM, EPROM, and EEPROM Sockets	244
1.1.9. VME System Controller	244
1.1.10. P1 And P2 Connector	244
1.2. Kernel Processor Board Configuration	245
1.3. Network Processor Board Configuration	246
2. Parallel Interface	247
2.1. MZ8305 Board	247
2.2. Parallel Interface/Timer	247
2.2.1. Parallel I/O Connector	247
2.3. Input Port Parallel Board Configuration	248
2.4. Output Port Parallel Board Configuration	248
3. Shared Memory	249
3.1. Shared Memory Board Configuration	249
4. VME Chassis	250
4.1. Backplane Jumper Configuration	251
5. Equipment Rack	252
6. Host System	253
6.1. Serial I/O Ports	253
7. Test Equipment	255
7.1. Test Equipment Hardware	255
8. Low-Level I/O	256
8.1. Software	257
9. Interrupts	258

9.1. Interrupt Request Levels	258
9.2. Interrupt Vector Numbers	258
9.3. Interrupt Configuration Summary	258
10. Memory Map	260
11. Network Cable	261
12. Synchronization Bus	262
12.1. Bus Description	262
12.2. Bus Operation	262
13. P2 Backplane Connector Wiring	266
IX. TeleSoft Ada Compiler Dependencies	267
1. Major Dependencies	268
1.1. Software Architecture and Design	268
1.2. Basic Data Types	268
1.3. Encapsulation of Assembler	269
2. Software Architecture and Design Dependencies	270
2.1. Code Customization	270
2.2. Representation of Errors	271
2.3. Module Initialization	272
2.4. Chapter 13 Issues	272
2.5. Pragmas	273
2.6. Ada Use Subset	274
3. Basic Data Types and Operations	275
3.1. Sizes of Data Types	275
3.2. Untyped Storage	275
3.3. Integer Types	276
3.4. Duration	277
3.5. Machine Addresses	278
3.6. Strings	278
4. Encapsulation of Assembly Code	279
4.1. Linkage	279
4.2. Program and Data Sections and Attributes	279
4.3. Data Representation	280
4.4. Access to Ada Objects from Assembly Code	280
4.5. Access to Assembler Objects from Ada Code	280
4.6. Procedural Interface	280
4.7. Exceptions	281
Appendix A. Data and Control Flow Diagrams	282

Appendix B. Kernel Interface Control Document	285
Appendix C. Race Conditions	287
C.1. Process Table - Context Save Area	288
C.2. Process Table - Schedule Attributes	288
C.3. Process Table - Message Queue	288
C.4. Process Table - Pending Activities Attributes	290
C.5. Process Table - Semaphores Attributes	290
C.6. Table - Tool Interface Attributes	290
C.7. Interrupt Table	291
C.8. Network Configuration Table	291
C.9. Timeslice Parameters	291
Appendix D. 68020 Assembler Interface	292
D.1. Linkage	292
D.2. Program and Data Sections	293
D.3. Data Representation	293
D.4. Access to Ada Objects from Assembly Code	293
D.5. Access to Assembler Objects from Ada Code	293
D.6. Procedural Interface	293
D.6.1. Entry and Exit Protocol	293
D.6.2. Register Usage	294
D.6.3. Stack Manipulation	294
D.6.4. Parameter Passing	294
D.6.4.1. Mode of Transmission	294
D.6.4.2. Manner and Order of Transmission	294
D.6.4.3. Accessing Parameters and Returning Function Results	294
D.6.5. Example	295
D.7. Exceptions	296
D.7.1. Raising Exceptions	296
D.7.2. Exception Propagation	296
D.7.3. Guarded Regions	296
Appendix E. 68020 Tailoring	297
E.1. Sizes of Data Types	297
E.2. Untyped Storage	297
E.3. Integer Types	298
E.4. Duration	298
E.5. Machine Addresses	298
E.6. Strings	299

Appendix F. Procedure to Requirement Mapping	300
Appendix G. Requirement to Procedure Mapping	334
Appendix H. Short Names	355
Appendix I. Overview of VMS Version	358
Appendix J. VMS Ada Compiler Dependencies	360
J.1. introduction	360
J.1.1. Relevant Documents	360
J.2. Major Dependencies	360
J.2.1. Software Architecture and Design	360
J.2.2. Basic Data Types	361
J.2.3. Encapsulation of Assembler	361
J.3. Software Architecture and Design Dependencies	361
J.3.1. Code Customization	361
J.3.2. Representation of Errors	362
J.3.3. Module Initialization	362
J.3.4. Chapter 13 Issues	362
J.3.5. Pragmas	363
J.3.6. Ada Use Subset	364
J.4. Basic Data Types and Operations	364
J.5. Encapsulation of Assembly Code	364
Appendix K. VAX-11 Assembler Interface	365
K.1. Linkage	365
K.2. Program and Data Sections	365
K.3. Data Representation	365
K.4. Procedural Interface	366
K.4.1. Entry and Exit Protocol	366
K.5. Register Usage	366
K.5.1. Stack Manipulation	366
K.5.2. Parameter Passing	366
K.5.3. Mode of Transmission	367
K.5.4. Manner and Order of Transmission	367
K.5.5. Accessing Parameters and Returning Function Results	367
K.6. Exceptions	367
K.6.1. Raising Exceptions	367
K.6.2. Exception Propagation	367
K.6.3. Guarded Regions	368
Appendix L. VAX-11 Tailoring	369
L.1. Sizes of Data Types	369

L.2. Untyped Storage	370
L.3. Integer Types	371
L.4. Duration	371
L.5. Machine Addresses	371
L.6. Strings	372

List of Figures

Figure 1: Stylized Package Template	2
Figure 2: Kernel Context Diagram	5
Figure 3: Kernel Level 1 DFD	6
Figure 4: User View of the Kernel	8
Figure 5: Host Configuration	11
Figure 6: Distributed Network	11
Figure 7: Network Processor Hardware	12
Figure 8: DARK Network Overview	14
Figure 9: Node-to-Node Connections	15
Figure 10: Parallel Interface Hardware	16
Figure 11: Kernel Processor Hardware	17
Figure 12: Kproc-to-Nproc Interface	17
Figure 13: Non-Kernel Message Header	19
Figure 14: Set Alarm	23
Figure 15: Network Initialization Protocol: Phase 1	55
Figure 16: Network Initialization Protocol: Phase 2	56
Figure 17: Network Configuration Table	114
Figure 18: Process Mapping Table	120
Figure 19: Process Table	121
Figure 20: Pending Activity States	122
Figure 21: Application Blocks	126
Figure 22: Application Unblocks	127
Figure 23: Run Queue	129
Figure 24: Setting an Alarm Event	140
Figure 25: Event Expiration	141
Figure 26: Time Event Queue	147
Figure 27: Data Flow Through the Kernel and Network	154
Figure 28: Packet Layout	171
Figure 29: Send_Message – Application Message to Datagram	184
Figure 30: Output Message Queue	185
Figure 31: Datagram to Packet Data Flow	186
Figure 32: Packet Traffic onto Network	187
Figure 33: Packet Traffic off Network	188
Figure 34: Receive_Message – Datagram to Application Message	189
Figure 35: Sync Processing	211
Figure 36: Chassis Hardware	250
Figure 37: Equipment Rack	252

Figure 38:	VAX Ports to Testbed Ports Cross Reference	254
Figure 39:	Interrupt Summary	259
Figure 40:	Memory Map	260
Figure 41:	Flat Ribbon Cable Schematic	261
Figure 42:	Kproc to Synchronization Bus Interface	263
Figure 43:	P2 Cable Harness Schematic	264
Figure 44:	Store Notation	282
Figure 45:	Process Notation	283
Figure 46:	Flow Notation	284
Figure 47:	VMS Overview	359

List of Tables

Table 1: Kernel Message Formats	285
Table 2: Synchronization Message Formats	286
Table 3: Initialization Message Formats	286
Table 4: Tool Interface Message Formats	286

Kernel Architecture Manual

Abstract: This document contains the detailed design description of the Kernel. The overall system architecture and the rationale for it are presented as relevant to both the application (i.e., the *external* view of the Kernel) and the Kernel maintainer (i.e., the *internal* view of the Kernel). This document presents the algorithms and data structures needed to implement the functionality defined in the *Kernel Facilities Definition*. This document also contains an in-depth description of the communication protocol used by the Kernel, both the network software and hardware that compose the DARK testbed at the SEI, and a detailed enumeration of all compiler dependencies exploited by Kernel software. This document is geared toward engineers responsible for porting and maintaining the Kernel and engineers requiring detailed information about the internals of the Kernel.

I. Overview

1. Introduction

Part I of this document describes the architecture of the 68020/TeleSoft TeleGen2 version of the Distributed Ada Real-Time Kernel (DARK). This description is divided into the following parts:

- Overview: provides a top-level architectural view of the Kernel software and hardware.
- Application Interface: describes the design of the user-visible portions of the Kernel. The user invokes Kernel operations via this interface (found in Part II.)
- Core Kernel: describes the design of the internal packages of the Kernel. These packages implement the functionality of the primitives, but are not visible to the user. The internal Kernel packages are described in Part III.
- Communication Subsystem: describes the design of the software portion of the communication network implemented for the DARK testbed (found in Part IV).
- General Utilities: describes the design of the general, internal support packages used by the Kernel (found in Part V).
- Target-Specific Utilities: describes the design of the target-specific, internal support used by the Kernel (found in Part VI).
- Debug Utilities: describes the testing and debugging support packages used in implementing the Kernel (found in Part VII).
- DARK Testbed: describes the target hardware configuration on which the software described in this document depends (found in Part VIII).

- **Ada Compiler Dependencies:** describes the interface to the Ada compiler of the Kernel (found in Part IX).

The packages discussed in this document are all described in the same manner:

- A stylized picture showing the exported types, operations, and exceptions for the package, as illustrated in Figure 1.

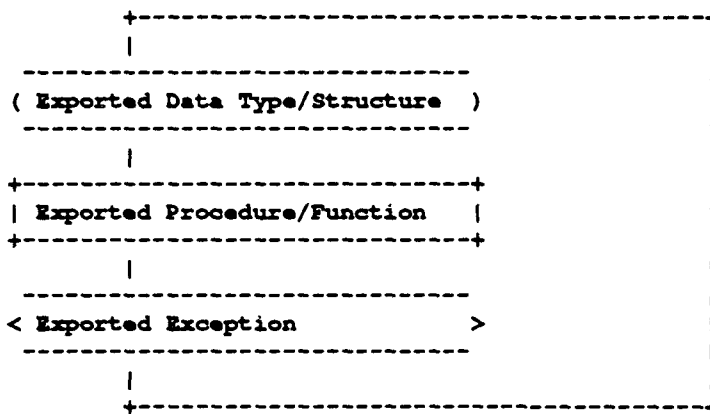


Figure 1: Stylized Package Template

- A general discussion of the package.
- For each visible entry in the package and each major internal object of the package:
 1. General remarks about the object.
 2. An invocation sequence (where appropriate).
 3. PDL. In the PDL, *italic font is used to denote abnormal returns or blocking conditions.*

Some of the packages will contain data flow diagrams (DFD) illustrating the functioning of the package in the overall system context. The notation used in these diagrams is explained in Appendix A.

1.1. Definition of Terms

The following terms and their definitions are intended to clarify their meaning and identify how they are used in the context of this document.

DARK hardware. All of the hardware equipment used to build the processor nodes and ring network.

Datagram. The basic unit of communication between a Kproc and an Nproc at the ISO Data Link layer. A datagram contains an application or Kernel message that is transmitted to another process.

DARK testbed. Testbed for short; comprises all of: DARK hardware, host computers, and terminals.

Host system. Comprises μ VAX-II computers; used to download executable images to the nodes and debug Kernel and application code online.

Kernel processor. Kproc for short; one of two processor boards located on a processor node. It executes the Kernel and application processes and provides all of the computing resources for the node.

Network processor. Nproc for short; one of two processor boards located on a processor node. It has been programmed and configured to operate as a network port on the ring network.

Packet. The basic unit of communication between Nprocs at the ISO Physical layer. Each packet comprises 32 bits of information: 8-bit sender address, 8-bit receiver address, and 16 bits of data. The Nproc breaks datagrams into packets for transfer across the DARK network.

Processor node. All the hardware components co-located in a single VME chassis to provide the functionality of a single part of a distributed processor system. It includes a parallel interface, two processor boards, shared memory board, and chassis hardware. The term **node** is synonymous with **processor node**.

Shared Memory. That memory in a processor node provided by the shared memory board. It is accessible by both the Nproc and Kproc, and is separate from the local memory on each processor board.

2. Software Architecture Overview

The detailed architecture of the Kernel is presented in the parts that follow. This chapter is an overview and rationale for the Kernel software architecture.

When viewed as a single entity, the Kernel's context diagram looks like that shown in Figure 2. This shows that the Kernel interfaces with four external entities:

1. Application
2. Event timer
3. Communication subsystem
4. Real time clock

Decomposing the context diagram one level (shown in Figure 3) reveals that the Kernel is comprised of five major areas.

1. Application interface
2. Time keeper
3. Datagram management
4. Clock interface
5. Core Kernel

The **Application Interface** is the functionality exported by the Kernel to the user. These packages are shown in Figure 4 as windows extending into the Kernel (the use of this interface is characterized in [KUM 89]). The packaging structure shown in Figure 4 was arrived at by applying the following design goals:

1. Closely related functions grouped together
2. No cross dependencies between user visible packages
 - a. to allow deletion of unneeded functionality
 - b. to allow selective replacement of functionality
3. Common types extracted into support packages
4. Conditional compilation flags to control Kernel-generated exceptions
5. Generics used to control compilation flags and tailorable parameters

Each of these packages is discussed in more detail in Part II.

The **Clock interface**, **Time Keeper** and **Core Kernel** represent the working part of the Kernel. These packages are shown inside the black box of Figure 4. These packages are structured to meet the following design constraints:

1. Isolation of the compiler-dependent parts, to enable rehosting the Kernel on different compilers.
2. Isolation of hardware-dependent parts, to enable rehosting the Kernel on different target configurations.

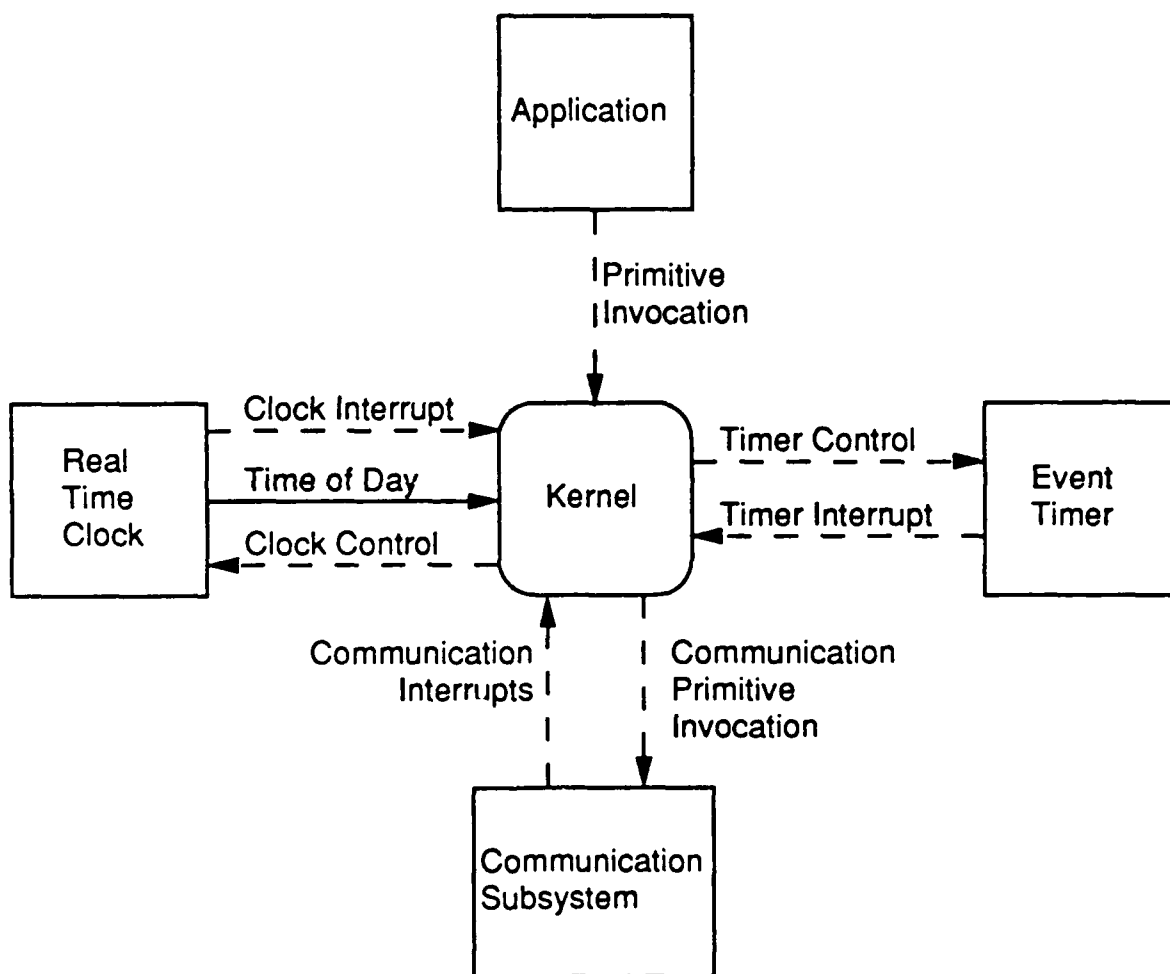


Figure 2: Kernel Context Diagram

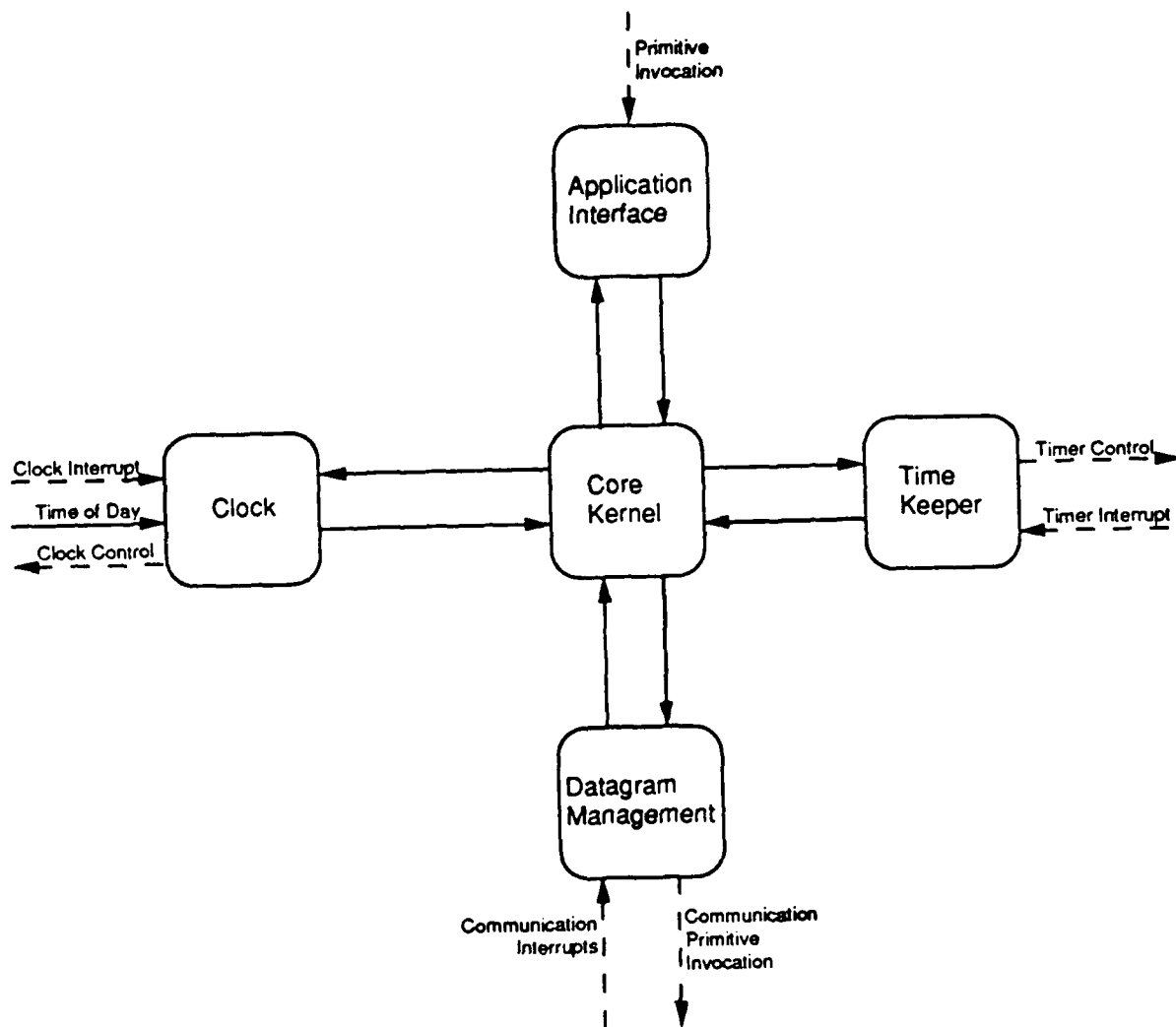


Figure 3: Kernel Level 1 DFD

3. Isolation of the Scheduler, to allow different scheduling regimes to replace that provided by the DARK project.
4. Isolation of network initialization, to allow for different network startup schemes to replace that provided by the DARK project.
5. Selection of a small set of key data structures with which to drive the execution of the Kernel.

These packages are discussed in Part III.

The final piece is **Datagram Management**. This package is also inside the black box of Figure 4 and is the Kernel's interface to the communication subsystem (thus isolating both the application and the Kernel from the network). This piece is discussed in Part IV.

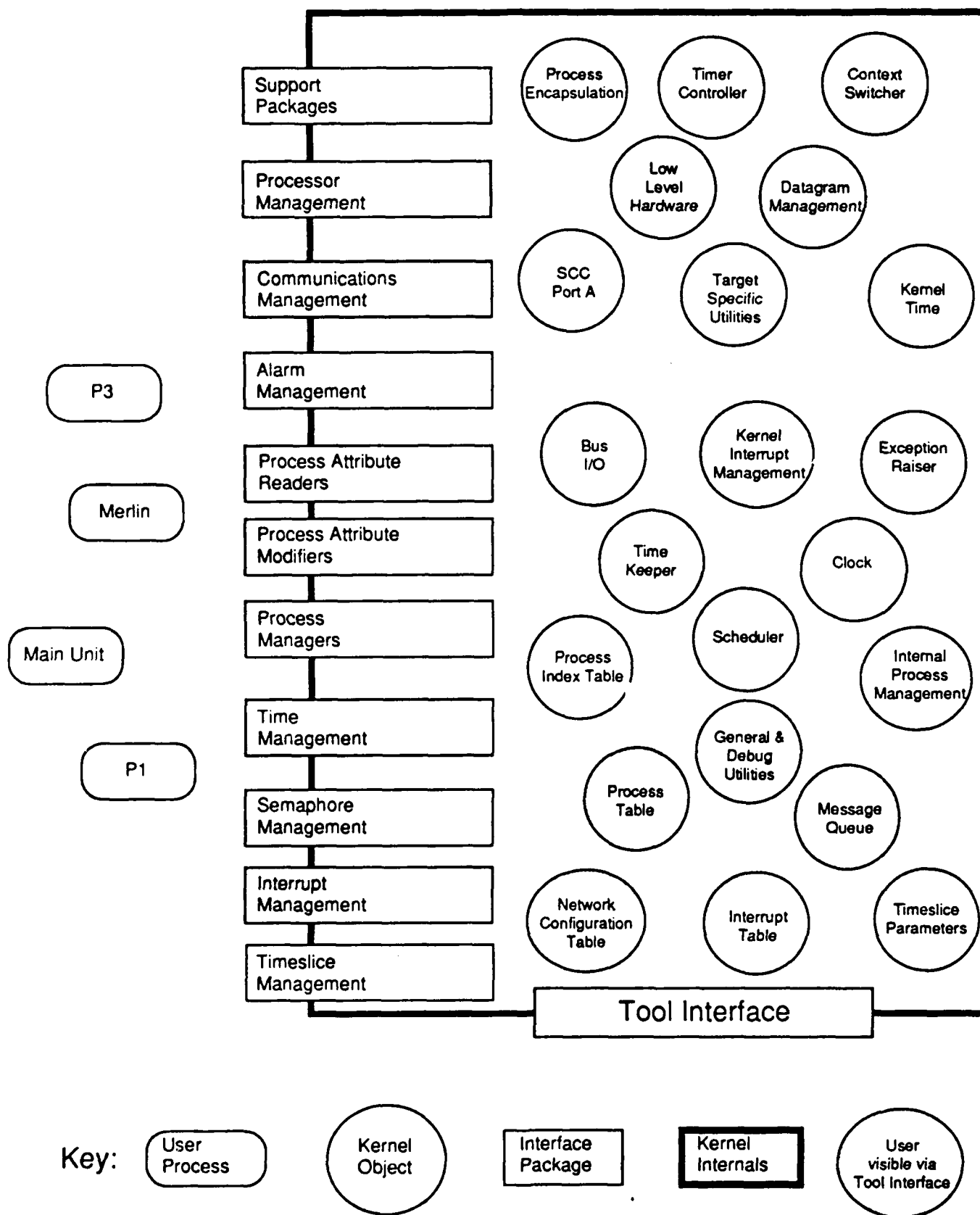


Figure 4: User View of the Kernel

3. Hardware Architecture Overview

The DARK prototype has been developed for a MC68020 microprocessor target. The prototype hardware configuration uses two processors per node on the network. One processor is designated the Kernel processor (Kproc); while the other is designated the network processor (Nproc). The Nproc is responsible for low-level communication across the network; the Kproc is responsible for all other Kernel operations, including execution of the application code.

All communication between nodes in the network is via datagrams. One implication of datagram-based communication is that the network is not responsible for verifying the correct or complete delivery of messages between processes; if a datagram cannot be sent (or received) by a node in the network due to network overload or data transmission errors, the datagram is discarded, and the sending (and receiving) processes receive no notification of the message's loss. It is the application engineer's responsibility to build any needed message validation and verification into the application code.

Parts IV and VIII describe all the software and hardware necessary to implement the datagram communication model described.

3.1. The DARK Testbed

This section presents a high-level overview of the system issues of the DARK testbed; more details can be found in Part VIII.

The DARK testbed was set up with the following goals:

- Use a validated Ada compiler.
- Exemplify a typical loosely coupled distributed system.
- Facilitate obtaining performance measurements.
- Use hardware components compatible to those already in use at the SEI.

The DARK testbed comprises a set of distributed processor nodes connected by a network. Each node has two 68020 processors: the Kproc for running the application and the Kernel, and the Nproc for handling node-to-node communication across the network. Using dual processor nodes allows the communication needs to be isolated from the processing needs of the node and simulates a number of real world applications where the lowest level of interprocessor communication is handled by special hardware.

3.1.1. Target Processor

The Motorola 68020 microprocessor is used in both industry and military systems for a variety of general-purpose and specialized applications, including embedded and distributed real-time systems. It is a popular choice for hardware designs requiring fast, efficient, and compact processing.

There are several Ada cross compilers and other development tools available for the MC68020. The MC68020 is available on several board formats, such as Multibus II and VME bus.

3.1.2. VME Bus

The DARK testbed is built around the VME board format. VME is a standard board supported by many third-party vendors. A large variety of processors and peripheral devices are available in this format. The VME bus specification document defines the VME bus, which specifies the type of board connector, number, name, and type of control and data signals available at the connector; and the protocol for interfacing to other devices, including interrupts, and read/write accesses.

The VME bus is used in the DARK testbed to couple together the Nproc, Kproc, shared memory, and two parallel interface boards of one node.

3.1.3. Host Computers

Four clustered μ VAX-II computers operate as host to the DARK target processors. They are used for various phases of software development, including creating and editing Ada source modules, compiling, linking, loading, and debugging.

The TeleSoft TeleGen2 Ada development system (see [TeleSoft 88]) and support tools are used for compile, link, loading, and testing of DARK software.

The various activities for programming and testing the target software can be carried out on the host remotely over the SEI Ethernet at user terminals.

Each processor node is connected, either directly or through a switch, to a host computer by two serial lines (see Figure 5). These ports are for downloading and debugging executable images.

3.1.4. Distributed Network

The distributed network consists of processor nodes and the data path connecting them. The DARK network and host system are shown in Figure 6. The network is a ring with four nodes, providing a computing resource for application use.

3.2. Network Processor

The Nproc is a VME board with a Motorola 68020 microprocessor running at 20MHz, as shown in Figure 7.¹ It is the part of a processor node that manages network communication for the Kernel.

¹It is identical to the Kproc for ease of replacement and maintenance.

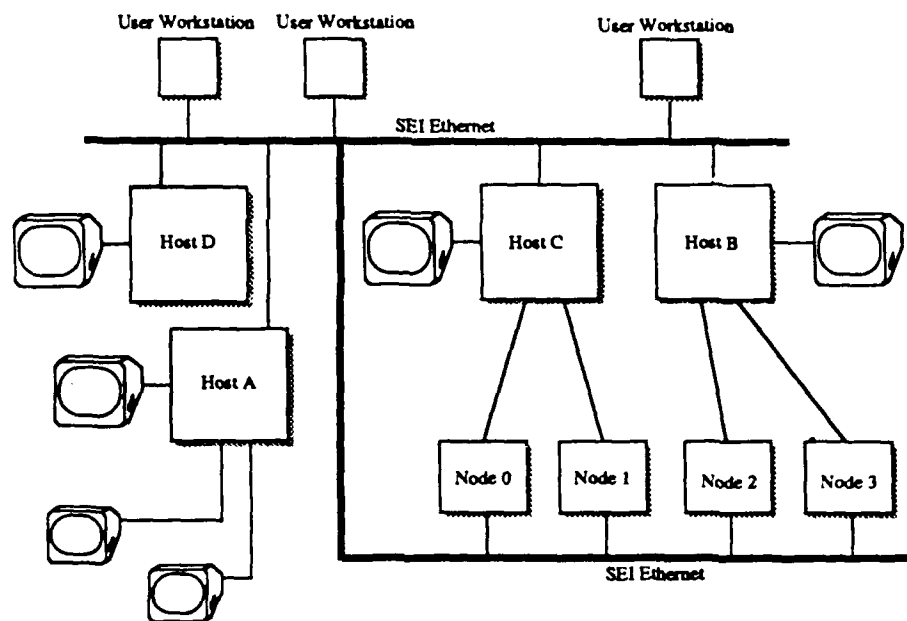


Figure 5: Host Configuration

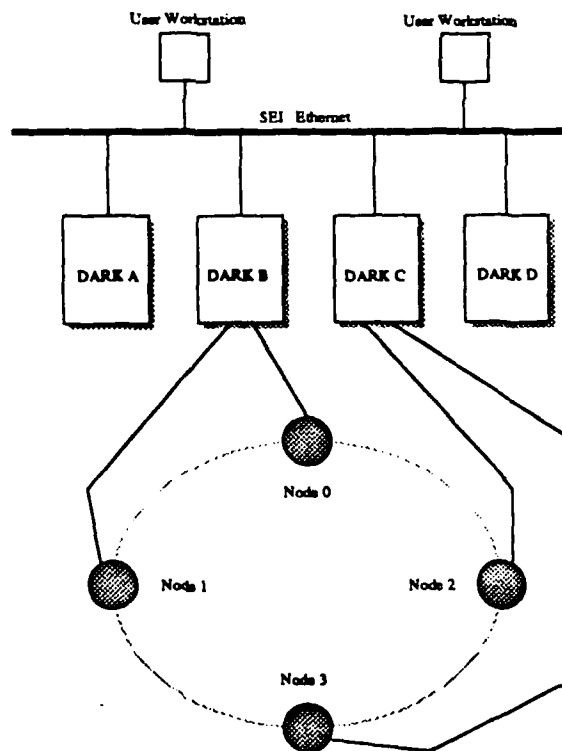


Figure 6: Distributed Network

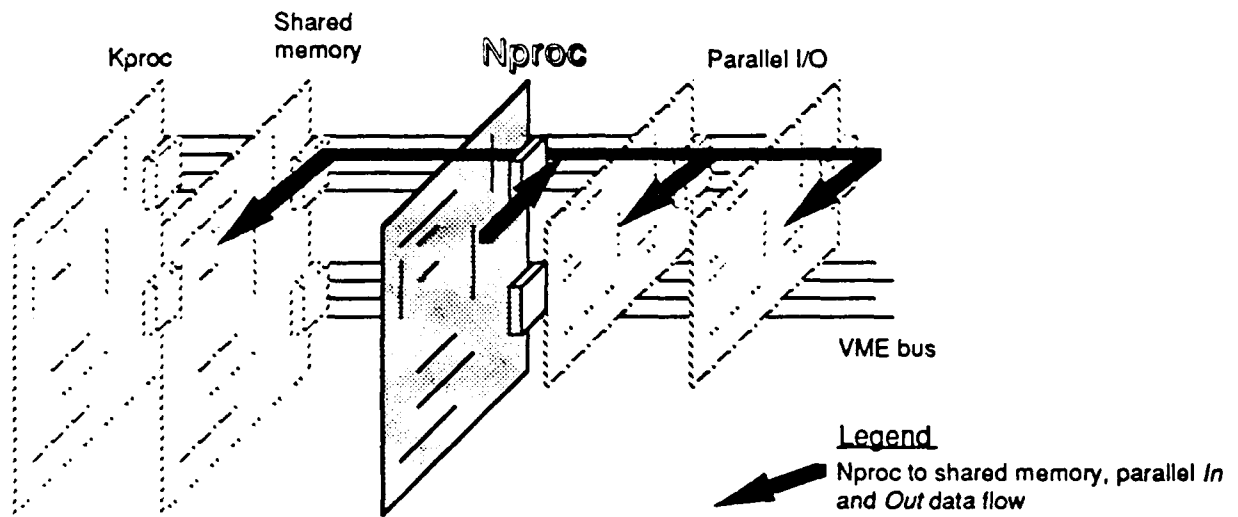


Figure 7: Network Processor Hardware

The Nproc has two serial ports connected to the host system. One is for debugging; the other is for downloading executable images. These ports are connected to one of the host machines.

The Nproc uses interrupts to communicate with the Kproc during message handling. In the current DARK implementation, there is no need for the Kproc to interrupt the Nproc, although the hardware configuration does permit it.

3.2.1. Nproc-to-Nproc Interface

The ring network, although circular in topology, is not continuous and relies on the operation of the Nprocs to keep information moving. The network is formed by a series of Nproc-to-Nproc interfaces. A packet, the basic unit of information, is passed from Nproc-to-Nproc until it reaches its destination. The purpose and content of the packets are discussed in Part IV.

Figure 8 shows the components of all four nodes and how they are connected. Each block corresponds to a VME board. This diagram is the basis for the following discussion.

The network hardware consists of Nprocs, parallel interfaces (ports), and flat ribbon cable segments. Each of these hardware components is discussed in more detail in Chapters 2 and 11, respectively. Figure 9 illustrates how they are connected.

For example, In a single transfer, the sending Nproc moves four bytes (one packet) of data from shared memory to the registers of the *out* parallel port. The port, in turn, transfers the register data to the cable segment that is connected to another (*in*) parallel port. The data are captured by the *in* parallel port and eventually transferred by the receiving Nproc to memory. What happens to the packet after that depends on whether or not it has reached its assigned destination. These software operations are discussed in Section 7.4.

The interface between the Nproc and the *in* and *out* parallel ports is interrupt-driven. When a packet is received at the *in* parallel port, the port raises an interrupt that is serviced by a special interrupt handler on the Nproc. When a packet is successfully transferred to an *out* parallel port, it too raises an interrupt. In this case, however, the interrupt is to indicate the port is no longer busy.

The parallel ports used provide a double buffering capability, thus permitting faster operation. Because of this feature, the *in* parallel port is capable of latching a second packet before the first has been transferred to memory by the Nproc, and the *out* parallel port can accept a second packet to output before the first has been completely accepted by the receiver on the other end of the interface.

Two pairs of handshake control lines and special read/write sequences make up the protocol for Nproc-to-Nproc transfers. A 32-bit packet is moved to the *out* parallel port in 4-byte pieces. The hardware is designed to manipulate the handshake lines automatically when the bytes are written into the respective registers in a specific order. The result of

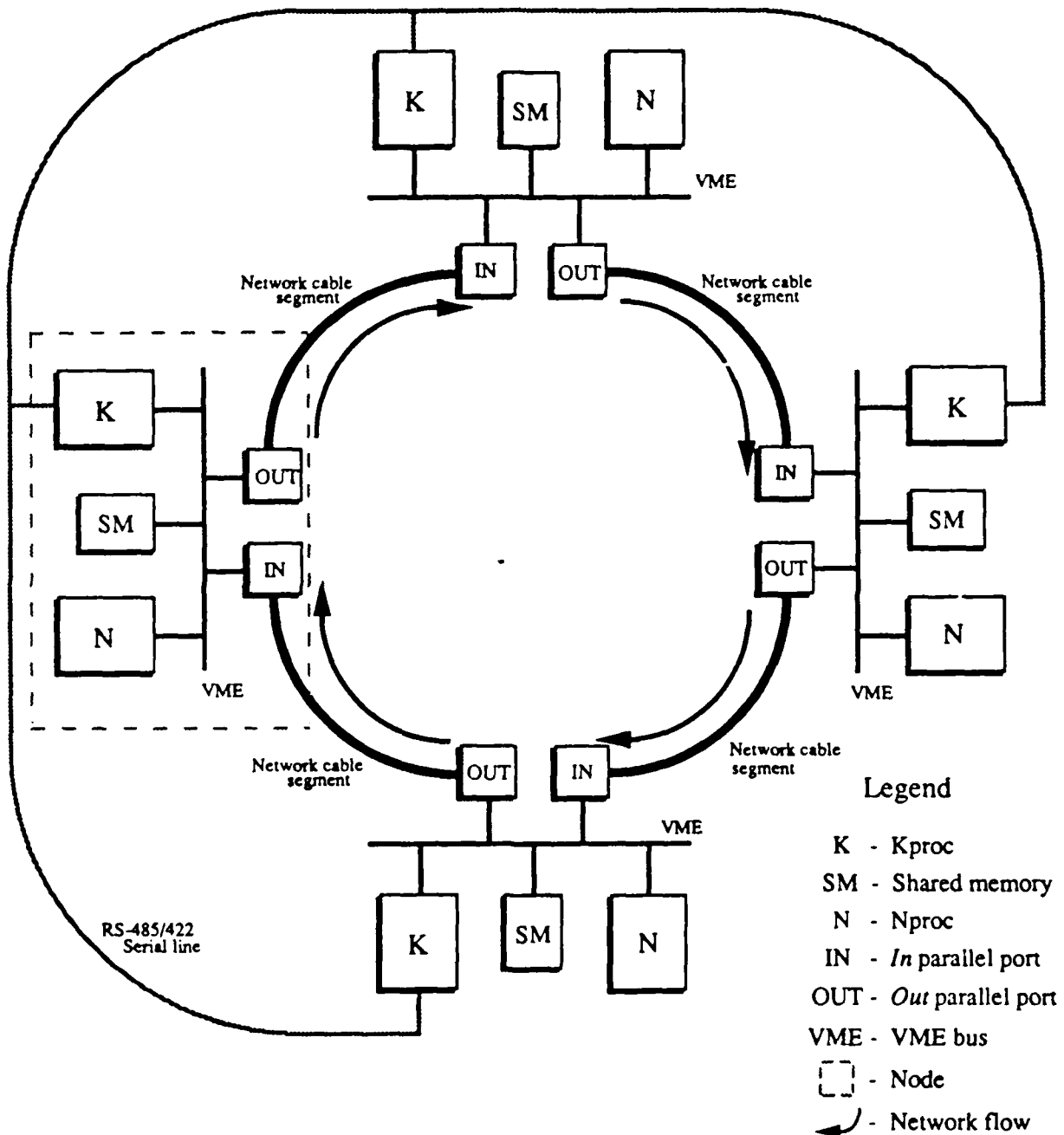


Figure 8: DARK Network Overview

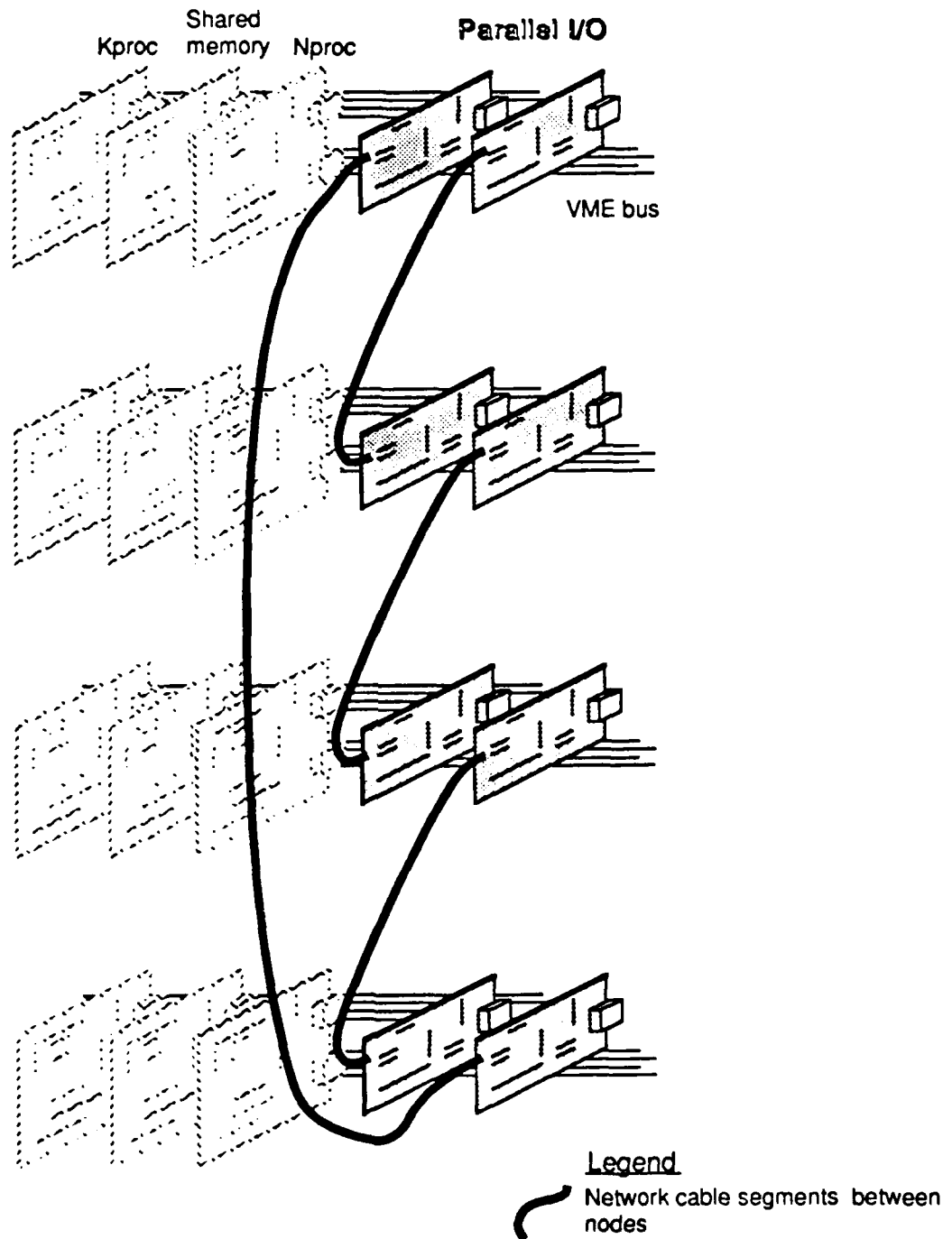


Figure 9: Node-to-Node Connections

doing so correctly is that the handshake signal causes the data to be latched, and in turn an interrupt is generated on the receiving end. There are two sets of handshake lines between parallel ports: one for the low 16-bits and one for the high 16-bits. They both operate the same, the only difference being that the high-order set is involved in generating the receiving port's interrupt after the last byte is moved into the port.

3.2.1.1. Parallel Interface

The parallel interface boards provide a 32-bit data path between two adjacent nodes in the ring network. Each node requires two of these interfaces, as shown in Figure 10. One is designated the *in* port and the other the *out* port.

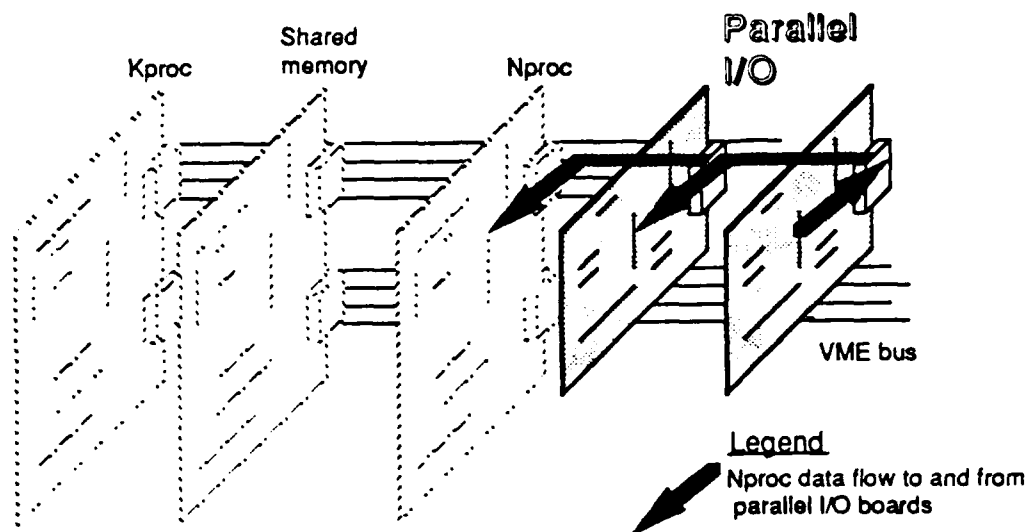


Figure 10: Parallel Interface Hardware

3.3. Kernel Processor

The Kproc is a VME board with a Motorola 68020 microprocessor running at 20MHz, as shown in Figures 11. It is that part of a processor node where the Kernel and application execute.

3.3.1. Kproc-to-Nproc Interface

The interface between the Kproc and Nproc is established using the VME bus, interrupts, and shared memory accesses. Both processors are part of the same node and are attached to the same VME bus (see Figure 8), which permits one processor to issue an interrupt to get the attention of the other.

When data need to be exchanged between the Kproc and Nproc, such as a message being

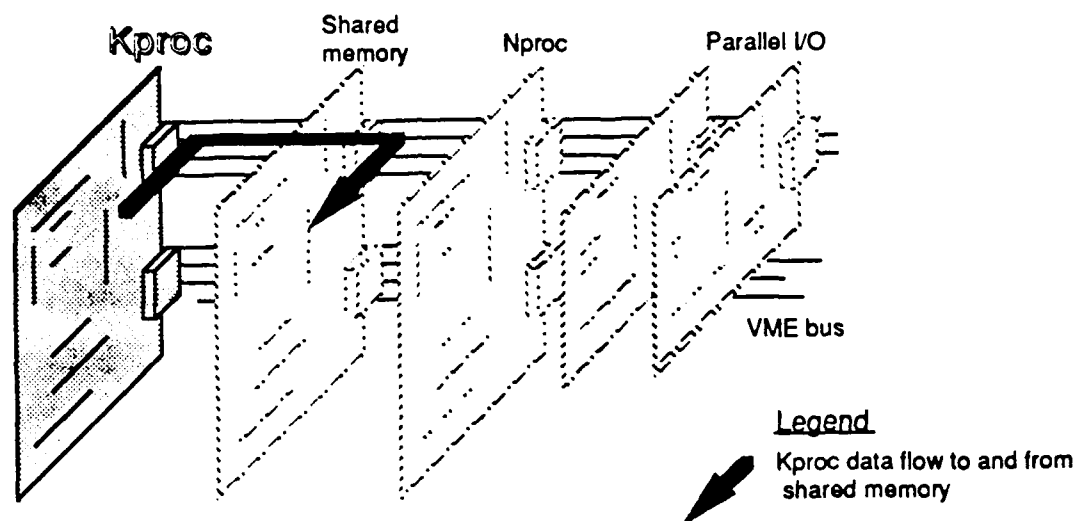


Figure 11: Kernel Processor Hardware

sent or received, the Kproc writes into the shared memory to send, and reads from the shared memory to receive (see Figure 12). The Nproc issues an interrupt, which is recognized and serviced by the Kproc, to indicate that at least one message has been received and has been placed in shared memory. Chapter 6 contains a more detailed discussion of the Kproc-to-Nproc interface.

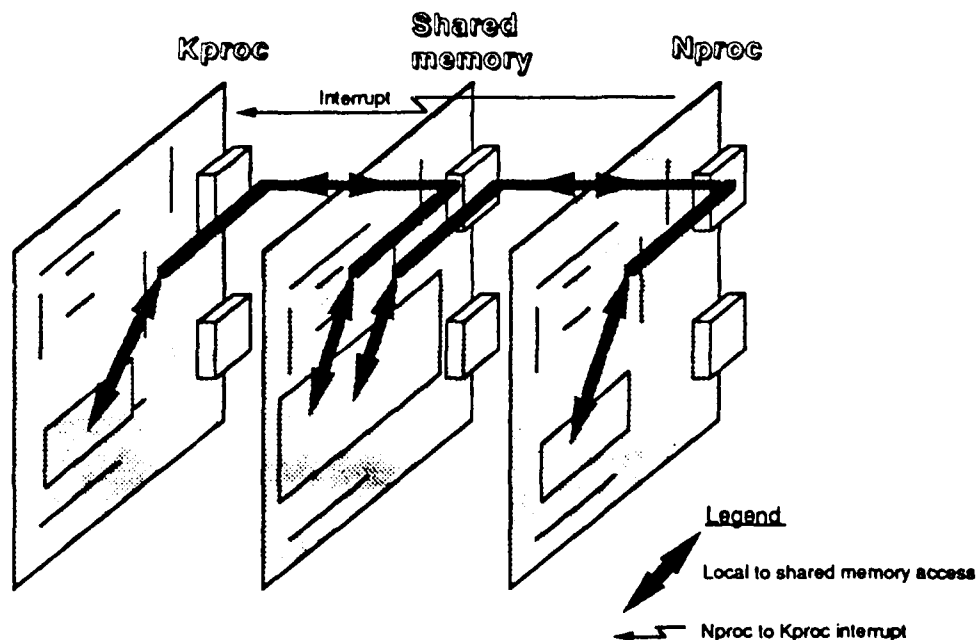


Figure 12: Kproc-to-Nproc Interface

3.3.2. Time Synchronization Bus Interface

The Kernel uses a serial interface connected to all of the Kprocs to implement time synchronization. Port A of each Kproc is connected to this serial interface, called the synchronization (or sync) bus.

Normally, serial interfaces are used in point-to-point communications, such as RS-232C. However, if the line drivers for the serial interface are RS-485/422 compatible, such as those used for Port A, more than one receiver (called a slave) can be serviced by a sender (called the master). This capability is used to essentially "broadcast" the notification to synchronize time across all the nodes.

Initially, the Kernel configures all the Kprocs as slaves. When the application calls the synchronize time primitive, the Kernel asserts Port A of that Kproc as master and sends the new time information to all the other nodes on the sync bus. The interrupt-driven software on the slave Kprocs accept the new time and continue processing.

3.3.3. Non-Kernel Device

The non-Kernel device may be connected to the network and may communicate with other portions of the application using the DARK network protocol. This requires that the non-Kernel device be able to send and receive the 32-bit packets used to communicate via the network. For a non-Kernel device to communicate with a Kernel device requires that the non-Kernel device append to the start of any message the 8-word datagram header (shown in Figure 13).

All of the fields **must** have the values shown in Figure 13, with the following fields specified by the non-Kernel device:

1. Message length: size of the message in bytes
2. From node addr: the network address of the non-Kernel device
3. To node addr: the network address of the Kernel device receiving the message

This is the minimum information needed by the Kernel on the receiving node (to insure the message is correctly routed). The exact format of a datagram is discussed in Part IV.

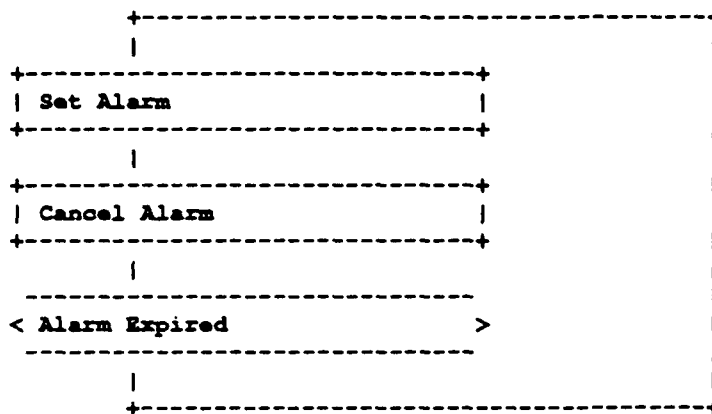
0	15 16	31
message length		0
	0	
	0	
from node address		
to node address		
	0	
	0	
	0	

Figure 13: Non-Kernel Message Header

II. Application Interface

This part defines the user view of the Kernel; the user invokes Kernel operations via calls to this interface. The packages in this section parallel the primitives and requirements described in [KFD 89]. Please refer to the corresponding sections in the [KFD 89] for additional background information.

1. Alarm Management

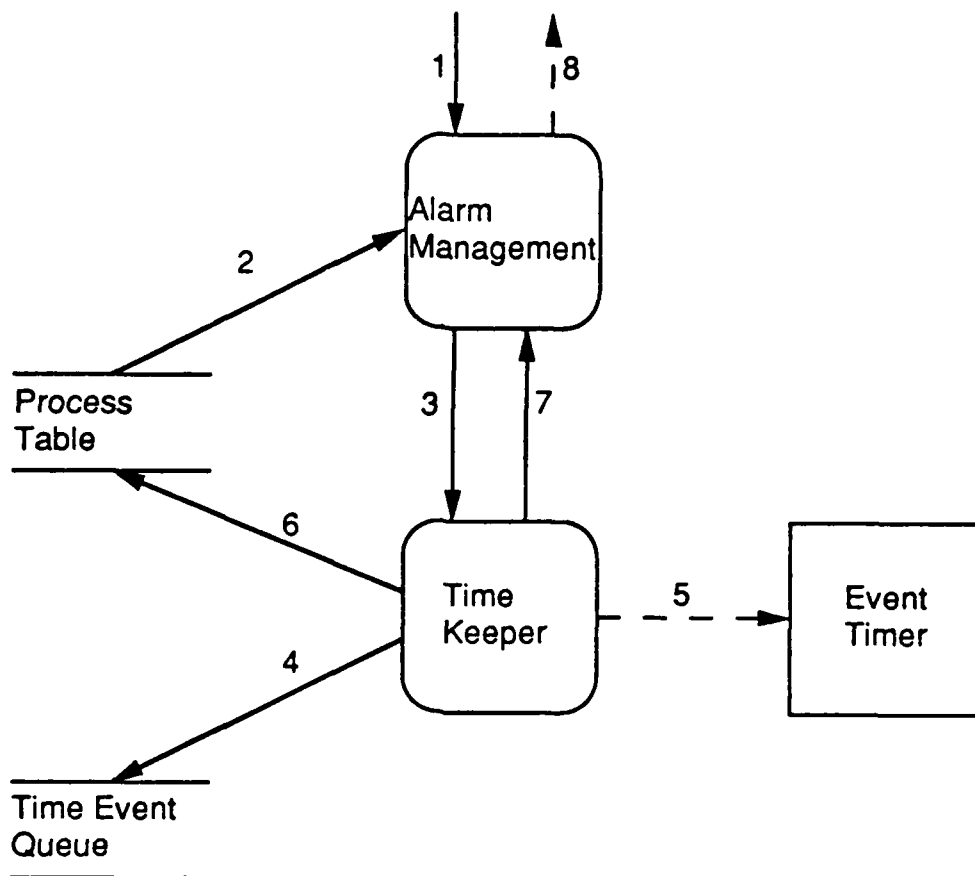


For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 22. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 13.

The functioning of alarms is straightforward. Setting an alarm causes a countdown timer to be initiated: when this timer reaches 0, the *alarm_expired* exception is raised in the setting process. Three situations exist:

1. If the process is executing when the alarm expires, the exception is raised by the Scheduler immediately after the alarm expires.
2. If the process is blocked or suspended when the alarm expires and its alarm resumption priority is higher than that of the current running process, then the Scheduler preempts the current running process and immediately raises the *alarm_expired* exception in the alarmed process.
3. If the process is blocked or suspended when the alarm expires and its alarm resumption priority is lower than that of the current running process, then the Scheduler does not immediately raise the *alarm_expired* exception. Rather, the Scheduler raises the *alarm_expired* exception sometime later, when the process's alarm resumption priority allows it to become eligible to run.

Figure 14 illustrates the key facets of *Set_alarm*. In this scenario, there are no other pending events.



1. The application invokes *set_alarm*.
2. The current alarm state of the process is read from the *process_table* and error checking is performed.
3. An *insert_event* request is made of the *time_keeper*.
4. The *alarm_event* is enqueued in the *time_event_queue*.
5. The event timer is configured for the alarm.
6. The *process table* is updated to reflect the pending alarm event.
7. The alarm event's internal identifier is returned.
8. Control returns to the application.

Figure 14: Set Alarm

1.1. Set Alarm

1.1.1. Interface

```
set_alarm (alarm time
           resumption priority)
```

1.1.2. PDL

```
If called from an interrupt handler then
    raise illegal_context
Else
    If an alarm event is pending then
        Remove the pending alarm event
        Set the pending exception name to resetting_alarm
    End if
    Insert the new alarm event
End if
```

1.2. Cancel Alarm

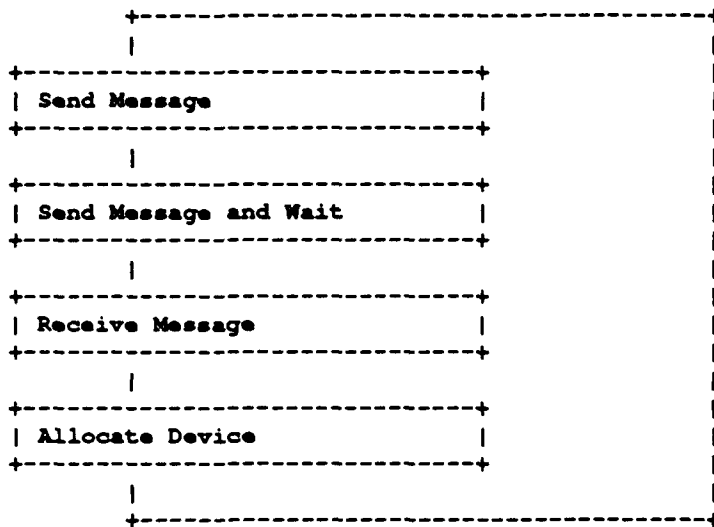
1.2.1. Interface

```
cancel_alarm
```

1.2.2. PDL

```
If called from an interrupt handler then
    raise illegal_context
Else
    If an alarm event is pending then
        Remove the pending alarm event
    Else
        Raise no_alarm_set
    End if
End if
```

2. Communication Management



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 19. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 10. This package contains all of the communication primitives needed to send and receive messages.

The send primitives support acknowledged and unacknowledged sends. The *Send_Message* primitive is non-blocking and is considered an unacknowledged send. It does not generate a response from the receiving Kernel when the receiver receives the message. The *Send_Message_And_Wait* primitive, on the other hand, is considered an acknowledged send and can block. It solicits a response from the receiving Kernel when the message is received by the receiver. As with all blocking kernel operations, a timeout can be specified so the application can control the length of time it is blocked.

The *Receive_Message* primitive blocks if there are no messages in the process' input message queue. Here again, the calling process can specify a timeout to control how long it is blocked. If an acknowledgment is required for a received message then the Kernels take care of it.

Local optimization is performed when the receiving and sending processes are on the same node. When a call to *Send_Message* or *Send_Message_And_Wait* is made the node of the sender and receiver are compared. If the nodes are not the same then the message must be sent over the network to the proper node. However, if the nodes are the same then the network and the associated overhead can be avoided. The amount of processing saved during local optimization will depend on whether the receiving process is waiting or not.

The following discussion deals with the different situations that are taken into consideration during send and receive (message) processing. Except where explicitly stated, all the situations apply equally for remote and local sends.

The actions taken by the communication management to successfully send and receive a message will depend on the form of the send call and what situation the receiving process is in when the message arrives. The following paragraphs describe the different situation that affect the processing of a message.

When a message is sent the receiver is specified. Normally, the receiver will be in one of several possible situations:

1. Receiver is not waiting
2. Receiver is waiting with no messages queued for it
3. Receiver is waiting with at least one message queued for it

The actions that result in each of these cases will depend on whether it is an acknowledged or unacknowledged send. And furthermore, if it is an acknowledged send the actions will depend on whether the specified timeout is:

1. Not specified,
2. Negative and zero, or
3. Greater than zero.

First, consider the state of the receiver when the message arrives. If the receiver's current pending activity is not *Receive_Pending* then it is not waiting at a *Receive_Message* call. On the other hand, if the pending activity is *receive_pending* then it will be either blocked on or unblocked. Waiting and blocked means no other message have been received that would have cause the receiver to be unblocked (changed to suspended). Therefore, the new message would cause the receiver to be unblocked. Keep in mind that even though a receiver is unblocked when a message is received it may or may not resume execution right away depending on its resumption priority relative to the currently running process.

If the receiver is waiting but not blocked then that means it has apparently been unblocked already due to a previously received message and it can be assumed there is at least one message on its msg queue.

In the cases where the receiver is not waiting or it is waiting but there are messages already queued for the receiver, the message can't be copied directly into the receiver's buffer. The copy will have to take place later when the receiver dequeues the datagram from its msg queue. Otherwise, if the receiver is waiting and no other messages are in the queue the message can be copied directly into the receiver's buffer.

Local optimization takes place in varying degrees, depending on the exact situation. Here are two important optimizations that are possible:

1. Copy sender's message directly into the waiting receiver's buffer.
2. Pass a datagram directly from send processing to receive processing.

The specified timeout value in an acknowledged send dictates the response from the receiving Kernel. The calling application specifies a timeout value less than or equal to zero

to be NAKed immediately if the receiver is not ready to receive the message when it arrives. It specifies a timeout value greater than zero to wait for a certain amount of time before being NAKed. And, finally, it does not specify a timeout if it is to wait forever for the receiver to receive the message. Timeout values can be specified as an elapsed or epoch time.

Here is a summary of the send primitive cases, the different forms of the acknowledged and the results:

Desired Result	Send Message form(s)	Timeout Value
Infinite wait	<i>Send_Message_And_Wait(...)</i>	Not applicable
No wait	<i>Send_Message_And_Wait (...Timeout_At => X,...)</i> <i>Send_Message_And_Wait (...Timeout_After => X,...)</i>	$X \leq 0$
Wait until X	<i>Send_Message_And_Wait (...Timeout_After => X,...)</i>	$X > \text{Current time}$
Wait for X	<i>Send_Message_And_Wait (...Timeout_At => X,...)</i>	$X > 0$

2.1. Send message

2.1.1. Invocation

```
send_message (  
    receiving process identifier,  
    message tag,  
    message length,  
    message text)
```

2.1.2. PDL

```
If the tool interface is enabled  
    Log the message attributes for this process  
    Log the message contents for this process  
End if  
  
Check to see according to local information if receiver has been  
    declared and is still alive.  
If receiver is not ok then  
    call scheduler  
Else  
    If receiver is the same node as sender then  
        do local optimization  
    Else  
        Send a process datagram to receiver.  
        Return to caller without blocking  
    Endif  
Endif
```

2.2. Send Message and Wait

During acknowledged sends, a timeout event is set on the receiving node as opposed to the sending node. If the timeout expires, the receiving node's Kernel sends a negative acknowledgment response. Otherwise, it will respond with a positive acknowledgment when the receiver issues a *Receive_Message* and the message is copied into its working space.

2.2.1. Invocation

```
send_message_and_wait (  
    receiving process identifier,  
    message tag,  
    message length,  
    message text,  
    resumption priority of sender)
```

or

```
send_message_and_wait (  
    receiving process identifier,
```

```
message tag,  
message length,  
message text,  
timeout after,  
resumption priority of sender)
```

or

```
procedure send_message_and_wait (  
    receiving process identifier,  
    message tag,  
    message length,  
    message text,  
    timeout at,  
    resumption priority of sender)
```

2.2.2. PDL

```
If the tool interface is enabled  
    Log the message attributes for this process  
    Log the message contents for this process  
End if  
Check to see if receiver has been declared and  
    is still alive and  
    call has not been made from an interrupt handler  
If receiver is not ok then  
    Call scheduler  
End if  
If receiver is local (on the same node as the) sender then  
    Indicate sender has a SEND WITH ACKNOWLEDGMENT PENDING  
    Do local optimization  
Else the receiver is on a remote node  
    Indicate sender has a SEND WITH ACKNOWLEDGMENT PENDING  
    Send a process datagram with the sender's message to the  
        receiver on a remote node  
    Block the sender  
End if
```

2.3. Receive Message

2.3.1. Invocation

```
receive_message (  
    sender of message,  
    message tag,  
    message length,  
    message buffer,  
    buffer size,  
    resumption priority,  
    messages lost)
```

or

```
receive_message (  
    sender of message,  
    message tag,  
    message length,  
    message buffer,  
    buffer size,  
    resumption priority,  
    timeout after,  
    messages lost)
```

or

```
procedure receive_message (  
    sender of message,  
    message tag,  
    message length,  
    message buffer,  
    buffer size,  
    resumption priority,  
    timeout at,  
    messages lost)
```

2.3.2. PDL

Perform error check for:

ILLEGAL CONTEXT FOR CALL

If there is a datagram on the process' msg queue then

If datagram has been optimized then

Do local receive optimization

Call the scheduler

Else datagram is from a remote node

Do copy action that includes copying message from datagram
to receiver's buffer, and clearing any timeout event
and sending an acknowledgment for acknowledged sends

Call scheduler

End if

Else no datagram is available

Store the receiver's buffer address and size in the process
table in case the sender uses local optimization

If a receiver timeout is specified then

Set the timeout type

Try to set a timeout event

Perform error check for no timeout event set

End if

Loop

If there is no datagram available on the process' msg queue then

Set the receiver's pending activity to RECEIVE PENDING

Call the Scheduler to Block the receiver

Else a datagram is available

If the tool interface is enabled

```

        Log the message attributes for this process
        Log the message contents for this process
    End if
    If datagram has been optimized then
        Do local receive optimization
    Else datagram is from a remote node
        Do copy action that includes copying message from datagram
        to receiver's buffer, and clearing any timeout event and
        sending an acknowledgment for acknowledged sends
    End if
    Exit
End if
End loop
Call Scheduler

```

2.4. Allocate Device

2.4.1. Invocation

```

allocate_device_receiver (
    pid of process to receive the non-Kernel device messages,
    device id of non-Kernel device originating the messages)

```

2.4.2. PDL

```

If the indicated device is a Kernel device then
    raise no such device
Else
    Log the receiver in the NCT for the non-Kernel device
    If this allocation is replacing a previous allocation then
        raise replacing old allocation
    End if
End if

```

2.5. Copy Message

2.5.1. Invocation

```

copy_message (
    datagram to copy,
    sender of the datagram,
    message tag,
    message length,
    message buffer,
    buffer size,
    messages lost)

```


2.5.2. PDL

```
Set out parameter from datagram header information
Perform error checks for:
    RECEIVER BUFFER TOO SMALL
Copy message from datagram to buffer to receiver's buffer
Clear receiver's overflow status
If acknowledge send and sender has a SEND WITH ACK PENDING then
    Send a kernel datagram to acknowledge sender
    Clear any pending activity for the sender
End if
Delete datagram from receiver's msg queue
Free up the datagram
Clear any pending activity for the receiver
```

2.6. Do_Local_Send_Optimization

2.6.1. Invocation

```
do_local_send_optimization(
    receiver of message,
    operation,
    message tag,
    message length,
    message text,
    timeout,
    resumption priority)
```

2.6.2. PDL

```
If receiver is waiting and blocked then
    Allocate an empty datagram
    If no is datagram available then
        If message is an acknowledged send then
            Set MESSAGE_NOT_RECEIVED exception
        End if
        If not called from an interrupt handler then
            Call Scheduler
        Else
            return
        End if
    Else a datagram is available
        Set the datagram header fields from input parameters
        Indicate LOCAL OPTIMIZATION is taking place
        Indicate the sender's message has already been copied to
            the receiver's buffer
        Perform error check on receiver's buffer size
        Copy from sender's buffer to receiver's buffer
        Clear any queue overflow status
        Clear any pending activity for the sender
```

```

    Enqueue the datagram
    Call Scheduler to unblock the receiver
    If not called from an interrupt handler then
        Call Scheduler
    Else it was called from interrupt handler
        return
    End if
End if
Else the the receiver is not waiting and blocked then
    Perform message queue overflow error check
    If timeout is less than zero or call is a Send_Message then
        Allocate an empty datagram
        If no datagram is available then
            If Send_Message_And_Wait was issued then
                Clear sender's pending activity
                Set MESSAGE_NOT_RECEIVED exception for the sender
                Call Scheduler
            Else
                Call Scheduler
            End if
        Else a datagram is available
            Set fields of the datagram header with input parameters
            Indicate LOCAL OPTIMIZATION is taking place
            If Send_Message_And_Wait was issued then
                Indicate receiver needs to copy the message from
                    sender's buffer to its buffer
                Set the sender's buffer address in the datagram for
                    receive message processing
                Enqueue the datagram on the receiver's msg queue
                Indicate the sender's pending activity as a SEND
                    WITH ACKNOWLEDGMENT PENDING
                Call Scheduler to block the sender
            Else send issued with a Send_Message
                Indicate receiver needs to copy the message from the
                    datagram's buffer
                Copy message from sender's buffer into datagram's
                    buffer
                Enqueue the datagram on the receiver's msg queue
            End if
        End if
    End if
Elsif the timeout is equal to zero then
    Clear any pending activity for the sender
    Set the MESSAGE_NOT_RECEIVED exception for the sender
    Call Scheduler with the resumption priority
Else the timeout is greater than zero
    Allocate a zero length datagram
    If no datagram is available then
        Clear any pending activity
        Set the MESSAGE_NOT_RECEIVED for the sender
        Call Scheduler with the resumption priority
    Else a datagram is available

```

```

Set datagram header fields with input parameters
Indicate LOCAL OPTIMIZATION is taking place
Indicate the receiver needs to copy the message from
the sender's buffer
Set the sender's buffer address in the datagram for
receive message processing
Try to set a timeout event
If the timeout has not expired (event set ok) then
    Save the queue name and datagram pointer so the
    datagram can be removed if the timeout expires
    Set the sender's pending activity to
        SEND WITH ACKNOWLEDGMENT PENDING
    Enqueue the datagram on the receiver's msg queue
    Call Scheduler to block the sender
Else the timeout has expired (no event set)
    Clear any pending activity for the sender
    Set the MESSAGE_NOT_RECEIVED exception for the sender
    Free up the datagram
    Call Scheduler with the resumption priority
End if
End if
End if
End if

```

2.7. Do_Local_Rcv_Optimization

2.7.1. Invocation

```

do_local_rcv_optimization(
    datagram to receive,
    sender of message,
    message tag,
    message length,
    message buffer,
    buffer size,
    resumption priority,
    timeout value,
    has timeout,
    messages_lost)

```

2.7.2. PDL

```

Set the out parameters, such as message tag, message length, and
sender from the appropriate datagram header fields
Perform error checks for BUFFER TOO SMALL FOR MESSAGE
If COPY FROM THE SENDER'S BUFFER is required then
    Using the sender buffer's address passed in the datagram header
    Copy message from sender's buffer to receiver's buffer
Elsif COPY FROM THE DATAGRAM'S BUFFER is required then
    Copy message from datagram buffer to receiver's buffer

```

Else NO COPY IS NECESSARY

Do nothing

End if

If this is an acknowledged send and the sender has a SEND WITH
ACKNOWLEDGMENT PENDING and the MESSAGE ID match then

Acknowledge sender by clearing any pending activity for sender

Call Scheduler

Else no need to acknowledge

Do nothing

End if

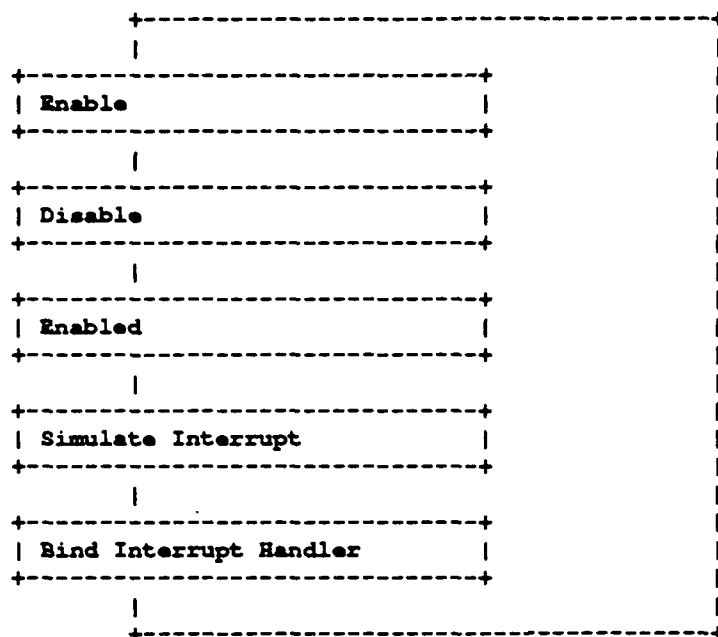
Delete the datagram from the receiver's msg queue

Free up the datagram

3. Hardware Interface

This package encapsulates the basic Ada types in a form that is more transportable. Since this package is compiler and hardware dependent, the issues involved in this encapsulation are discussed in Part IX with this package shown in Appendix E.

4. Interrupt Management



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 20. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 11.

Note: in all cases, an "illegal interrupt" raises a `CONSTRAINT_ERROR`, since the parameter (sub) type domain is the domain of legal interrupts. Thus, there is no explicit check for the legality of an interrupt.

4.1. Enable

4.1.1. Interface

Enable (interrupt name)

4.1.2. PDL

```
If the interrupt_owner in the interrupt table is not application then
    Raise illegal_interrupt
End if
If handler state in interrupt table is not bound then
    Raise illegal_interrupt
End if
Set interrupt state in the interrupt table to enabled
Set interrupt's entry in the Kernel interrupt vector to point to
    the user-supplied interrupt handler
```

4.2. Disable

4.2.1. Interface

Disable (interrupt name)

4.2.2. PDL

```
If the interrupt_owner in the interrupt table is not application then
    Raise illegal_interrupt
End if
Set interrupt state in the interrupt table to disabled
Set interrupt's entry in the Kernel interrupt vector to point to
    the null interrupt handler
```

4.3. Enabled

4.3.1. Interface

Enabled (interrupt name)
return boolean

4.3.2. PDL

```
If the interrupt_owner in the interrupt table is not application then
    Raise illegal_interrupt
End if
Return the interrupt_state field of the interrupt table
```

4.4. Simulate Interrupt

4.4.1. Interface

`Simulate_interrupt (interrupt name)`

4.4.2. PDL

```
If the interrupt_owner in the interrupt table is not application then
    Raise illegal_interrupt
End if
If the interrupt_state in the interrupt table is not bound then
    Raise illegal_interrupt
End if

Set interrupt_source in the interrupt table to internal
Increment interrupt_nesting level
Begin atomic
    Perform an indirect call of the interrupt handler pointed
    to by the interrupt's entry in the interrupt table

    Exception
        when others =>
            'handle all exceptions, taking no action (to simulate the
            effect of an unhandled exception in a real interrupt
            processing)
End atomic

Decrement the interrupt_nesting level
Set the interrupt source to be External
If the interrupt can preempt then
    Schedule
Else
    return to the caller
End if
```

4.5. Bind Interrupt Handler

4.5.1. Interface

`bind_interrupt_handler (interrupt name,`
 `address of interrupt handler procedure,`
 `interrupt can cause process preemption)`

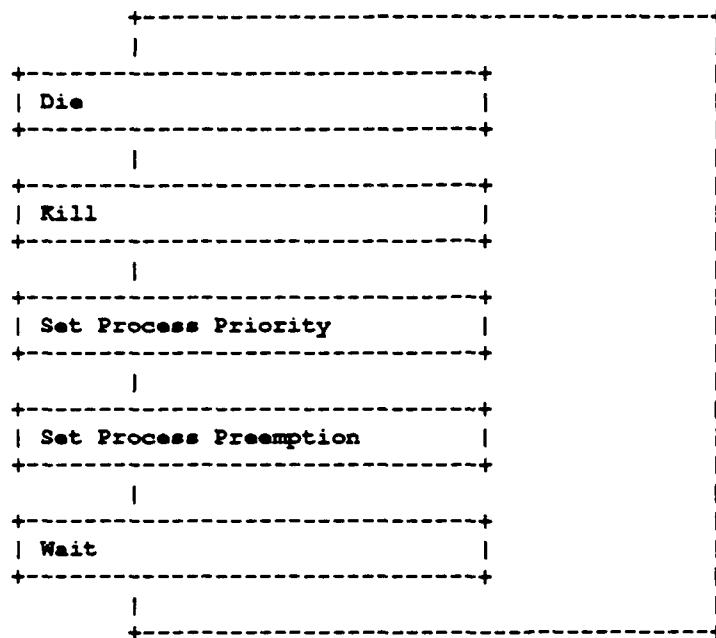
4.5.2. PDL

```
If the interrupt_owner in the interrupt table is not application then
  Raise illegal_interrupt
End if

If interrupt handler is already bound =>
  Store new handler address in the interrupt table
  Store new value for can_preempt in the interrupt table
  Raise re-binding interrupt handler
Else
  Insert interrupt name into interrupt table
  Store handler address in the interrupt table
  Store value for can_preempt in the interrupt table
  Set handler state to bound in the interrupt table

  If the interrupt can_preempt then
    Call machine-dependent routine to bind a "slow" interrupt
  Else
    Call machine-dependent routine to bind a "fast" interrupt
  End if
End if
```

5. Process Attribute Modifiers



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 18. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 9.

5.1. Die

5.1.1. Interface

Die

5.1.2. PDL

```
If called from an interrupt handler then
  Raise illegal_context
End if
Purge_message_queue
Release all claimed semaphores
Deallocate any non-Kernel devices assigned to this process
Schedule (new state => dead)
```

5.2. Kill

For a remote process kill, a special Kernel-to Kernel message is formatted and transmitted to the node hosting the process to kill. See Appendix B, Table 1, for the exact format of this message.

5.2.1. Interface

Kill (process to kill)

5.2.2. PDL

```
If the process to kill is not dead then
  If the process to kill is remote then
    Send_kernel_datagram ("kill_message")
  Else
    Remove_process (process to kill) from the scheduler
    Purge_message_queue
    Release all claimed semaphores
    Deallocate any non-Kernel devices assigned to this process
    If the process to kill is the current_running_process then
      Schedule (new state => dead)
    Else
      If the tool interface is enabled
        Log the process attributes for process being killed
      End if
      Remove_process (pid    => process to kill,
                     new_state => dead)
    End if
  End if
Else
  null...the process to kill is already dead
  and there is nothing to do
```

End if

5.3. Set Process Priority

Changing the priority of a process has been optimized to account for the following situations:

1. Setting the priority to its current value is a null operation.
2. Raising the priority of a process does not affect its eligibility to run; it remains the current_running_process.
3. Lowering the priority of a process may cause it to be descheduled.

5.3.1. Interface

`set_process_priority (new process priority)`

5.3.2. PDL

```
If called from an interrupt handler then
  Raise illegal_context
End if
```

```
If there is no change in priority then
  return
```

```
Else
  Schedule (new priority => new process priority)
End if
```

5.4. Set_process_preemption

Changing the preemption of a process has been optimized to account for the following situations:

1. Setting preemption to its current value is a null operation.
2. Setting preemption to disabled does not reschedule the process; it removes the current slice event, so the timeslice doesn't expire.
3. Setting preemption to enabled simply calls the Scheduler (which selects the highest priority process – if it is the invoking process, it automatically inserts a slice event). Consequently, setting process preemption to enabled may cause the calling process to be descheduled.

5.4.1. Interface

`Set_process_preemption (new process preemption)`

5.4.2. PDL

```
If called from an interrupt handler then
  Raise illegal_context
End if

If there is no change in preemption then
  return
Else if the current preemption is enabled and
        the new preemption is disabled then
  If time slicing is enabled then
    Remove_event (timeslice)
  End if
  Set the preemption to disabled
Else...the current preemption is disabled and
        the new preemption is enabled
  Schedule (new_preemption => enable)
End if
```

5.5. Wait

This primitive has been constructed to have the following semantics:

1. All calls to *wait* cause the Scheduler to be invoked.
2. If the wait is for a future time, the process blocks until that time arrives.
3. If the wait is for a non-future time, no wait is done, but the process relinquishes control of the processor and may be descheduled.²

5.5.1. Interface

```
Wait (until epoch time,
      resumption priority)
```

or

```
Wait (for elapsed time,
      resumption priority)
```

5.5.2. PDL

```
If called from an interrupt handler then
  Raise illegal_context
End if
Insert_event (wait_timeout)
If the_event_identifier is null then (waiting for a non-future time)
  If there is no change in priority then
```

²Thus, if two processes of equal priority need to execute in a co-routine like paradigm, for example, a *wait* with a time of 0 may be used to switch between the co-routine processes, see Appendix E of [KUM 89].

```

        Remove_process (current_running_process)
    end if
    Schedule (new_priority => resumption priority)
Else...the wait is for a future time
    Schedule (new_priority => resumption priority,
              new_state => blocked)
End if

```

5.6. Purge Message Queue

5.6.1. Interface

```

purge_message_queue (process identifier)

```

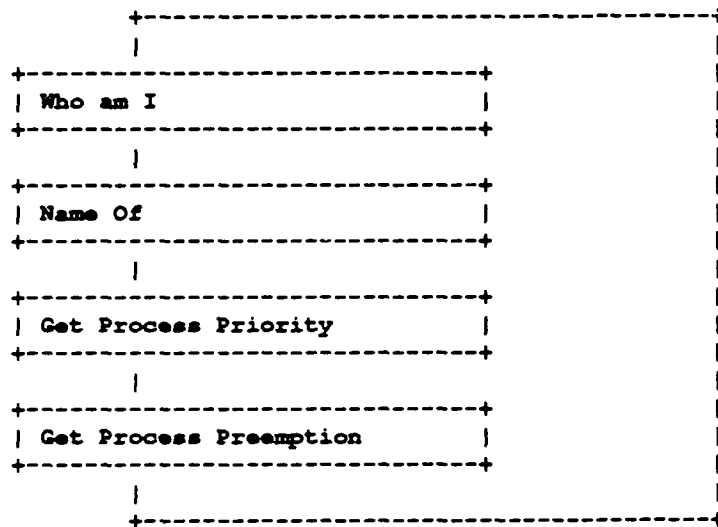
5.6.2. PDL

```

Locate the message queue of the process being killed
Cap it so that the Kernel immediately rejects
all future incoming messages
For each message in the message queue
    If the message requires an acknowledgment then
        Remove the timeout event associated with the message
        Send_kernel_datagram ("nak") to the message originator
    End if
    Delete the message from the message queue
    Free the datagram holding the message
End loop
Free the current send buffer

```

6. Process Attribute Readers



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 18. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 9.

6.1. Who am I

6.1.1. Interface

`who_am_i return process_identifier`

6.1.2. PDL

`If called from an interrupt handler then
 Raise illegal_context_for_call
End if
Return current_running_process`

6.2. Name Of

6.2.1. Interface

`name_of (process identifier)
 return process logical name...as a string`

6.2.2. PDL

`Return the logical_name field of the specified process
from the process table`

6.3. Get Process Priority

6.3.1. Interface

`get_process_priority return priority`

6.3.2. PDL

`If called from an interrupt handler then
 Raise illegal_context_for_call
End if
Return priority of the current_running_process`

6.4. Get Process Preemption

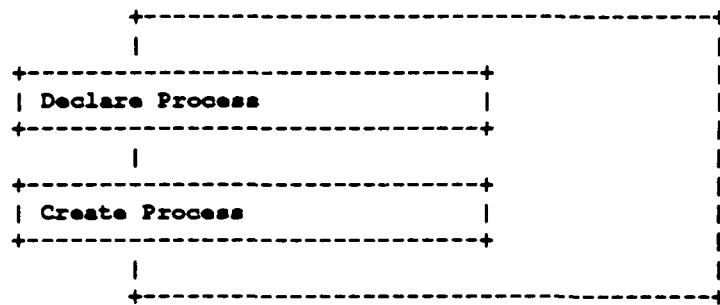
6.4.1. Interface

`get_process_preemption return preemption state`

6.4.2. PDL

```
If called from an interrupt handler then
    Raise illegal_context
End if
Return preemption status of the current_running_process
```

7. Process Managers



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 16. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 7.

7.1. Declare Process: for Kernel Process

7.1.1. Interface

```
Declare_process (process logical name)
    return process identifier
```

7.1.2. PDL

```
If calling unit is not the Main Unit then
    Raise calling_unit_not_main_unit
End if
Declare_process_real_work
```

7.2. Declare Process: for non-Kernel device

7.2.1. Interface

```
Declare_process (device logical name)
    return process identifier
```

7.2.2. PDL

```
If calling unit is not the Main Unit then
    Raise calling_unit_not_main_unit
End if

If the device logical name is not in the NCT then
    Raise unknown_non_kernel_device
End if

Declare_process_real_work
```

7.3. Create Process

The act of creating a process broadcasts a special Kernel-to-Kernel message. The exact format of this message is shown in Appendix B, Table 1.

7.3.1. Interface

```
Create_process (process identifier,
    address of the process code,
    stack size for process local data,
    message queue size as number of messages,
    how to handle message queue overflow,
    initial priority,
    initial preemption)
```

7.3.2. PDL

```
If the process identifier is null then
    Raise illegal_process_identifier
End if

If calling unit is not the Main Unit then
    Raise calling_unit_not_main_unit
End if

If process identifier is not in the process table then
    Raise illegal_process_identifier
End if

If process being created was declared as a non-Kernel device then
    Raise no_kernel_process_on_non_kernel_device
End if

If process has already been created then
    Raise process_already_created
End if

If the address of the process code is illegal then
    Raise illegal_process_address
End if

Generate the process_index
Allocate the stack space
Set the stack_low_address field in the process table
Set the stack_high_address field in the process table
Set the priority field in the process table
Set the preemption field in the process table
Set the message_queue_size field in the process table
Set the queue_overwrite_rule in the process table
Set the message_queue field in the process table to a new,

Create the call frame via process_encapsulation
Insert the process into the Scheduler
Set the locally_created field in the process table
Broadcast the process_created message
```

7.4. Declare_process_real_work

The PDL below embodies the following design issues:

1. All stack address are longword (i.e., 32-bit) aligned due to hardware constraints.
2. The stack must be allocated before the initial call frame may be created herein.

7.4.1. Interface

```
declare_process_real_work (process logical name,  
                           kind of process)  
    return process identifier
```

7.4.2. PDL

```
If the process logical name is already in use then  
    Raise process_already_exists  
End if  
Create a new process table entry  
If there is insufficient space to do this then  
    Raise insufficient_space  
End if  
  
Log the process logical name in the process table entry  
Log the kind of process in the process table entry  
Mark the process as declared  
Enqueue the new process table entry in the process table
```

7.5. Null_procedure

This procedure is used by create process if a user error is detected and a clean recovery is not possible. In such a case, Create_process creates a truly void process.

This procedure is never called; it does nothing.

7.5.1. Interface

N/A

7.5.2. PDL

Null

7.6. Calling_unit_is_main_unit

The PDL below embodies the design issues:

1. During processor/process initialization, the Main Unit must be the null_process.

7.6.1. Interface

`calling_unit_is_main_unit` return boolean

7.6.2. PDL

Return indication whether or not the `current_running_process` is the `null_process`

7.7. Is_illegal

This function tests for an illegal address for process code. As this may be highly application specific, a simple default is currently implemented: the function accepts anything as a legal address.

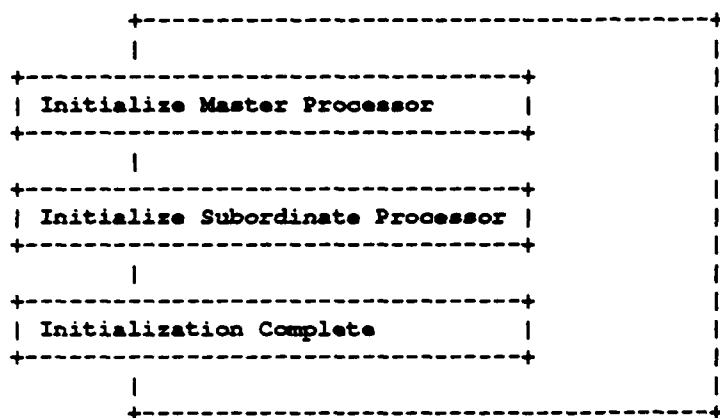
7.7.1. Interface

`is_illegal` (address to test)
return boolean

7.7.2. PDL

return false

8. Processor Management



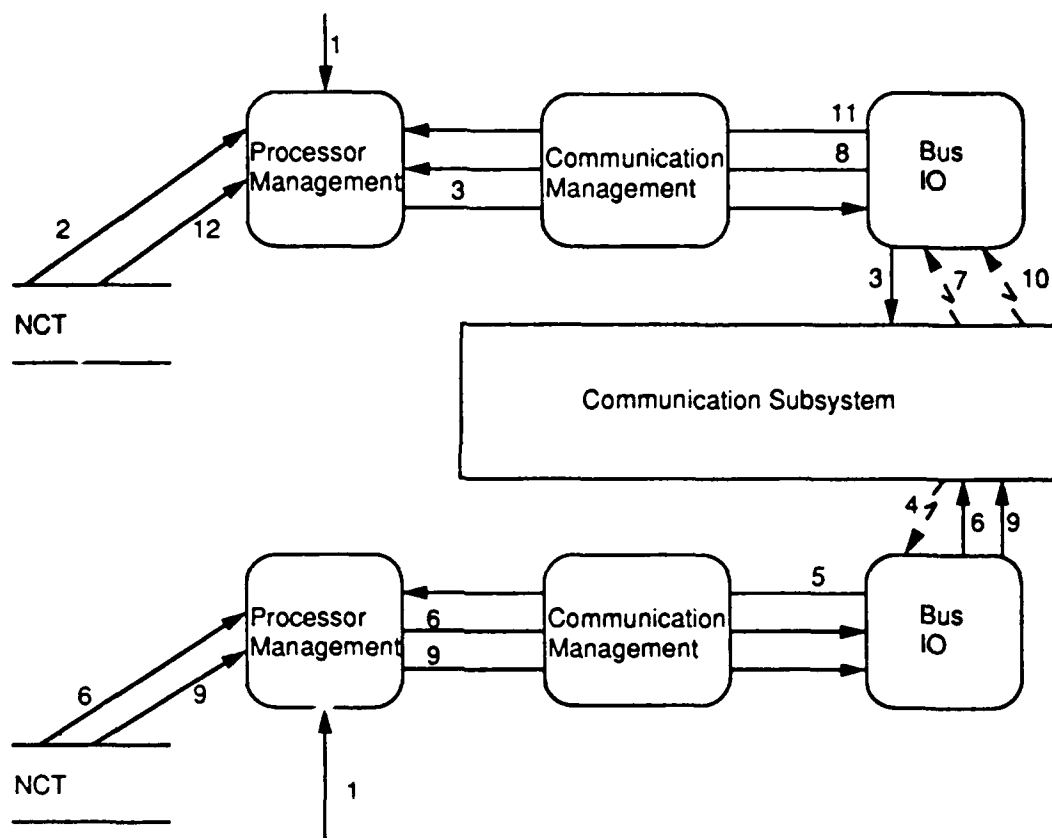
For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapters 15 and 16. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapters 6 and 7.

The implementation of processor or network initialization requires that a number of messages be exchanged among the nodes. The exact format of each message is described in Appendix B, Table 3.

The initialization protocol occurs in two phases:

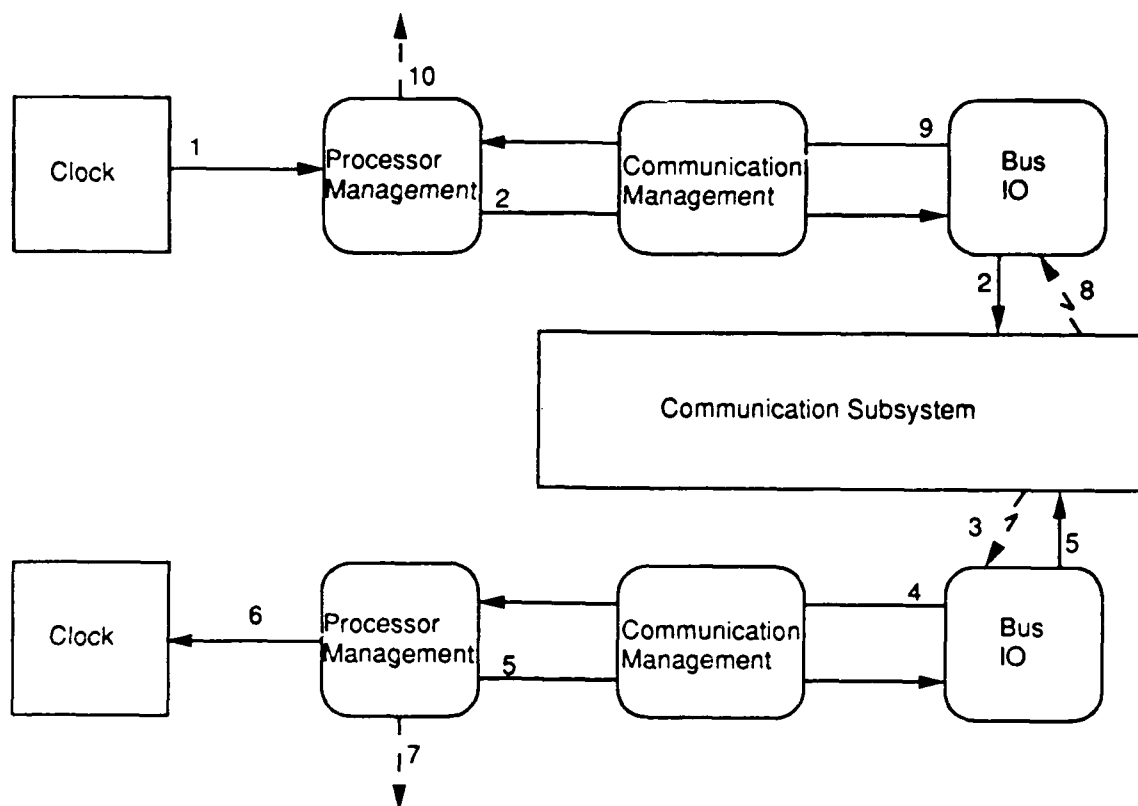
- Phase 1: where the Master processor verifies the connectivity of the network. This is shown in Figure 15.
- Phase 2: where the Master processor synchronizes all the Kernel clocks and commences process creation. This is shown in Figure 16.

These figures illustrate the protocol as it works for a simple two-node network.



1. *Initialize_subordinate_processor* call is issued. *Initialize_master_processor* call is issued.
2. The NCT is read and the initialization order of the network is determined.
3. The *master_ready* message is sent to the subordinate via *communication_management* and *bus_io* over the communication subsystem.
4. A message interrupt arrives at the subordinate.
5. The *master_ready* message is delivered to the subordinate via *bus_io* and {*communication_management*}.
6. The subordinate responds by determining the size of the NCT, formatting an *nct_size* message and sending it to the Master.
7. A message interrupt arrives at the Master.
8. The *nct_size* message is delivered to the Master.
9. The subordinate continues its response by reading the NCT, formatting an *nct_entry* message for each row of the NCT, and sending each message to the Master.
10. A message interrupt for each *nct_entry* message arrives at the Master.
11. Each *NCT_entry* message is delivered to the Master.
12. The Master reads its copy of the NCT and verifies the validity of the subordinate's NCT.

Figure 15: Network Initialization Protocol: Phase 1



1. The Master reads the current time of day.
2. The Master sends the go message to the subordinate.
3. An interrupt arrives at the subordinate.
4. The go message is delivered to the subordinate.
5. The subordinate sends a go_acknowledge message to the Master.
6. The subordinate resets its local clock to be synchronized with the Master's local clock.
7. Control returns to the subordinate's Main Unit.
8. An interrupt arrives at the Master.
9. The go_acknowledgment is delivered to the Master.
10. Control returns to the Master's Main Unit.

Figure 16: Network Initialization Protocol: Phase 2

8.1. Initialize_master_processor

8.1.1. Interface

```
initialize_master_processor (current time of day
                             initialization timeout)
```

8.1.2. PDL

```
Phase 0.....
  Check that calling unit is main unit
  If the data structures have not yet been initialized then
    Initialize the real time clock
    Initialize the process table
    Initialize the scheduler
    Initialize datagram management
    Initialize bus io
    Initialize the time keeper
    Indicate that the data structures are now initialized
  End if
  Determine_order

Phase 1.....
  For each node in the initialization order loop
    Lookup the process index for the main unit on the subordinate
    Insert a timeout event
    Send master_ready message to the subordinate
    Receive_nct
    Remove the timeout event
    If the subordinate is needed to run then
      If the subordinate's NCT matches the Master's NCT then
        The subordinate has successfully completed phase 1
      Else
        The subordinate has failed to complete phase 1
        Broadcast a network failure message
        Raise network_failure
      End if
    Else
      If the subordinate's NCT matches the Master's NCT then
        The subordinate has successfully completed phase 1
      Else
        The subordinate has failed to complete phase 1
      End if
    End if

  Exception
    When the timeout expires =>
      The subordinate has failed to complete phase 1
      If the subordinate is needed to run then
        Broadcast a network failure message
        Raise an exception
      End if
    When some other exception occurs =>
      The subordinate has failed to complete phase 1
      If the subordinate is needed to run then
```

```

        Broadcast a network failure message
        Raise network_failure
    End if
End loop

Phase 2.....
For each node in the initialization order loop
    If the subordinate completed phase 1 then
        Lookup a process index for the main unit on the subordinate
        Insert a timeout event
        Format a go message
        Send go_enclosed message to the subordinate
        Wait for subordinate to respond
        Remove the timeout event
        If the subordinate is needed to run then
            If the response is a go_acknowledgment then
                The subordinate has successfully completed phase 2
            Else
                The subordinate has has failed to complete phase 2
                Broadcast a network failure message
                Raise network_failure
            End if
        Else
            If the response is a go_acknowledgment then
                The subordinate has successfully completed phase 2
            Else
                The subordinate has has failed to complete phase 2
            End if
        End if
    End if

Exception
    When the timeout expires =>
        The subordinate has has failed to complete phase 2
        If the subordinate is needed to run then
            Broadcast a network failure message
            Raise an exception
        End if
    When some other exception occurs =>
        The subordinate has has failed to complete phase 2
        If the subordinate is needed to run then
            Broadcast a network failure message
            Raise network_failure
        End if
    End loop
When some other exception occurs =>
    If the subordinate is needed to run then
        Broadcast a network failure message
        Raise network_failure
    End if

```

8.2. Initialize_subordinate_processor

8.2.1. Interface

`initialize_subordinate_processor (initialization timeout)`

8.2.2. PDL

```
Phase 0.....
  Check that calling unit is main unit
  If the data structures have not yet been initialized then
    Initialize the real time clock
    Initialize the process table
    Initialize the scheduler
    Initialize datagram management
    Initialize bus io
    Initialize the time keeper
    Indicate that the data structures are now initialized
  End if

Phase 1.....
  Insert a timeout event
  Wait for the Master to send a master_ready message
  Remove the timeout event
  If the message is a master_ready message then
    Send_nct
  Else
    Broadcast a network failure message
    Raise network_failure
  End if

Exception
  When the timeout expires =>
    Broadcast a network failure message
    Raise an exception
  When some other exception occurs =>
    Broadcast a network failure message
    Raise network_failure

Phase 2.....
  Insert a timeout event
  Wait for the Master to send a go_enclosed message
  Remove the timeout event
  If the message is a go_enclosed message then
    Send the a go_acknowledge message to the Master
    Reset the local epoch time
  Else
    Broadcast a network failure message
    Raise network_failure
  End if

Exception
  When the timeout expires =>
    Broadcast a network failure message
    Raise an exception
```

Exception

When some other exception occurs =>
Broadcast a network failure message
Raise network_failure

8.3. Initialization_complete

8.3.1. Interface

initialization_complete (final initialization timeout)

8.3.2. PDL

Check that the calling unit is the Main Unit
Insert a timeout event
Broadcast the init_complete message

Phase 1.....

Determine if all the nodes have broadcast their init_complete messages,
this is done by looping thru all the nodes in the NCT while
their local initialization complete flags are "and"ed together
If the result is false, then the loop is repeated (since the
initialization complete messages arrive asynchronously)
If the result is true, then
all the nodes call have issued a call to initialization_complete
and continue by pruning the process table of unneeded entries

Phase 2.....

Initialize the process table iterator
While there are more entries to process loop
Get the next process table entry
If the process is declared then
If the process was (remotely created and locally created) or
not created at all then
Broadcast network failure
Raise network_failure
Else
The process entry is correct
Else...it was remotely created by never locally declared
Clear the entry in the process mapping table
Purge the entry from the process table
End loop

If the resulting process table is too large then
Broadcast network failure
Raise network_failure
End if

Log initialization complete
Remove the timeout event
Schedule the first eligible application process

Exception

When the timeout expires =>
Broadcast network failure
Raise an exception

8.4. Determine_order

8.4.1. Interface

determine_order

8.4.2. PDL

```
For each node entry in the NCT loop
  If the node is a kernel device and the initialization order is set then
    Increment the participating node count
    Log the node in the initialization order
  End if
End loop

If the participating node count is zero (indicating no explicit
initialization order was specified by the user) then
  For each node entry in the NCT loop
    If the node is a kernel device then
      Increment the participating node count
      Log the node in the initialization order
    End if
  End loop
End if
Log the participating node count
```

8.5. Receive_nct

8.5.1. Interface

receive_nct (expected subordinate
received subordinate nct)

8.5.2. PDL

```
Wait for a message from the subordinate
If the message is not an nct_count or
the wrong subordinate sent the message then
  Raise an exception
End if

Once the count is successfully received then loop until
that number of NCT entries is received
  Wait for a message from the subordinate
  If the message is not an nct_entry or
  the wrong subordinate sent the message then
```

```
        Raise an exception
    End if
End loop

Return the received NCT to the Master
```

8.6. Send_nct

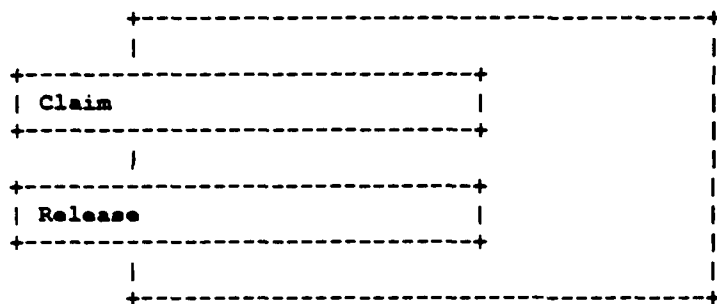
8.6.1. Interface

```
send_nct (master processor)
```

8.6.2. PDL

```
Send the size (in entries) of the NCT to the master processor
For each entry in the NCT loop
    Send the NCT entry to the Master processor
End loop
```

9. Semaphore Management



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 17. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 8.

9.1. Claim

9.1.1. Interface

Claim (semaphore name,
resumption priority)

or

Claim (semaphore name,
elapsed timeout,
resumption priority)

or

Claim (semaphore name,
epoch timeout,
resumption priority)

9.1.2. PDL

```
If called from an interrupt handler then
  Raise illegal_context
Else if semaphore already claimed by this process then
  Raise illegal_context
End if

If queue depth = -1 then...the semaphore is available
  Set semaphore's wait queue depth to 0
  Schedule (priority => resumption priority)
Else
  Set semaphore pending field in Process Table
  Increment the semaphore's wait queue depth by 1
  Enqueue process in the semaphore's wait queue
  If a timeout was specified then
    Insert_event (semaphore timeout)
  End if
  Schedule (priority => resumption priority,
    state => blocked)
End if
```

9.2. Release

9.2.1. Interface

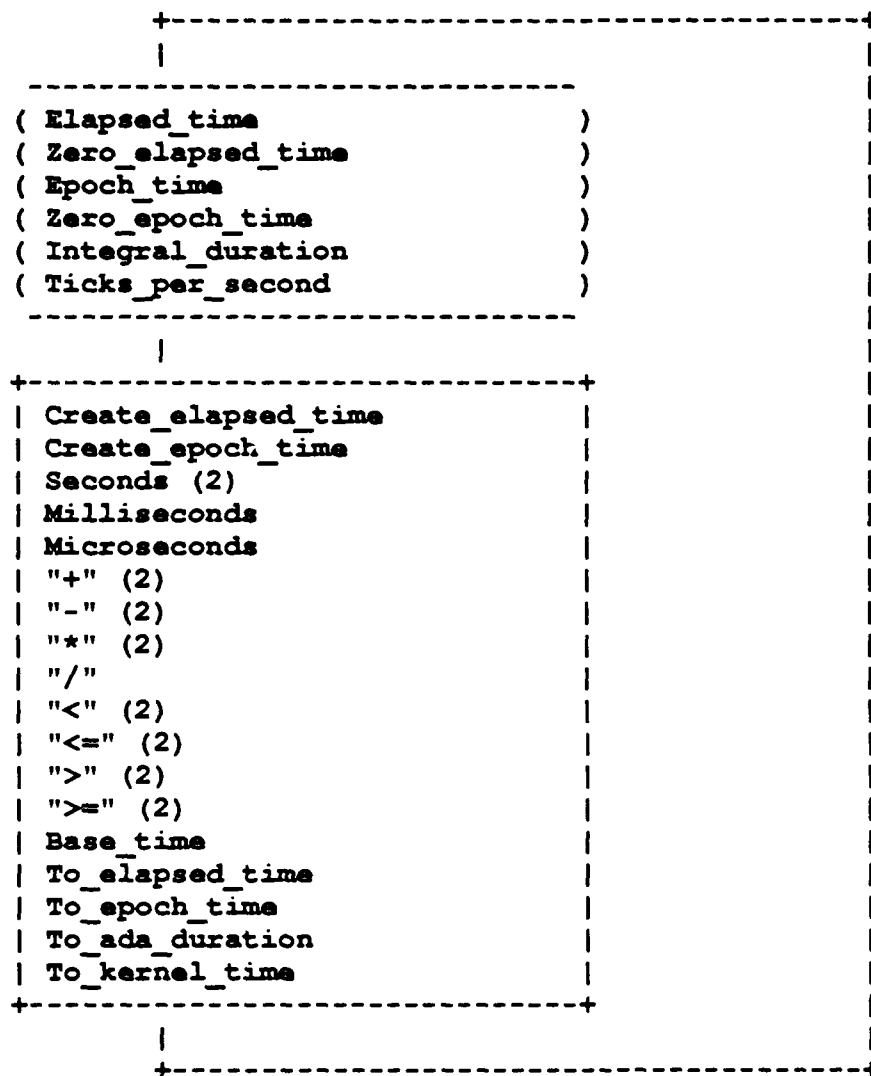
Release (semaphore name)

9.2.2. PDL

```
If called from an interrupt handler then
  Raise illegal_context
Else if semaphore not claimed by this process then
  Raise not_my_semaphore
End if

If the semaphore's wait queue depth = 0 => no process is waiting
  Set semaphore's wait queue depth to -1
  Set wait queue head to null
Else
  Decrement queue depth by 1
  Dequeue process from the semaphore's wait queue
  If a timeout is pending then
    Remove_event (semaphore timeout)
  End if
  Clear semaphore pending field of the dequeued process
  in Process Table
  Insert Process(dequeued process) into Scheduler
  Schedule
End if
```

10. Time Globals



10.1. Create_elapsed_time

10.1.1. Interface

Create_elapsed_time (day, second) return elapsed_time

10.1.2. PDL

Multiply day by kernel_time for 1 day (86400 seconds)
Convert second to kernel_time
Add to result of multiplication
Convert result from kernel_time to elapsed_time
Return elapsed_time value

10.2. Create_elapsed_time

10.2.1. Interface

Create_elapsed_time (day, second) return epoch_time

10.2.2. PDL

Multiply day by kernel_time for 1 day (86400 seconds)
Convert second to kernel_time
Add to result of multiplication
Convert result from kernel_time to elapsed_time
Return epoch_time value

10.3. Seconds

10.3.1. Interface

Seconds(an Ada duration) return elapsed_time

or

Seconds(an integral duration) return elapsed_time

10.3.2. PDL

Convert argument to kernel_time
Convert result to elapsed_time
Return elapsed_time

10.4. Milliseconds

10.4.1. Interface

Milliseconds(integral milliseconds) return elapsed_time

10.4.2. PDL

Convert integral milliseconds to kernel_time
Convert result to elapsed_time
Return elapsed_time

10.5. Microseconds

10.5.1. Interface

Microseconds(integral microseconds) return elapsed_time

10.5.2. PDL

Convert integral microseconds to kernel_time
Convert result to elapsed_time
Return elapsed_time

10.6. "+"

10.6.1. Interface

"+" (elapsed time, elapsed time) return elapsed time

or

"+" (epoch time, elapsed time) return epoch time

10.6.2. PDL

Convert arguments to kernel_time
Call kernel_time "+" to perform operation
Convert result from Kernel_time to appropriate type
Return converted time value

10.7. "-"

10.7.1. Interface

"-" (elapsed time, elapsed time) return elapsed time

or

"-" (epoch time, elapsed time) return epoch time

or

"-" (epoch time, epoch time) return elapsed time

10.7.2. PDL

Convert arguments to kernel_time

Call kernel_time."-" to perform operation

Convert result from Kernel_time to appropriate type

Return converted time value

10.8. "*"

"*" (elapsed time, integer) return elapsed time

or

"*" (integer, elapsed time) return elapsed time

10.8.1. PDL

Convert arguments to kernel_time

Call kernel_time.*" to perform operation

Convert result from Kernel_time to appropriate type

Return converted time value

10.9. "/"

10.9.1. Interface

"/" (elapsed time, integer) return elapsed time

10.9.2. PDL

Convert arguments to kernel_time

Call kernel_time."/" to perform operation

Convert result from Kernel_time to appropriate type

Return converted time value

10.10. "<"

10.10.1. Interface

"<" (elapsed time, elapsed time) return boolean

or

"<" (epoch time, epoch time) return boolean

10.10.2. PDL

Convert arguments to kernel_time

Call kernel_time."<" to perform comparison

Return result

10.11. "<="

10.11.1. Interface

"<=" (elapsed time, elapsed time) return boolean

or

"<=" (epoch time, epoch time) return boolean

10.11.2. PDL

Convert arguments to kernel_time

Call kernel_time."<=" to perform comparison

Return result

10.12. ">"

10.12.1. Interface

">" (elapsed time, elapsed time) return boolean

or

">" (epoch time, epoch time) return boolean

10.12.2. PDL

Convert arguments to kernel_time

Call kernel_time.">" to perform comparison

Return result

10.13. ">="

10.13.1. Interface

">=" (elapsed time, elapsed time) return boolean

or

">=" (epoch time, epoch time) return boolean

10.13.2. PDL

Convert arguments to kernel_time

Call kernel_time.">=" to perform comparison

Return result

10.14. Base_time

10.14.1. Interface

base_time return epoch_time

10.14.2. PDL

Return base_time_value

10.15. To_epoch_time

10.15.1. Interface

To_epoch_time (kernel time) return epoch time

10.15.2. PDL

Convert argument to epoch_time

Return converted value

10.16. To_elapsed_time

10.16.1. Interface

To_elapsed_time (Ada duration) return elapsed time

or

To_elapsed_time (Ada duration) return elapsed time

10.16.2. PDL

Convert argument to elapsed_time
Return converted value

10.17. to_ada_duration

10.17.1. Interface

To_ada_duration (elapsed time) return ada duration

10.17.2. PDL

If elapsed time is not in range -86400 seconds to +86400 seconds then
 Raise constraint_error
Else
 Compute multiple of DURATION'SMALL not greater than
 the elapsed time
 Convert the result to duration
 Return converted value
End if

10.18. To_kernel_time

10.18.1. Interface

To_kernel_time (elapsed time) return kernel_time

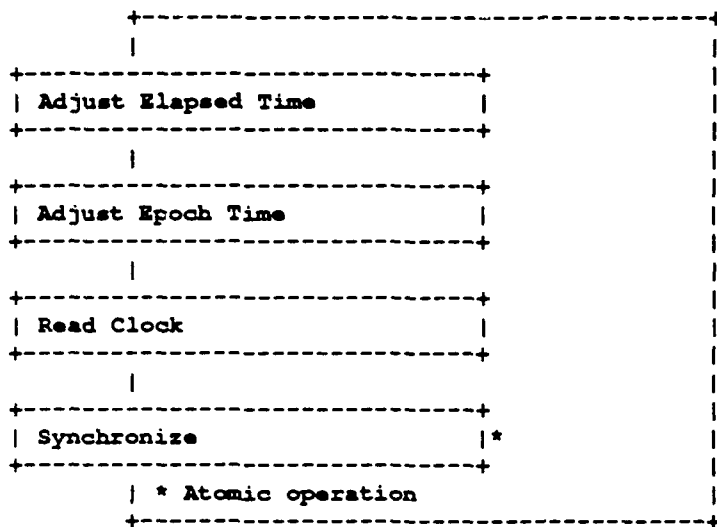
or

To_kernel_time (epoch time) return kernel_time

10.18.2. PDL

Convert argument to kernel_time
Return converted value

11. Time Management



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 21. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 12.

See Chapter 6 for a detailed discussion of the implementation of synchronize.

11.1. Adjust Elapsed Time

11.1.1. Interface

`adjust_elapsed_time (adjustment)`

11.1.2. PDL

```
If adjustment would result in a negative time of day then
    Raise illegal_elapsed_time exception
Else
    adjust_elapsed_time (adjustment) via time_keeper
End if
```

11.2. Adjust Epoch Time

11.2.1. Interface

`adjust_epoch_time (new time of day)`

11.2.2. PDL

```
If the new time of day is meaningless (i.e., less than zero) then
    Raise illegal_time_of_day exception
Else if the new time of day has already occurred then
    Raise ok_but_time_already_passed
Else
    reset_epoch_time (new time of day) via time_keeper
End if
```

11.3. Read Clock

11.3.1. Interface

`read_clock return current time of day`

11.3.2. PDL

`return get_time via Clock.get_time`

11.4. Synchronize

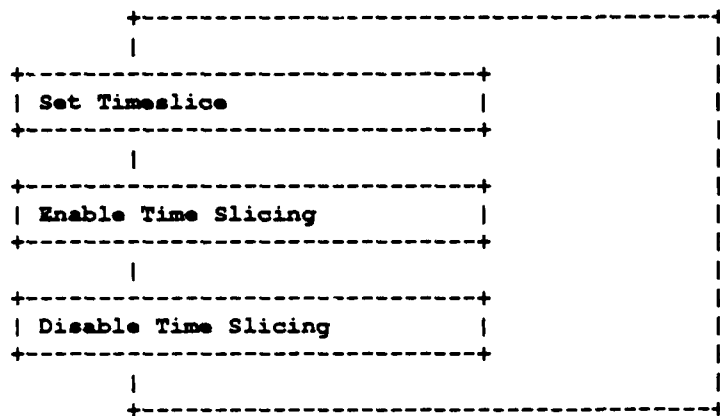
11.4.1. Interface

synchronize (timeout, resumption_priority)

11.4.2. PDL

```
Allocate synchronization bus
If allocated then
    Read current time of day
    Send current time of day to all other nodes
Else
    Set the exception_name to sync_n_progress
    Schedule (new priority => resumption priority)
End if
```

12. Timeslice Management



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 18. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 9.

12.1. Set Timeslice

This primitive does not affect the currently pending slice_expiration event. The next slice event uses the new slice quantum.

12.1.1. Interface

```
Set_timeslice (new quantum)
```

12.1.2. PDL

```
If illegal_quantum_enabled then
  If the new_quantum < minimum_time_slice then
    Raise illegal_quantum
  End if
End if

If the new_quantum < minimum_time_slice then
  Set timeslice_duration := minimum_time_slice
Else
  Set timeslice_duration := new quantum
End if
```

12.2. Enable Time Slicing

If time slicing is currently enabled, then this primitive performs no action.

12.2.1. Interface

```
Enable_time_slicing
```

12.2.2. PDL

```
If time_slicing_enabled is false =>
  Set time_slicing_enabled to true
  If the current_running_process is preemptable =>
    Set slice_event_id := Insert_event (slice expiration)
  End if
Else
  Null
End if
```

12.3. Disable_time_slicing

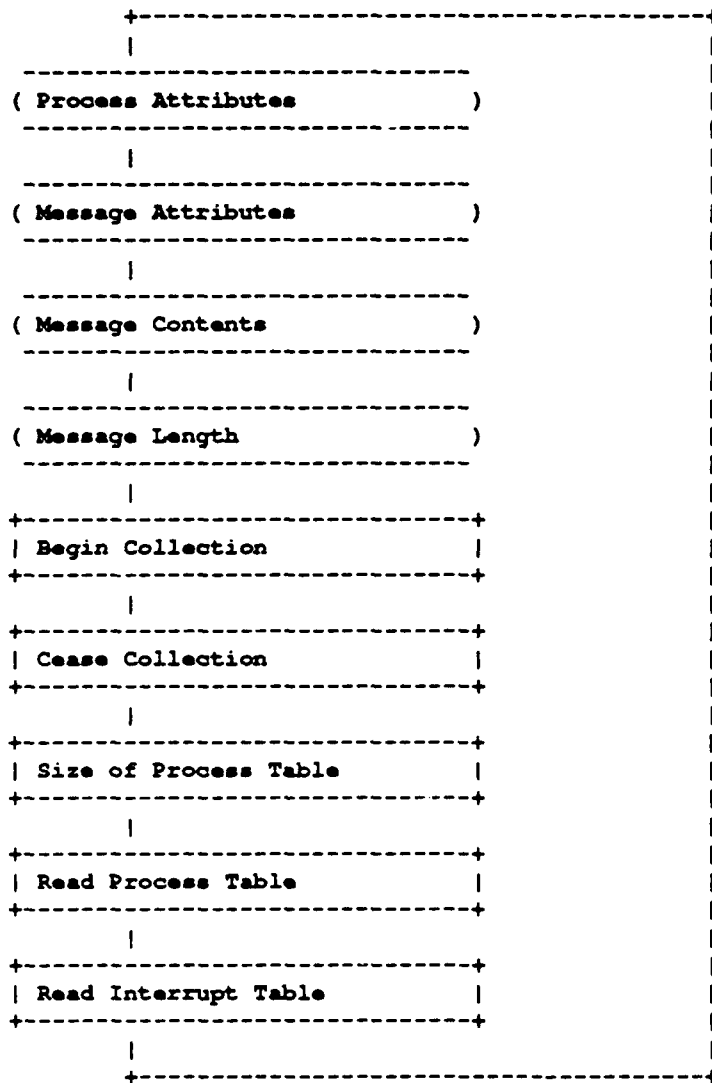
12.3.1. Interface

Disable_time_slicing

12.3.2. PDL

```
If time_slicing_enabled is true =>
    Set time_slicing_enabled to false
    Remove_event (slice_event_id)
Else
    Null
End if
```

13. Tool Interface



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 23. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 14.

The *tool_interface* package defines the user visible access to the Kernel's internal functioning. Three types of per-process attributes are available:

1. process attributes
2. message attributes
3. message contents

Each of these attributes is defined in more detail below. This package simply turns on and off the collection of tool data; it does not perform any of the actual collection or processing. Additional information on the collection of tool interface data can be found in Chapter 14. The format for a tool interface message is shown in Table 4.

13.1. Process Attributes

13.1.1. PDL

```
type process_attributes_entry is record
    id:                process identifier;
    state:              process state;
    time_state_change:  epoch_time;
    current_priority:   priority;
    current_preemption: preemption;
    alarm_pending:      boolean;
    primitive_identity: Kernel_primitive_name_type;
    primitive_return_status: kernel_exceptions;
end record;
```

13.2. Message Attributes

13.2.1. PDL

```
type message_attributes_entry is record
    sender_process_id:  full_process_id;
    receiver_process_id: full_process_id;
    message_length:     message_length_type;
    message_tag:        message_tag_type;
    time_Kernel_got_message: epoch_time;
end record;
```

13.3. Message Contents

13.3.1. PDL

```
type message_contents_type is array (hw_natural range <>) of hw_byte;
```

13.4. Message Length

13.4.1. PDL

```
message_length: array (range of per-process attributes)
    of message_length_type;
```

13.5. Begin Collection

13.5.1. Interface

```
procedure begin_collection (process on which to start collecting data,  
                           message tag for this information,  
                           type of information to collect)
```

13.5.2. PDL

```
If the process id of the process to monitor is bad then  
  Ignore the request  
Else  
  Mark the tool interface as enabled for the process  
  Log the tool process  
  Log the message tag  
End if
```

13.6. Cease Collection

13.6.1. Interface

```
procedure end_collection (process on which to stop collecting data,  
                         type of information to stop)
```

13.6.2. PDL

```
If the process id of the process to monitor is bad then  
  Ignore the request  
Else  
  Clear the tool process  
  If none of the individual tool attributes are still active then  
    Mark the tool interface as disabled (if none of the individual  
  End if  
End if
```

13.7. Size of process table

13.7.1. Interface

```
Function size_of_process_table  
  return number_of_entries_in_process_table
```

13.7.2. PDL

Return the size of the process table

13.8. Read Process Table

This operation is atomic so that a time-consistent snapshot of the Process Table is obtained.

13.8.1. Interface

```
procedure read_process_table (the user's process table buffer,  
                             the last entry filled in by the copy)
```

13.8.2. PDL

```
Begin atomic  
  Initialize a process table iterator  
  While there are more entries in the process table loop  
    Get the next entry in the process table  
    Store it in the caller's copy  
  End loop  
  Mark the end of the process table  
End atomic
```

13.9. Read Interrupt Table

13.9.1. Interface

```
procedure read_interrupt_table (the user's interrupt table buffer)
```

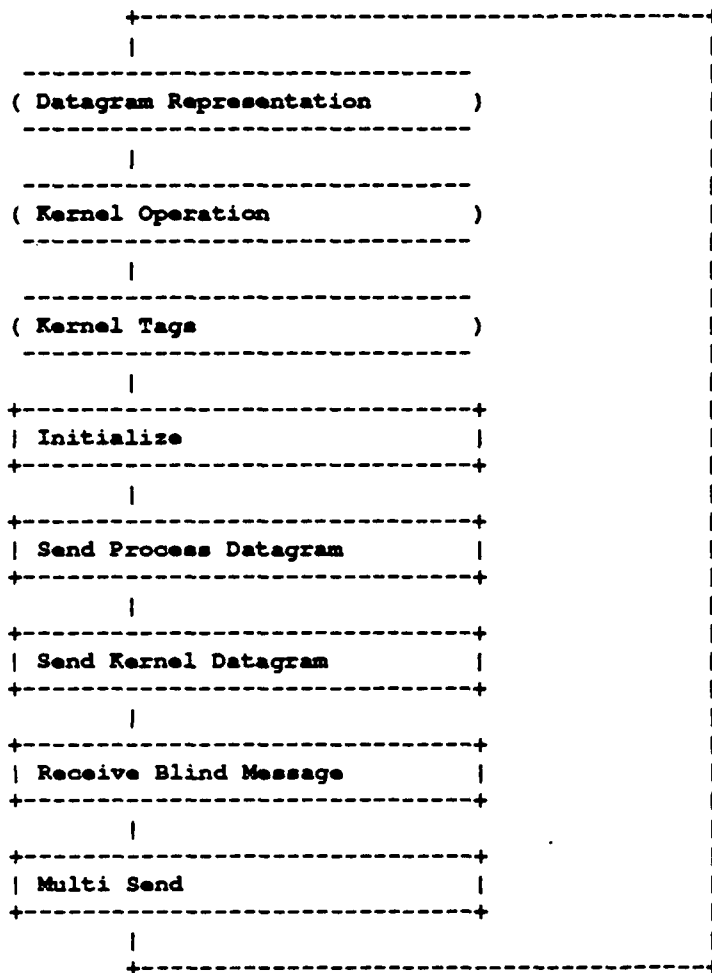
13.9.2. PDL

```
Copy the interrupt table for Kernel space to user space
```


III. Core Kernel

The packages described in this part are not directly visible to application-level code; but they are the means by which the functionality of the visible interface is achieved.

1. Bus I/O



This package is Kernel's interface to the network processor (NProc). As such, it is responsible for formatting datagrams for transmission over the network and receiving datagrams transmitted by other nodes in the network.

There are five message types in the system (the exact format of each of these messages is in Appendix B):

- Non-Kernel message: A datagram sent by a non-Kernel device. This is a user-level operation and the data contained in such a message is deposited in the allocated receiver's message queue.
- Blind send: A datagram sent without expectation of message receipt confirmation. This is a user-level operation and the data contained in such a message is deposited in the specified receiver's message queue.
- Acknowledged Send: A datagram sent with the expectation that an acknowledgment will be returned. This is a user-level operation, and the data contained in such a message is deposited in the specified receiver's message queue.

- **Kernel Message:** A datagram sent by one Kernel to its counterpart on another node. These messages are handled entirely within the receive datagram interrupt handler of the receiving Kernel. Associated with these messages is a set of message tags that supply additional information:
 - **Ack:** user message received correctly by destination process (sender's process ID and message id are part of the message contents).
 - **Nak:** user message not received by destination process (sender's process ID and message id are part of the message contents).
 - **Nak – Process Dead:** user message not received by destination process because it has terminated (sender's process ID and message id are part of the message contents).
 - **Info – Process Dead:** Kernel or user message not received by destination process because it has terminated.
 - **Kill Process:** a request to terminate a locally executing process.
 - **Network Failure:** a fatal communication error has been detected.
 - **Initialization Complete:** a node in the network has completed its initialization sequence.
 - **Process Created:** a node in the network has successfully created a local process.
- **Initialization Protocol Message:** a datagram sent during initialization by one Kernel to its counterpart on another processor. These messages are deposited in the "message queue" of the Main Unit for processing by the Kernel primitives that implement the initialization protocol. Associated with these messages is a set of message tags that supply additional information:
 - **Master Ready:** the Master processor is ready to commence the initialization protocol.
 - **NCT Enclosed:** a subordinate processor has sent its NCT to the Master Processor.
 - **Go Enclosed:** the Master processor has told a subordinate process to commence process creation.
 - **Go Acknowledgment:** a subordinate processor has acknowledged its receipt of the Go message.

The appearance of initialization complete and process created in the Kernel message tags and not as initialization protocol message tags requires some elaboration. First, while these tags are associated with initialization messages, their receipt by a processor does not correspond with a waiting primitive call by the recipient (as does the handshaking implemented by the initialization protocol messages). Second, since there is no corresponding primitive to capture these messages and process them; this processing is best done in the message interrupt handler of the receiver. Finally, this partitioning facilitates subsequent implementation of dynamic network configuration (at the process and processor levels).

1.1. Initialize

1.1.1. Invocation

Initialize

1.1.2. PDL

Bind Receive_datagram_interrupt_handler to the
interprocessor interrupt
Enable interprocessor interrupts

1.2. Send Process Datagram

1.2.1. Invocation

```
send_process_datagram (  
    message receiver,  
    message operation,  
    timeout,  
    message tag,  
    message identifier,  
    message length,  
    message text)
```

1.2.2. PDL

```
Allocate an empty datagram  
If a datagram is available then  
    If called from an interrupt handler then  
        Set sender's pid to null  
    Else  
        Set sender's pid to current_running_process  
    End if  
    Fill in datagram header information from parameters  
    Copy from the sender's buffer to the datagram's buffer  
    Enqueue the datagram on the output queue  
Else  
    Null...do nothing  
End if
```

1.3. Send Kernel Datagram

1.3.1. Invocation

```
send_kernel_datagram (  
    message sender,  
    message receiver,  
    message operation,  
    timeout,  
    message tag,  
    message identifier,  
    message length,  
    message text)
```

1.3.2. PDL

```
If receiver is a non-kernel device then  
    Return...Kernel messages don't go to non-Kernel devices  
End if  
Allocate an empty datagram  
If a datagram is available then  
    If called from an interrupt handler then  
        Set sender's pid to null  
    Else  
        Set sender's pid to current_running_process  
    End if  
    Fill in datagram header information from parameters  
    Copy from the sender's buffer to the datagram's buffer  
    Enqueue the datagram on the output queue  
Else  
    Null...do nothing  
End if
```

1.4. Receive Datagram I/H

1.4.1. Invocation

N/A

1.4.2. PDL

```
While there are messages waiting for the K-Proc  
    Dequeue a message  
    Case message operation is  
        When non-Kernel message =>  
            Receive_non_kernel_message  
        When blind send =>  
            Receive_blind_message  
        When acknowledged_send =>  
            Receive_acked_message  
        When kernel_message =>  
            Receive_kernel_message
```

```

    When initialization_protocol_message =>
        Indicate receiver has receive_pending
        Receive_Blind_message
    When sync_protocol_message =>
        null -- these are handled explicitly elsewhere...
    End Case
End if
End Loop

```

1.5. receive_non_kernel_message

1.5.1. Interface

```
receive_non_kernel_message (incoming datagram)
```

1.5.2. PDL

```

Perform error processing for:
    No process assigned to the non-kernel device for receiving messages
    Receiver process is dead
    Receiver message queue is full

Enqueue datagram on the receiver's message queue
If receiver is waiting then
    Schedule (new_state => suspended)
Else
    Null
End if

```

1.6. Receive Blind Message

1.6.1. Invocation

```
receive_blind_message (incoming datagram)
```

1.6.2. PDL

```

If the receiving process is dead then
    If the sender is not an interrupt handler then
        Send a Kernel-to-Kernel informational message back to the
            sender's node that the receiver is dead
    End if
    Free up the datagram buffer
Elsif the receiver's incoming message queue is full then
    Indicate the receiver's message queue has overflowed
    If receiver's overwrite rule is drop newest message then
        Free up the datagram buffer
    Else

```

```

        Null...do nothing
    End if
Else
    Enqueue a datagram on the receiver's msg queue
    If receiver is waiting then
        Schedule (new_state => suspended)
    End if
End if

```

1.7. Receive Acked Message

1.7.1. Invocation

```

receive_acked_message (incoming datagram)

```

1.7.2. PDL

```

If the receiving process is dead then
    If the sender is not an interrupt handler then
        Send a Kernel-to-Kernel informational message back to the
        sender's node that the receiver is dead
    End if
    Free up the datagram buffer
Elsif the receiver's incoming message queue is full then
    Indicate the receiver's message queue has overflowed
    If receiver's overwrite rule is drop newest message then
        Free up the datagram buffer
    Else
        Null...do nothing
    End if
Else
    Enqueue a datagram on the receiver's msg queue
    If receiver is waiting at a receive message call and
    it is also blocked then
        Enqueue datagram on receiver's msg queue
        Send an acknowledgment to the sender
        Schedule (new_state => suspended)
    Elsif the timeout is less than zero then
        Enqueue datagram on receiver's msg queue
        Indicate receiver is has a send with ack pending
    Elsif the timeout is zero then
        Send a negative acknowledgment to the sender
        Free up the datagram buffer
    Else
        Try to set a time event
        If no event was set then
            Send a negative acknowledgment to the sender
            Free up datagram
        Else
            Enqueue datagram on receiver's message queue

```

```

        Save pointer to datagram so it can be removed if the event
        expires.
        Indicate the sender has a send with ack pending
    End if
End if
End if

```

1.8. Receive Kernel Message

It is possible for a process to perform a `send_message_and_wait` operation and then to terminate before the acknowledgment is received. Since all operations involving a dead process are meaningless, all acknowledgments to a dead process are ignored. Also, the Kernel makes no attempt to inform other Kernels of this condition, preferring, instead, for a process to make a subsequent attempt to communicate with the dead process.

Also note that it is possible for two (or more) different processors to create the same process, but that condition is not detected by this procedure. The reasons being that this condition is caught either:

- When the second processor attempts to create the process, for the case when the two creations don't occur simultaneously.
- When all the processors have signaled initialization complete, for the case when the two creations occur simultaneously.

The situation is easy to detect, since a remote process should not have any Scheduler state on the local processor.

1.8.1. Invocation

```

receive_kernel_message (incoming datagram)

```

1.8.2. PDL

```

Obtain the message tag from the message header
Case message_tag is
  When ack =>
    If the receiver (the message originator) is waiting for an
      acknowledgment and the message id matches then
      Schedule (new_state => suspended)
    End if
  When nak =>
    If the receiver (original msg sender) is waiting for an
      acknowledgment and the message id matches then
      Setup to raise the No_Message_Received exception
      Schedule (new_state => suspended)
    End if
  When nak - process dead =>
    Indicate that the specified process (process to which the msg was
      sent to) is dead.

```

```

        If the receiver (original msg sender) is waiting for an
            acknowledgment and the message id matches then
                Setup to raise the Receiver_Dead exception
                Schedule (new_state => suspended)
        End if
    When info - process dead =>
        Indicate that the specified process is dead.
    When kill process =>
        Kill the specified process
    When Process Created =>
        Obtain process name from message
        If the process has not already been declared then
            Do the processing for declaring the process
        End if
        Map the global and local identifiers
    When Initialization Complete =>
        Set the processor identifier
        Indicate initialization complete
    When Network Failure =>
        Null
End Case

```

1.9. Multi Send

This procedure is used only to broadcast "Network Failure" messages to all the nodes capable of receiving the message.

1.9.1. Invocation

```

multi_send (
    message sender,
    message operation,
    message tag,
    message length,
    message text)

```

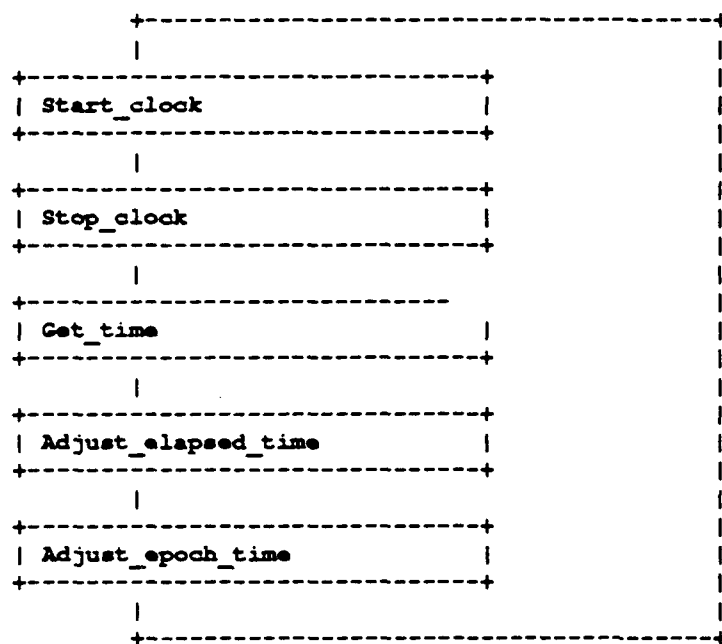
1.9.2. PDL

```

For all of the nodes that are not either the local node and
are Kernel devices loop
    Send a kernel datagram with the provided message tag
End loop

```

2. Clock



The real-time clock is implemented using one of the timers available on the MZ8305 board, see Part VIII for a more detailed discussion of the actual timer hardware. Using this timer allows the Kernel to have a time base independent of the Ada runtime and achieve a resolution of 2 μ s.

Time is represented as a 64-bit signed value. This representation consists of two parts:

1. Hardware: the lower 24 bits of the time and is updated autonomously by the timer hardware at a 2 μ s rate.
2. Software: the upper 40 bits of the time and is updated by the clock interrupt handler every 2^{23} clock ticks, i.e., every 16 seconds.

When the clock is read, the low 24 bits of the time are read from hardware and added to the upper 40 bits to produce the full 64-bit time value. See Chapters 10 and 7 for additional time details.

2.1. start_clock

2.1.1. Interface

start_clock (current time of day)

2.1.2. PDL

Save the base epoch time
Bind the interrupt handler for the clock
Initialize the timer that functions as the clock

2.2. stop_clock

2.2.1. Interface

stop_clock

2.2.2. PDL

Disable the interrupt of the clock's timer

2.3. get_time

2.3.1. Interface

get_time return current time of day

2.3.2. PDL

Get a copy of the upper 40 bits of the elapsed time
Read the lower 24 bits from timer (clock, counter register)
If the elapsed time has rolled over then
 it happened somewhere between reading the hardware and software portions
 of the current time...so, assume that the call hit the rollover point
 exactly and return the current value of elapsed time as the current time
 return the current time of day (as computed by the clock interrupt handler)
Else, there was no roll over, so
 Convert the timer count into microseconds
 Add it into the 64-bit representation of time
 Return the current time of day
End if

2.4. adjust_elapsed_time

2.4.1. Interface

`adjust_elapsed_time (adjustment)`

2.4.2. PDL

```
Read the current software time
Compute new time ::= current software time + adjustment
Begin atomic
  Read the current software time again
  If the two software times are not the same then
    a clock interrupt has occurred
    Compute new time := current software time + adjustment
  End if
  Save the new software time
End atomic
```

2.5. adjust_epoch_time

2.5.1. Interface

`adjust_epoch_time (new time of day)`

2.5.2. PDL

```
Compute delta time := new time of day -
                        time of day when clock was started
Adjust_elapsed_time by the delta time
Reset the the current time of day to the new time of day
```

2.6. clock_interrupt_handler

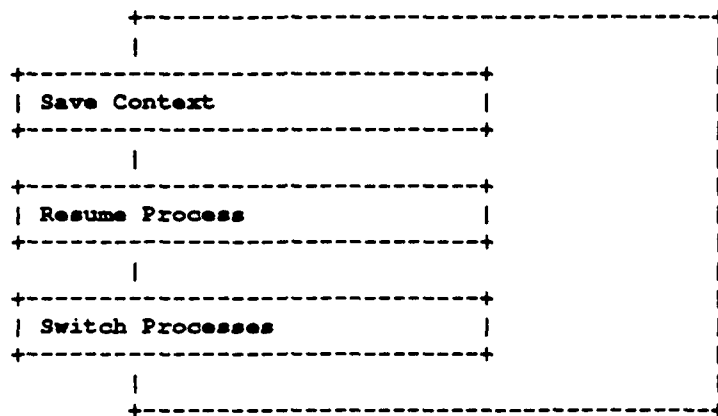
2.6.1. Interface

`clock_interrupt_handler`

2.6.2. PDL

```
Acknowledge the timer interrupt
Increment the current time of day
```

3. Context Switcher



This package is hardware-dependent and compiler-dependent. It must understand the register structure of the underlying hardware and the conventions used by the Ada compiler. This package is responsible for providing the mechanisms needed to save a process context (when an interrupt occurs or a context switch occurs) and to restore a saved process context (when the Scheduler selects a process for execution).

3.1. Save Context

3.1.1. Interface

`Save_context (process identifier)`

3.1.2. PDL

Set Context saved field of process table to by_call
Pop PC off interrupt stack
Copy PC into current running process's context save area
Pop status register off interrupt stack
Copy status register into current running process's context save area
Copy live registers into current running process's context save area

3.2. Resume Process

3.2.1. Interface

`Resume_context (process identifier)`

3.2.2. PDL

Copy live registers from CURRENT RUNNING PROCESS's context save area
Set saved context to none
If an exception is pending for this process =>
 Raise exception via Exception Raiser
Else
 Execute return instruction to resume process
End if

3.3. Switch Processes

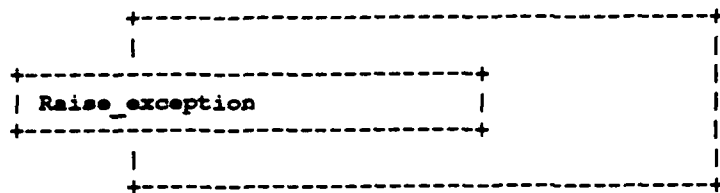
3.3.1. Interface

`Switch_processes (old process identifier,
 new process identifier)`

3.3.2. PDL

`Save_context (old process identifier)`
`Set current running process to new process identifier`
`Resume_process (new process identifier)`

4. Exception_Raiser



This package interfaces with the Ada compiler primitives that raise and propagate exceptions.

All the possible Kernel exceptions are contained in package `Kernel_exceptions` and documented in the *Kernel User's Manual*.

4.1. Raise Exception

4.1.1. Interface

`Raise_exception (exception name)`

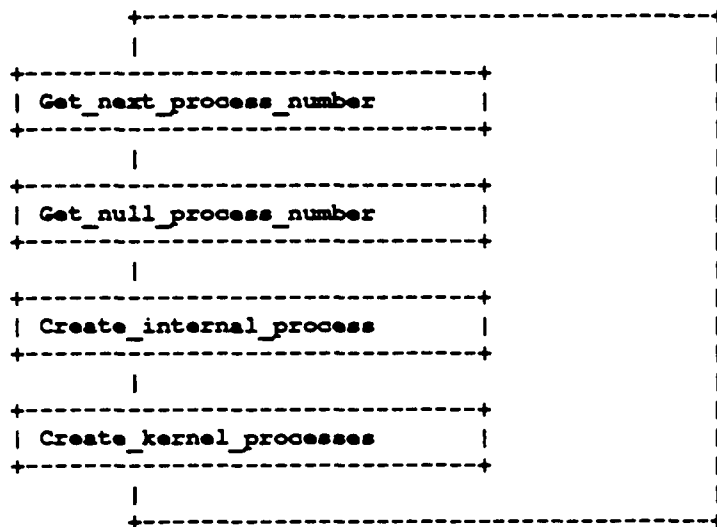
4.1.2. PDL

```
Case Exception name is
  When ... =>
    Raise indicated exception
  When ... =>
```

```
  .
  .
  .
```

`End case`

5. Internal Process Management



This package collects together a number of loosely related activities that support process creation.

While the process identifier is used to reference a process locally, it is not suitable for use in identifying processes remotely (because it is an access type). Thus, each process has a locally unique process number (a 16-bit integer), that, when appended to the node number where the process is executing, gives a globally unique 32-bit process index. These process numbers are created and doled out by this package.

Second, each Main Unit is treated by the Kernel as an executing process. Since the user does not explicitly create them as such, they are "created" internally by the Kernel during initialization. This allows all the Kernel facilities built to support user processes to be applied in support of the Main Unit and processor initialization.

Finally, one additional internal process is created: a time burner. This process has a priority lower than any user process and can only run when other user processes on the node are blocked or dead. This process allows for a more efficient scheduling algorithm, since there is never a situation where no process is eligible to run. It also allows the user to put in place code that measures idle time on a processor or runs some other appropriate background work.

5.1. Get_next_process_number

5.1.1. Interface

`get_next_process_number` return process number

5.1.2. PDL

Compute the next available process number

Return the value to the caller

Exception

when any exception occurs

propagate the exception

5.2. Get_null_process_number

5.2.1. Interface

`get_null_process_number`

return process number for null process

5.2.2. PDL

Return the process number reserved for the null process

5.3. Create_internal_process

This procedure does an abbreviated declare and create for processes that the Kernel needs to create internally.

5.3.1. Interface

`create_internal_process`

(node where the process resides

process number

local process indicator

process name

initial process priority

code address)

return process identifier

5.3.2. PDL

Reference `declare_process` and `create_process` for a detailed exposition on the functioning of this code. The primary differences are that the code assumes sufficient space exists to create the processes and it has preset values for the user options.

5.4. Create_kernel_processes

5.4.1. Interface

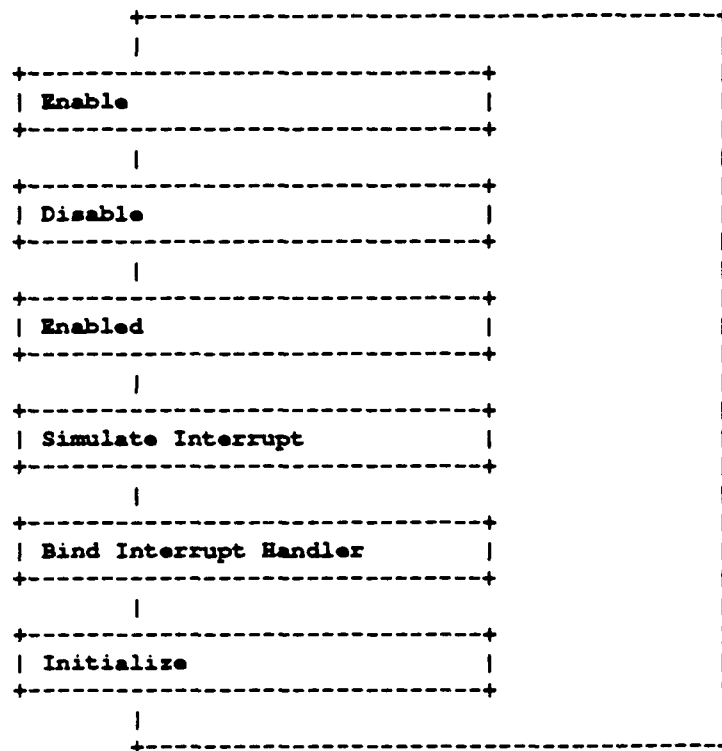
create_kernel_processes

5.4.2. PDL

```
For each node in the network loop
  If the node is this node then
    Create the local Main Unit
  Else
    Create the remote Main Unit
  End if
End loop

Create the time burner process
```


6. Kernel Interrupt Management



6.1. Enable

6.1.1. Interface

`Enable (interrupt name)`

6.1.2. PDL

Set interrupt state in the interrupt table to enabled
Set interrupt's entry in the Kernel interrupt vector to point to
the user-supplied interrupt handler

6.2. Disable

6.2.1. Interface

`Disable (interrupt name)`

6.2.2. PDL

Set interrupt state in the interrupt table to disabled
Set interrupt's entry in the Kernel interrupt vector to point to
the null interrupt handler

6.3. Enabled

6.3.1. Interface

`Enabled (interrupt name)`
return boolean

6.3.2. PDL

Return the `interrupt_state` field of the interrupt table

6.4. Simulate Interrupt

6.4.1. Interface

`Simulate_interrupt (interrupt name)`

6.4.2. PDL

```
Set interrupt_source in the interrupt table to internal
Increment interrupt_nesting level
Begin atomic
    Perform an indirect call of the interrupt handler pointed
        to by the interrupt's entry in the interrupt table

    Exception
        when others =>
            handle all exceptions, taking no action (to simulate the
                effect of an unhandled exception in a real interrupt
                processing)
End atomic

Decrement the interrupt_nesting level
Set the interrupt source to be External
If the interrupt can preempt then
    Schedule
Else
    return to the caller
End if
```

6.5. Bind Interrupt Handler

6.5.1. Interface

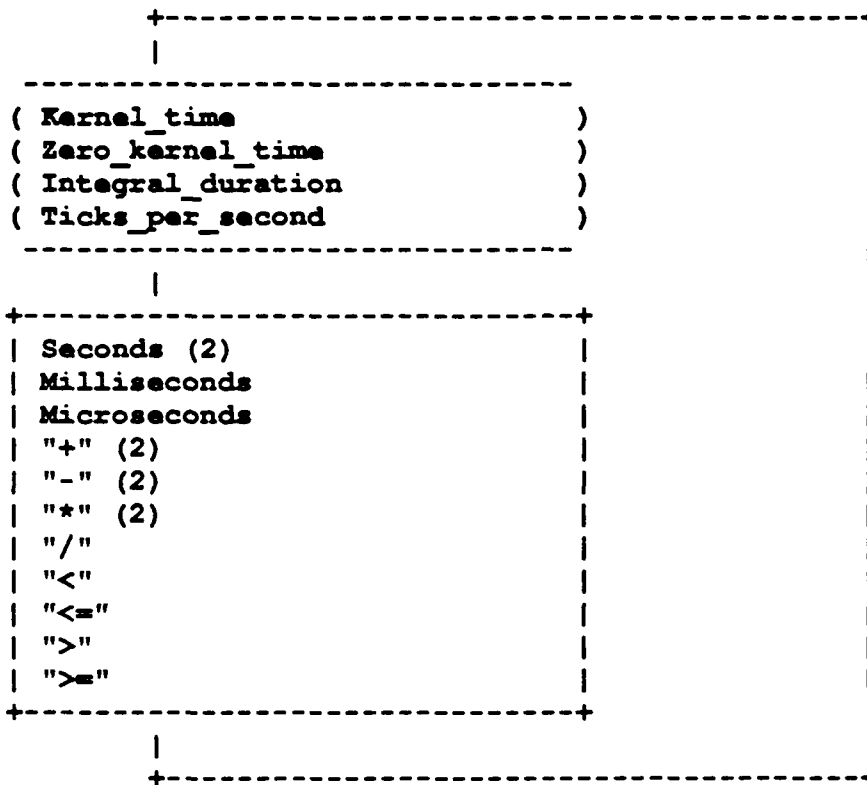
```
bind_interrupt_handler (interrupt name,
                        address of interrupt handler procedure,
                        interrupt can cause process preemption)
```

6.5.2. PDL

```
Insert interrupt name into Interrupt table
Store handler address in the interrupt table
Store value for can_preempt in the interrupt table
Set handler state to bound in the Interrupt table

If the interrupt can_preempt then
    Call machine-dependent routine to bind a "slow" interrupt
Else
    Call machine-dependent routine to bind a "fast" interrupt
End if
```

7. Kernel Time



7.1. Seconds

7.1.1. Interface

Seconds(an Ada duration) return kernel_time

7.1.2. PDL

Multiply the Ada duration by 1_000_000
Divide result by 16384 -- DURATION'SMALL
Return resulting kernel time

7.1.3. Interface

Seconds(an integral duration) return kernel_time

7.1.4. PDL

Multiply the Ada duration by 1_000_000
Return resulting kernel time

7.2. Milliseconds

7.2.1. Interface

Milliseconds(integral milliseconds) return elapsed_time

7.2.2. PDL

Multiply the integral milliseconds by 1_000
Return resulting kernel time

7.3. Microseconds

7.3.1. Interface

Microseconds(integral microseconds) return elapsed_time

7.3.2. PDL

Sign extend the integral microseconds to 64 bits
Return resulting kernel time

7.4. "+"

7.4.1. Interface

"+" (kernel time) return kernel

7.4.2. PDL

Return argument as result

7.4.3. Interface

"+" (kernel time, kernel time) return kernel time

7.4.4. PDL

Add low order 32 bits unsigned
Add with carry high order 32 bits signed
Trap on hardware overflow
Return 64 bit result

7.5. "-"

7.5.1. Interface

"-" (kernel time) return kernel

7.5.2. PDL

Negate 64 bit argument and return

7.5.3. Interface

"-" (kernel time, kernel time) return kernel time

7.5.4. PDL

Subtract low order 32 bits unsigned
Subtract with borrow high order 32 bits signed
Trap on hardware overflow
Return 64 bit result

7.6. "*"

7.6.1. Interface

"*" (LHS => kernel time, RHS => integer) return kernel time

7.6.2. PDL

```
If RHS < 0 then
    LHS := -LHS
    RHS := -RHS
End if
Multiply low 32 bits of LHS by RHS unsigned
    store the 64 bit result
Multiply high 32 bits of LHS by RHS signed
    store the 32 bit result
Trap on overflow
Add two partial products signed
Trap on overflow
Return 64 bit result
```

7.6.3. Interface

"*" (LHS => integer, RHS => kernel time) return kernel time

7.6.4. PDL

As above, interchanging LHS and RHS

7.7. "/"

"/" (LHS => kernel time, RHS => integer) return kernel time

7.7.1. PDL

```
If RHS < 0 then
    LHS := -LHS
    RHS := -RHS
End if

Divide high 32 of LHS by RHS signed, giving q1 and r1

If high 32 bits of LHS < 0 then
    q1 := q1 - 1
    r1 := RHS + r1
end if

Divide r1, LHS.low by RHS double length unsigned, giving q0 and r0
```

Return q1,q0 as the 64 bit result

7.8. "<"

7.8.1. Interface

"<" (LHS => kernel time, RHS => kernel time) return boolean

7.8.2. PDL

```
If high 32 bits of LHS < high 32 bits of RHS then -- signed comparison
    return true
Elsif high 32 bits of LHS > high 32 bits of RHS then -- signed comparison
    return false
Else
    If low 32 bits of LHS < low 32 bits of RHS then -- unsigned comparison
        return true
    Else
        return false
    End if
End if
```

7.9. "<="

7.9.1. Interface

"<=" (LHS => kernel time, RHS => kernel time) return boolean

7.9.2. PDL

```
If high 32 bits of LHS < high 32 bits of RHS then -- signed comparison
    return true
Elsif high 32 bits of LHS > high 32 bits of RHS then -- signed comparison
    return false
Else
    If low 32 bits of LHS <= low 32 bits of RHS then -- unsigned comparison
        return true
    Else
        return false
    End if
End if
```

7.10. ">"

7.10.1. Interface

">" (LHS => kernel time, RHS => kernel time) return boolean

7.10.2. PDL

```
If high 32 bits of LHS > high 32 bits of RHS then -- signed comparison
    return true
Elsif high 32 bits of LHS < high 32 bits of RHS then -- signed comparison
    return false
Else
    If low 32 bits of LHS > low 32 bits of RHS then -- unsigned comparison
        return true
    Else
        return false
    End if
End if
```

7.11. ">="

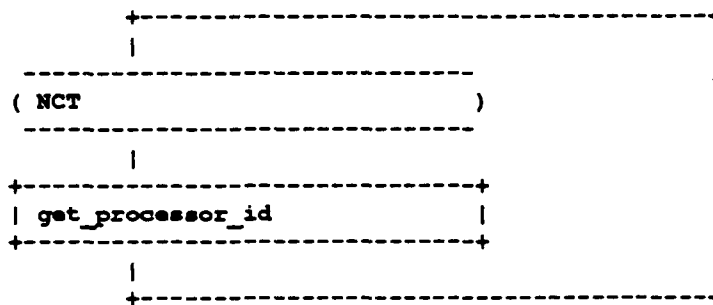
7.11.1. Interface

">=" (LHS => kernel time, RHS => kernel time) return boolean

7.11.2. PDL

```
If high 32 bits of LHS > high 32 bits of RHS then -- signed comparison
    return true
Elsif high 32 bits of LHS < high 32 bits of RHS then -- signed comparison
    return false
Else
    If low 32 bits of LHS >= low 32 bits of RHS then -- unsigned comparison
        return true
    Else
        return false
    End if
End if
```

8. Network Configuration



For a description of the functionality of this package, see *Kernel Facilities Definition*, Chapter 15. The requirements satisfied by this package are found in the *Kernel Facilities Definition*, Chapter 6.

8.1. get_processor_id

8.1.1. Interface

`get_processor_id (node address) return NCT index`

8.1.2. PDL

```
For each node in the NCT loop
  If the address of the node is the one in question then
    Return the NCT index of the node
  End if
End loop
```

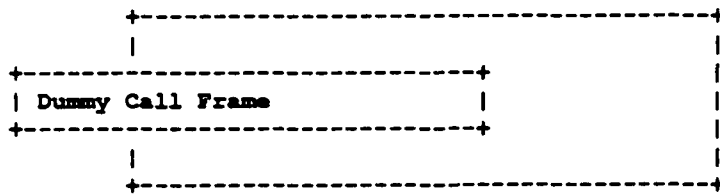
8.2. NCT

The NCT (shown in Figure 17) is discussed in detail in both [KFD 89] and [KUM 89].

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

Figure 17: Network Configuration Table

9. Process_Encapsulation



This package contains the procedure that handles unexpected terminations of Kernel processes. There are two kinds of unexpected terminations:

1. Termination: The process simply completes its processing and reaches its final end statement. When this occurs, the Kernel terminates the process.
2. Unhandled exception: an exception occurs in a process, but no piece of code within the process handles that exception. When this occurs, the Kernel terminates the process and optionally issues a diagnostic message.

9.1. Dummy_call_frame

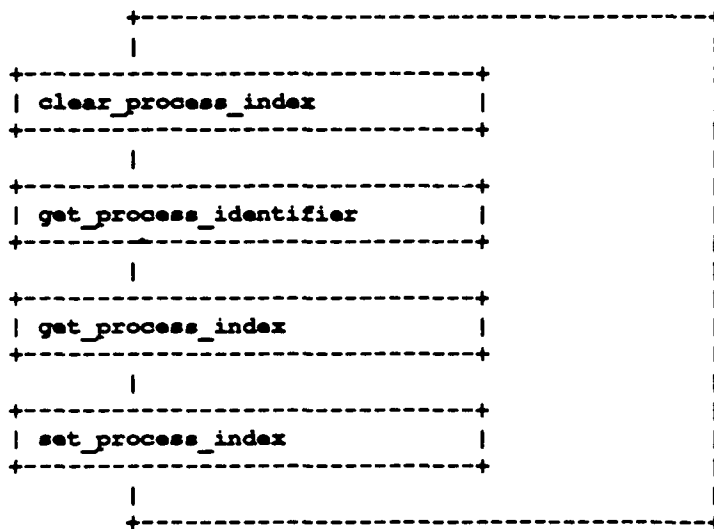
9.1.1. Interface

`dummy_call_frame (process identifier)`

9.1.2. PDL

```
Set initial scheduling parameters
Insert_process into Scheduler
Begin atomic
    Perform indirect call of user process code
    Die
Exception
    when others =>
        If traceback_enabled then
            Print stack traceback message
        End if
        Die
End atomic
Schedule
```

10. Process Index Table



As discussed elsewhere, a process had two internal handles:

1. Process identifier: the access variable of the process table entry for the process.
2. Process index: the globally unique identifier for the process.

This package encapsulates the mapping between the two handles. The mapping table is built dynamically during processor initialization by Main Unit interaction with the Kernel.

10.1. Clear_process_index

10.1.1. Interface

`Clear_process_index (process index)`

10.1.2. PDL

Remove a process identifier from the mapping table by setting the entry in the mapping table to null

10.2. Get_process_identifier

10.2.1. Interface

`get_process_identifier (process index)`
return process identifier

10.2.2. PDL

Return the process identifier contained in the mapping table for the requested process index

10.3. get_process_index

10.3.1. Interface

`get_process_index (process identifier)`
return process index

10.3.2. PDL

Return the process index from the process table for this process

10.4. set_process_index

10.4.1. Interface

`set_process_index (process identifier,
process index)`

10.4.2. PDL

Log the process identifier in the mapping table at the
process index position

10.5. Mapping

The mapping table is a simple table indexed by (bus address, process number) and internal to the package. Each entry in the table holds the process identifier (i.e., the index into the local process table for that process). Thus, when a message arrives at a node, the (bus address, process number) pair of the receiving process bundled in the datagram is used to index the mapping array and retrieve the index into the process table. The mapping table is shown in Figure 18.

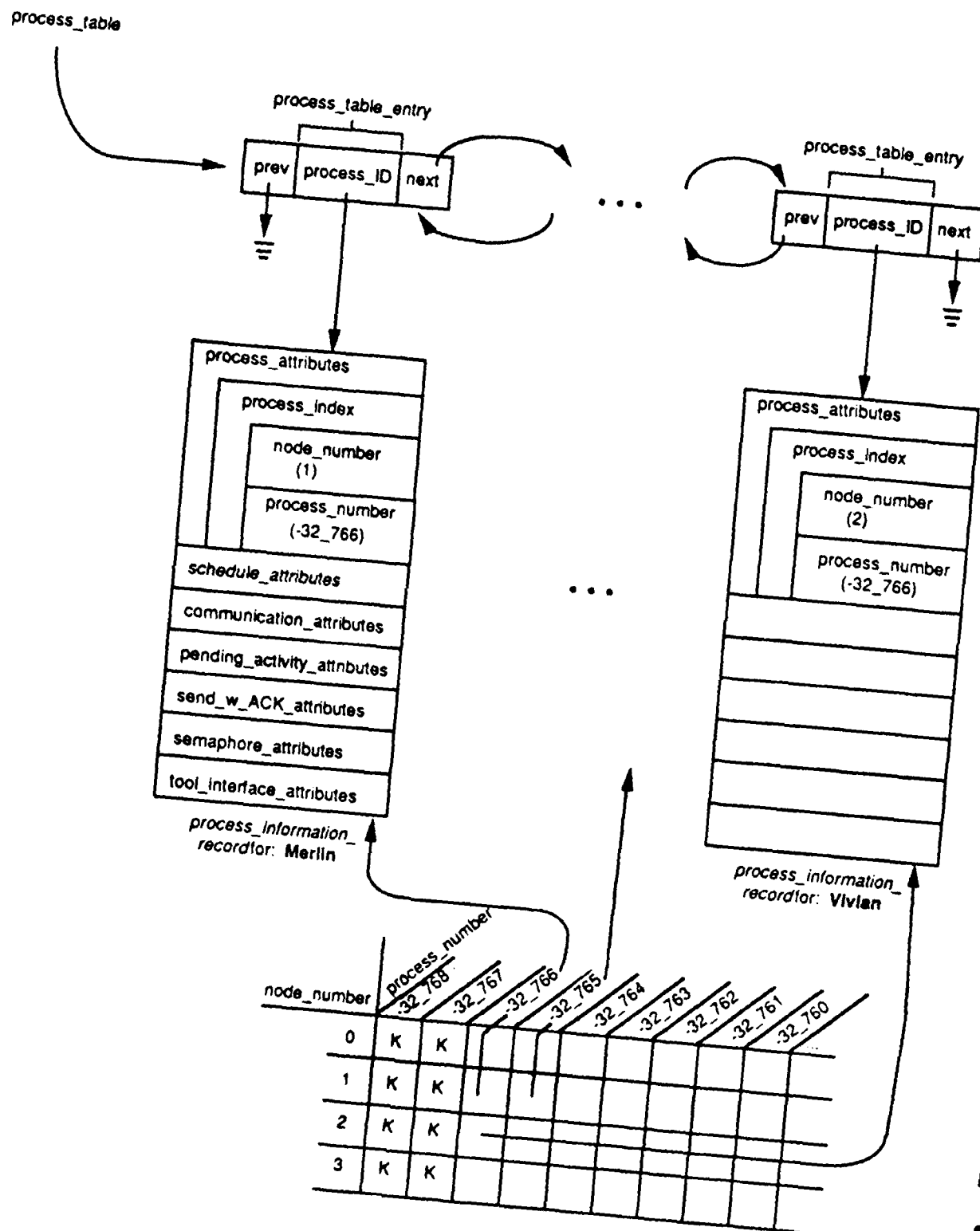


Figure 18: Process Mapping Table

11. Process Table

The process table is the primary data structure of the Kernel. It holds all the Kernel state related to a process. The process table is structured as a doubly linked list, shown in Figure 11, where the data in the linked list is a pointer to a process information record, i.e., a PID. This gives the Kernel quick access into the process data, the ability to traverse the entire process structure and the capability to expand in the future. It is documented fully in the package specification and discussed in more detail in [KUM 89].

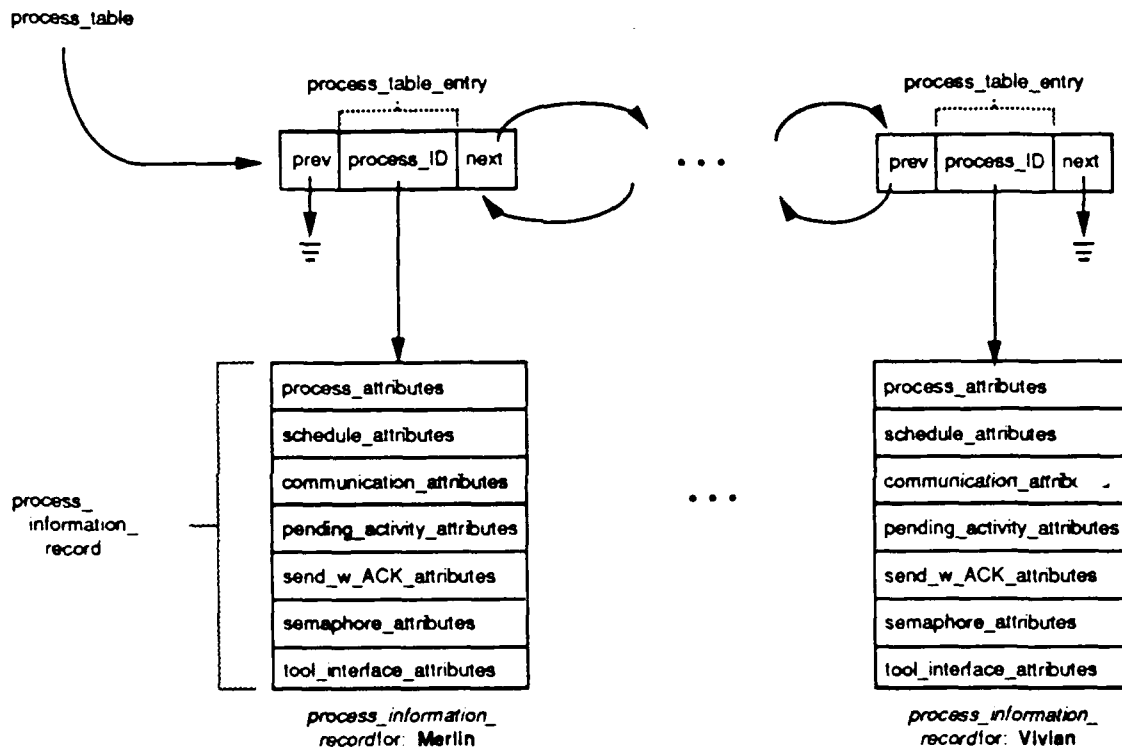


Figure 19: Process Table

One part of the process table that needs elaboration is the pending activity attributes record. This structure is responsible for maintaining the state of a process when it is blocked and no longer under the control of the Scheduler. There are five pending activities encapsulated by this structure:

1. Claim pending: the process is currently blocking on an unsatisfied claim operation.
2. Receive pending: the process is currently blocked waiting for a message to arrive.
3. Wait pending: the process is currently blocking waiting for the passage of some period of time.
4. Send with ACK pending: the process is currently blocked waiting for the arrive of a message acknowledgment.

5. Nothing pending: the process is not blocked. It is either dead or eligible to run (and thus under the control of the Scheduler again).

Figure 11 shows these pending activity states and the transitions that occur between.

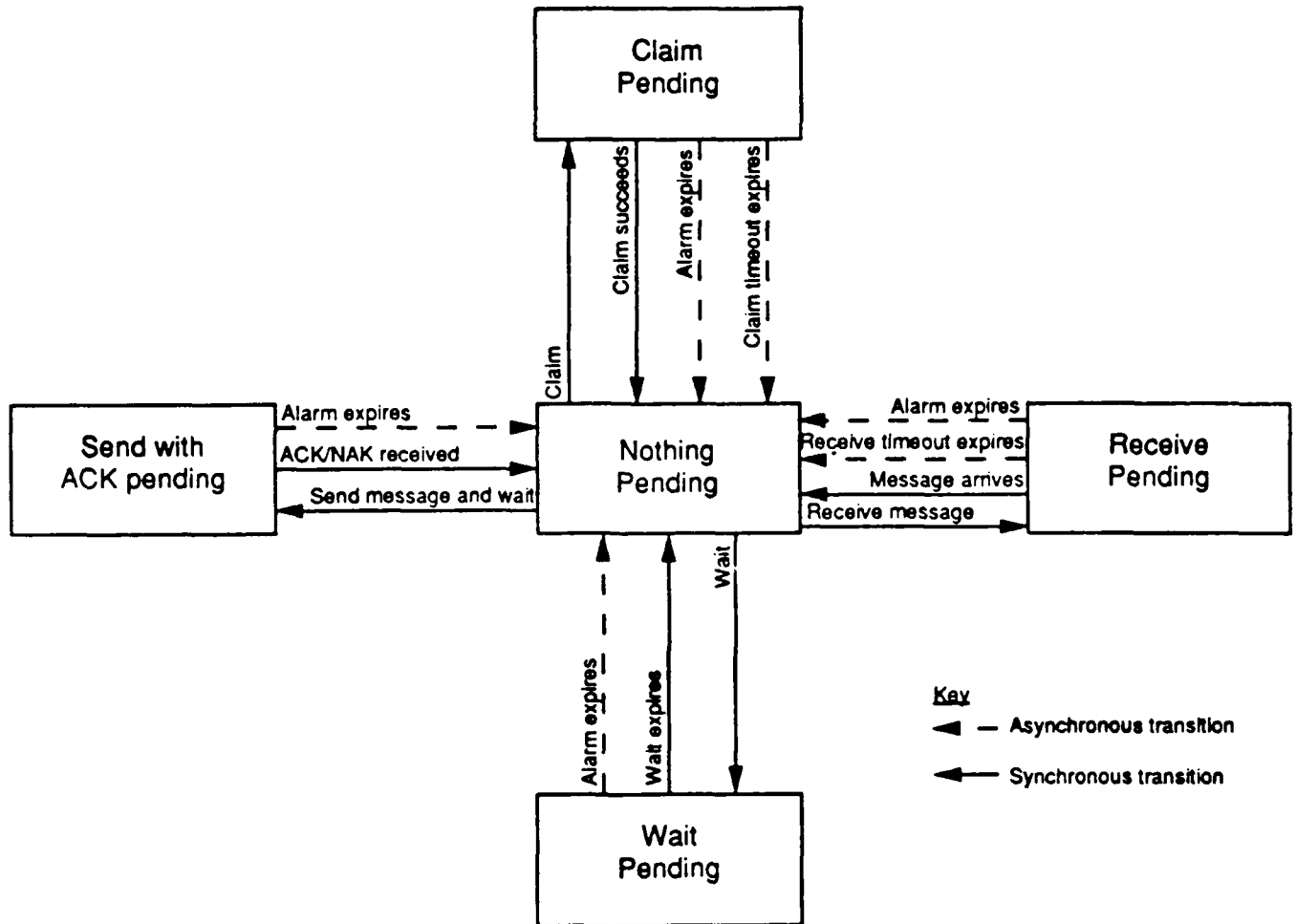
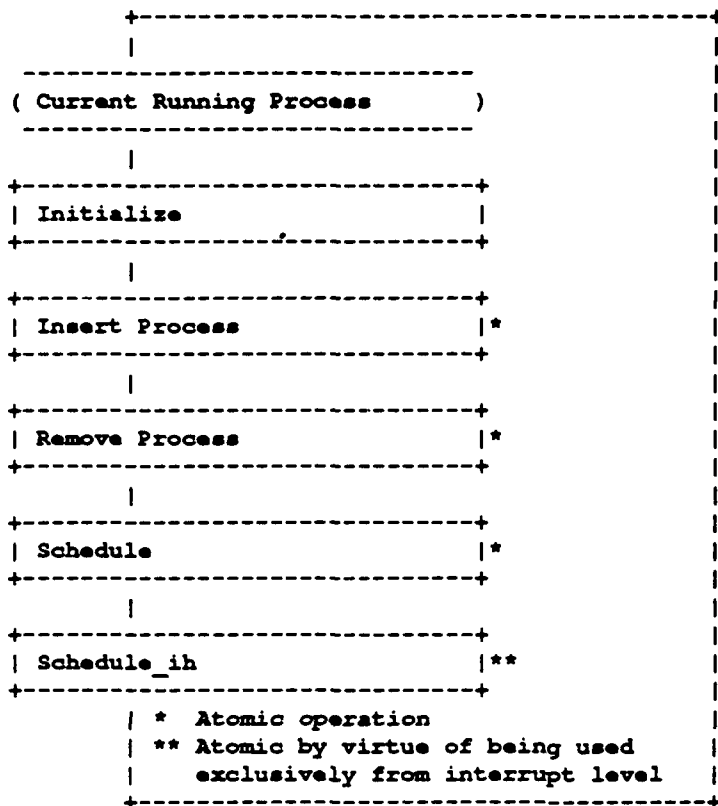


Figure 20: Pending Activity States

12. Scheduler



The Kernel Scheduler manages Kernel processes. In particular, the Scheduler, and it alone, makes the decision to resume a specific process (i.e., change the state of a process from suspended to running).

The Scheduler knows only about processes that are running or suspended. All processes in other (i.e., blocked or dead) states are maintained outside the knowledge and control of the Scheduler. There are three reasons for this:

1. Running and suspended are the only states relevant to the functioning of the Scheduler.
2. A blocked process is unable to run until the corresponding unblocking event occurs, thus there is no reason for the Scheduler to maintain any information about the process.
3. This facilitates replacing the default Scheduler with one of the user's choice.

Thus, when a process is not blocked or dead (i.e., it is capable of being run) it is *inserted* into the Scheduler. When a process blocks or dies, it is *removed* from the Scheduler.

All Kernel primitives that could cause a process to unblock (and consequently result in a context switch) end with a call to *schedule*.

Internally, the Scheduler maintains a run queue ordered by priority. The resumption priority

of a suspended process is the priority seen by the Scheduler; any priority of the process before an invocation of the relevant Kernel primitive is no longer germane.

Rules Implied by Kernel Requirements

1. Only the current running process and the set of suspended processes are available for scheduling.
2. A process of higher priority must be scheduled in preference to one of lower priority ([KFD 89] Chapter 18).
3. A suspended process of a given priority must be scheduled in preference to one at the same priority whose timeslice has just expired ([KFD 89] Requirement 9.1.10).
4. If two or more processes change state at the same time (that is, at the same slice), the changes happen simultaneously (that is, it cannot be the case that one process has changed state and can detect that another has not) ([KFD 89] Chapter 18).

Information Available to the Scheduler

The following information from the Process Table is used by the Scheduler:

1. Process state
2. Process priority
3. Process preemption state
4. The name of any pending exception (if the process is in an error state)

Scheduling Events

The following lists all the Kernel primitives where the Kernel requirements imply a scheduling action may occur:

- Adjust_elapsed_time
- Claim
- Die
- Initialization Complete
- Kill
- Receive_message
- Release
- Reset_epoch_time
- Send_message_and_wait
- Set_process_priority
- Set_process_preemption
- Synchronize
- Wait

The following events may also cause the a scheduling action to occur:

- Expiry of an alarm
- Expiry of a timeout
- Expiry of a timeslice
- Expiry of a wait
- Preemptive interrupt
- Obtaining a claimed semaphore
- Receipt of a message
- Receipt of a message ACK or NAK

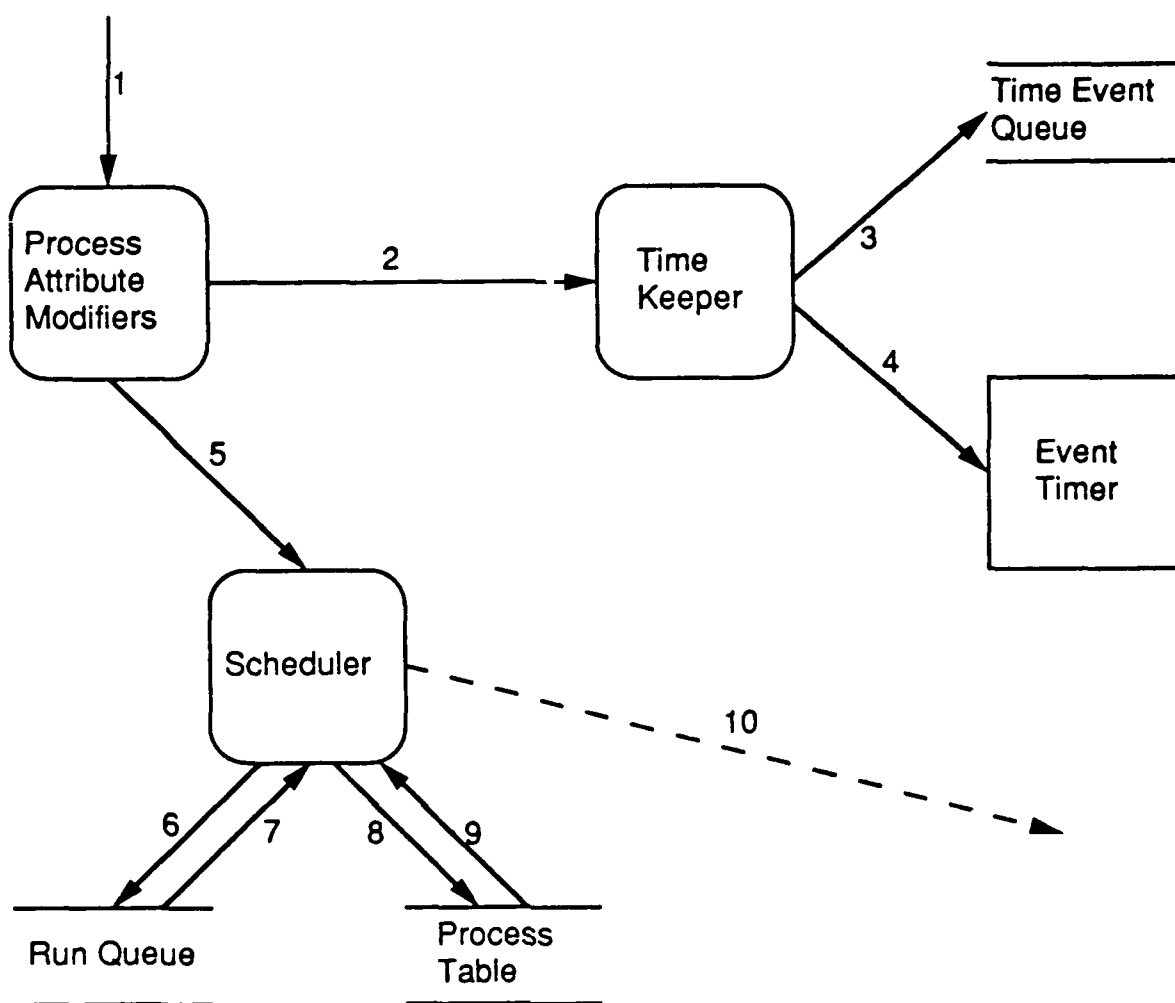
Scheduler Rules

The following Scheduler rules are universally applied:

1. Scheduler order does not change spontaneously.
2. Scheduler ordering is decided by the rules:
 - a. Higher priority before lower priority
 - b. Prefer a process in an error state (to one in a normal state)
 - c. FIFO order otherwise
3. When two processes become unblocked simultaneously, the process that has been blocked longest is considered to become unblocked first.³

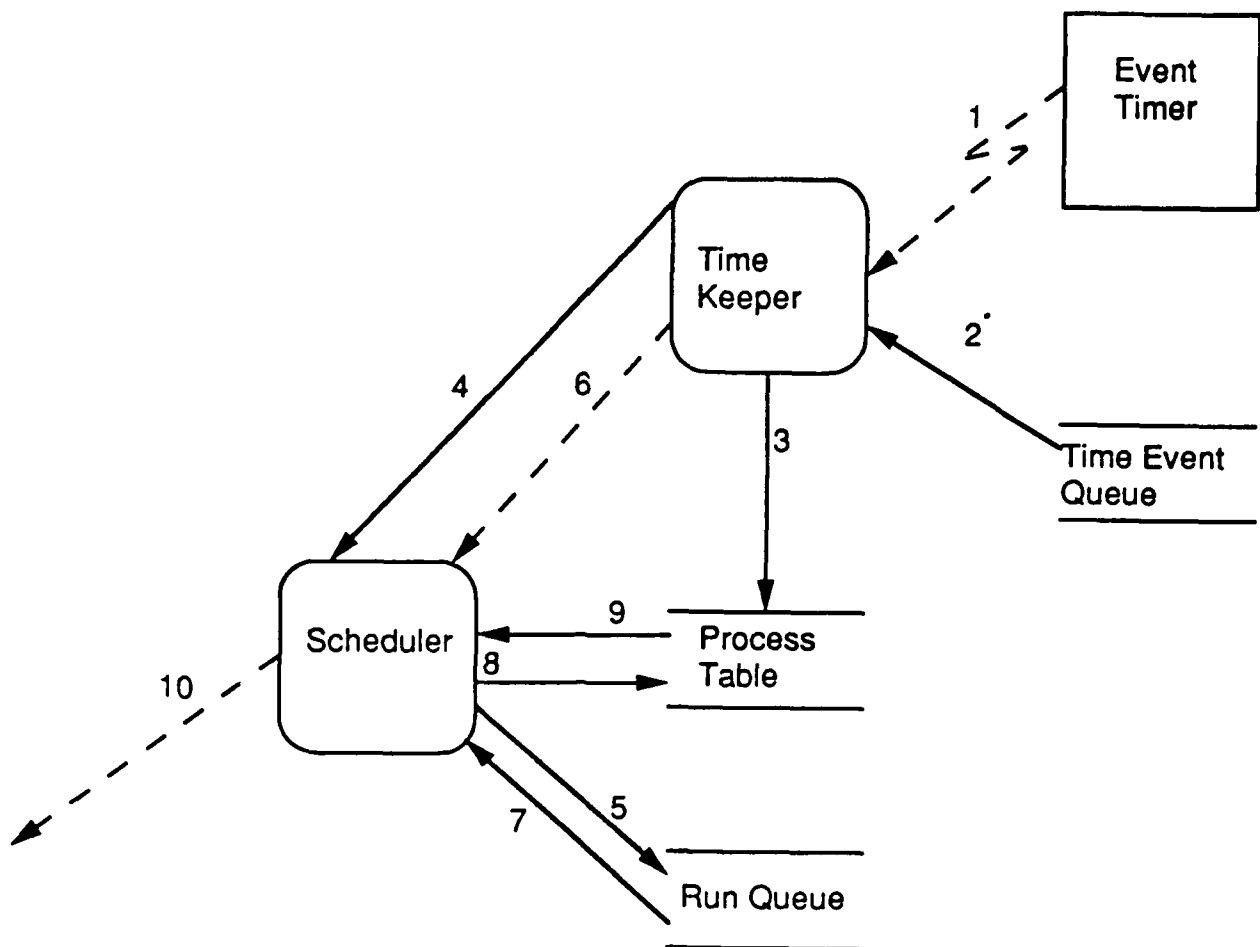
The following two examples illustrate the Scheduler's functioning. In Figure 21, the application has called the wait primitive to block its own execution for one second (there are no other pending events in this example). Then, Figure 22 shows the activities that occur when the wait timeout occurs and the waiting process is selected for execution.

³Note that two processes executing on the same processor cannot become blocked simultaneously.



1. Application issues a call to *wait (one_second)*.
2. An *insert_event* call is made to create the event.
3. Update the time event queue.
4. Configure the event timer to generate an interrupt in one second.
5. Invoke the Scheduler to block the waiting process and select the next process to run.
6. Remove the waiting process from the run queue.
7. Get the next process to run.
8. Save the context of the waiting process.
9. Restore the context of the next process run.
10. Scheduler transfers control to the resumed process.

Figure 21: Application Blocks



1. The wait timeout expires, and generates an interrupt.
2. The *event_interrupt_handler* within the *time_keeper* fields the interrupt and pulls the event at the head of the time event queue.
3. The process table is modified to reflect the occurrence of the event.
4. The *event_interrupt_handler* inserts the process into the Scheduler.
5. The Scheduler places the process back in the run queue.
6. The *event_interrupt_handler* returns control to the Scheduler (rather than the interrupted process)
7. The waiting process is now at the head of the run queue and selected to run.
8. Saved the context of the currently running process.
9. Restore the context of the waiting process.
10. Scheduler transfers control back to the waiting process.

Figure 22: Application Unblocks

12.1. Current Running Process

Visible value for Kernel primitives to use as identity of invoking process.

12.1.1. PDL

```
Current running process: process identifier
```

12.2. Process Run Queue

The process run queue is structured as a series of run queues, one for each legal process priority. At each priority is a singly linked list of suspended processes, where the link to the next process in that queue is embedded in the process table.

The head pointer points to the first process eligible to run at each priority. The tail pointer points to the last process eligible to run at each priority. The error pointer is an auxiliary tail pointer that points to the last process with a pending error eligible to run. Expressed as a record, the process run queue would look like:

```
Type run_queue_entry is record
  head: process_identifier := null;
  tail: process_identifier := null;
  error: process_identifier := null;
End record;
run_queue: array (ST.priority) of run_queue_entry;
```

This arrangement is not implemented in this version because of the inefficiencies of the compiler-generated code. Instead, the structure is represented by three arrays, each accessing a single component:

```
run_queue_head: array (ST.priority) of process_identifier
:= (others => null);
run_queue_tail: array (ST.priority) of process_identifier
:= (others => null);
run_queue_error: array (ST.priority) of process_identifier
:= (others => null);
```

Thus, the final incarnation of the process run queue is shown in Figure 23.

12.3. Get_next

This algorithm assumes there will always be a process to run somewhere in the run queue. This assumption is assured by the presence of the time_burner process with a priority lower than any user priority.

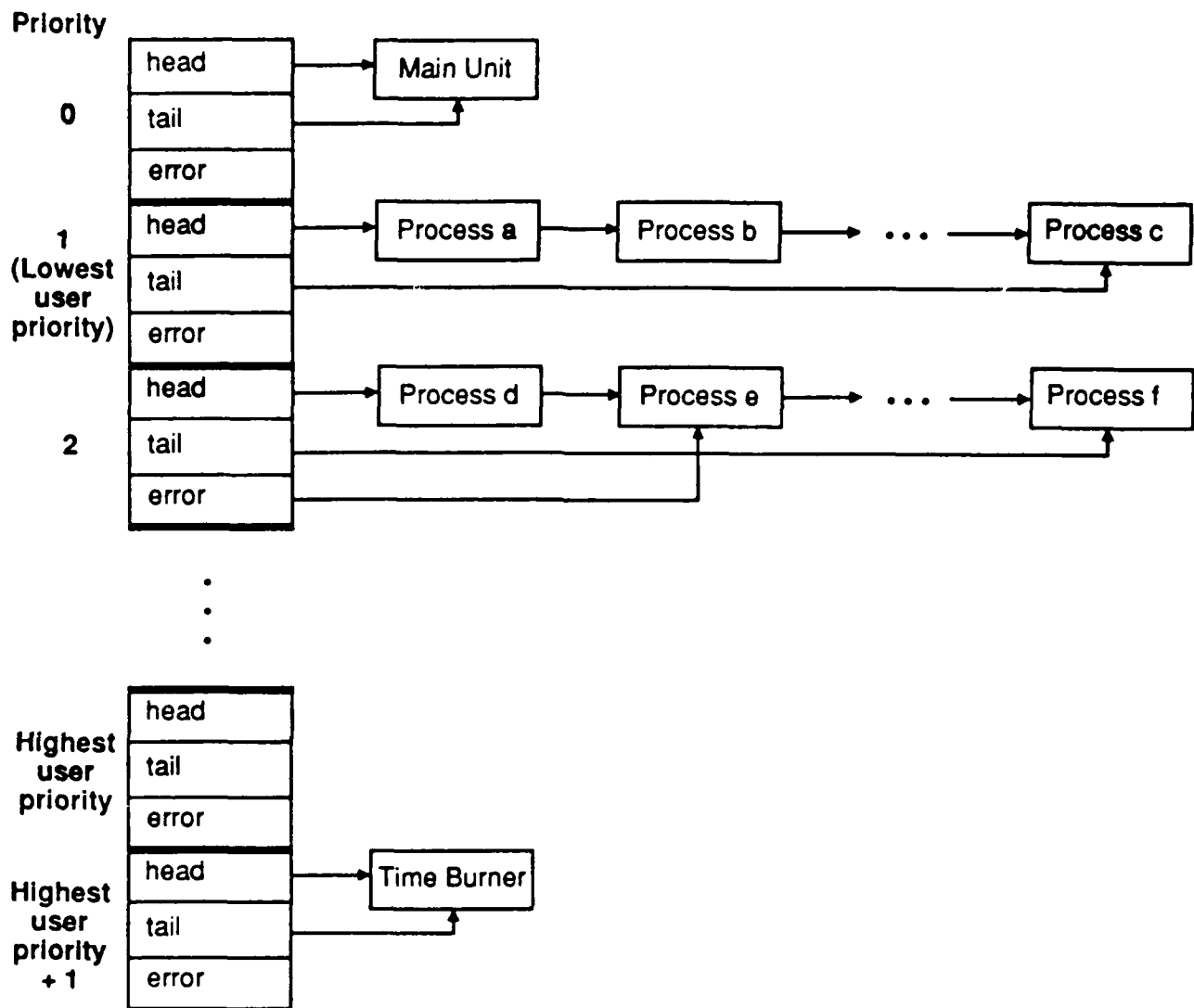


Figure 23: Run Queue

12.3.1. Interface

get_next return next process eligible to run

12.3.2. PDL

```
While we have not yet found a process eligible to run loop
  goto the next priority level
end loop
return the process with the highest priority
```

12.4. Initialize

12.4.1. Interface

initialize (initial number of processes)

12.4.2. PDL

```
Create all the internal Kernel processes
Log the main unit as the current running process
Place the Main Unit in the run queue
Mark the Main Unit as running (which it is)
```

12.5. Insert_process

12.5.1. Interface

insert_process (process identifier)

12.5.2. PDL

```
Case current process state
When suspended | running | dead =>
  Null...in the first two cases,
    the process is already in the run queue;
  in the last case,
    the process can not be run again
When blocked =>
  If the process is not in an error state then
    If the priority level of the process is empty then
      Insert process at head of its priority level
      Adjust the starting search location (if needed)
    Else the priority level is occupied
      Insert process at tail of its priority level
  Else
    If the priority level of the process is empty then
      Insert process at head of its priority level
    Insert process at error pointer of its priority level
```

```

        Adjust the starting search location (if needed)
    Else if the last process at this level is in an error state then
        Insert process at tail of its priority level
        Insert process at error pointer of its priority level
    Else...the error pointer is in the middle of the queue
        Insert process at error pointer of its priority level
    End if
End if
Mark the process as suspended
End case

```

12.6. Remove_process

If the head and tail pointers access the same object, the tail pointer will be left pointing at a dequeued object. This is not a problem, since the insertion algorithm is driven off the head pointer and performing the "proper" maintenance is not worth the run-time penalty.

When a process is running, it is no longer in an error state, because if it was in an error state, that state was cleared when the process was selected to run. Thus, there is no need to check the error pointer when a process blocks. *Schedule* performs the needed maintenance prior to resuming the process.

12.6.1. Interface

```

remove_process (pid of process to remove,
               new state of process after its removal)

```

12.6.2. PDL

```

Case current process state
When running =>
    Adjust queue head to point to next process (since the process must
    be at the head of the queue)
    Remove process from queue
    Change the process state
When suspended =>
    Loop thru queue looking for process
    Adjust queue head to point to next process
    Update queue tail
    Update error pointer
    Remove process from queue
    Change the process state
When blocked | dead =>
    Null...in the these cases, the process is not in the run queue
End case

```

12.7. Schedule

12.7.1. Interface

```
procedure schedule (new priority for caller,  
                   new preemption for caller,  
                   new state for caller)
```

12.7.2. PDL

```
If called from an interrupt handler then  
    Return...the interrupt encapsulation will handle the  
        return correctly  
End if  
  
Save the new preemption  
Case on new process state is  
When running =>  
    Null...this is not a legal invocation of the Scheduler,  
        so ignore it  
When suspended => the default case  
    If the new priority is different from the current priority then  
        Remove_process from the run queue  
        Update the process's priority  
        Insert_process back into the run queue...  
            this marks the process as suspended  
    Else  
        Mark the process as suspended  
    End if  
When blocked =>  
    Remove_process from the run queue  
    Mark the process as blocked  
    Update the process's priority  
When dead =>  
    Remove_process from the run queue  
    Mark the process as dead  
End case;  
  
Get_next process to run  
Schedule_slice_event  
If chosen process is the current running process then  
    null...no context change  
Else  
    If the tool interface is enabled  
        Log the process attributes for old process  
        Log the process attributes for new process  
    End if  
    Switch from the current running process to the chosen process  
End if  
If no Kernel exception is pending for the process to run  
    End atomic
```

```

Else...a Kernel exception is pending
  End atomic
  Raise the exception
End if

```

12.8. Schedule_ih

This entry is used exclusively by the interrupt encapsulation mechanism for returning from preemptive interrupts to the scheduler.

12.8.1. Interface

```

schedule_ih

```

12.8.2. PDL

```

Save the new preemption
Mark the process as suspended

Get_next process to run
Schedule_slice_event

If chosen process is the current running process then
  null...no context change
Else
  If the tool interface is enabled
    Log the process attributes for old process
    Log the process attributes for new process
  End if
  Switch from the current running process to the chosen process
End if
If no Kernel exception is pending for the process to run
  Return from interrupt level to user level (i.e. end atomic)
Else...a Kernel exception is pending
  Return from interrupt level to user level (i.e. end atomic)
  Raise the exception
End if

```

12.9. Schedule_slice_event

12.9.1. Interface

```

schedule_slice_event (next process to run,
                      current running process)

```

12.9.2. PDL

- (1) If time slicing is enabled then
- (2) If the scheduler was entered by a slice expiring then
 - If next process to run is slicable then
 - Setup a slice event for the process
 - End if
 - Else, some other action caused the scheduler to be entered
- (3) If next process was the last process to run then
 - no slice operations are required
- (4) Else, a new process is going to run, so
 - Cancel the pending slice event for the old process
- (5) If the new process is preemptable then
 - Setup a slice event for the new process
- End if
- End if
- End if

1. If time slicing has not been enabled (via an *enable_time_slicing* Kernel primitive call), then skip all of this. The overhead is a single test and branch.
2. Finally, consider the case of *slice_event_id* = *no_event*. In this case, one of two things has happened:

- a. A slice event has just expired. In this case, the slice event handler sets the *slice_event_id* to *no_event*, and invokes the Scheduler.
- b. A non-slicable process has just executed a blocking primitive. In this case, *slice_event_id* would already be *no_event*.

In all cases, if the chosen process is slicable, insert a new timeslice event into the event handler. Although it is possible that the next process to run is the current process, this is of no concern here. All that must occur is to insert a new slice event if the to-be-run process is slicable.

3. If the chosen process is the current running process, then do nothing. This means that:
 - a. There is a slice event (still) pending, so the slice has not expired.
 - b. The next process to be scheduled is the *current_running_process*, so the reason the Scheduler is running is because an interrupt routine has finished executing, and that routine did not elevate another process to a priority higher than the priority of the *current_running_process*.

Therefore do nothing. The time used by the interrupt routine is implicitly subtracted from the amount of time allocated to the timeslice.

4. If the process that has been selected is not the *current_running_process*, then cancel the pending timeslice for the current process. At this stage it is irrelevant whether or not the new process is slicable. All that matters is that the current process is about to be descheduled, so any pending timeslice event must also be canceled, since the process is giving way to a higher priority process.
5. If the new process that has been chosen to run is also slicable, then insert a

timeslice event for the to-be-run process. In this manner, each time a process unblocks, it starts with a new timeslice, irrespective of how much or how little of its previously allocated slice it used up.

12.10. Package Sch_debug

```
+-----+
|
+-----+
| print_run_queue_full |
+-----+
|
+-----+
| print_run_queue_brief |
+-----+
|
+-----+
| print_current_running_process |
+-----+
|
+-----+
```

This debug package gives visibility into the internal Scheduler run queue and as such, it is nested within the Scheduler package. It allows for printing of the entire run queue (either in full or just the schedule attributes) or just the current running process. This package iterates over the queue using ptb_debug entries to dump the needed data.

12.10.1. Interface

```
procedure print_run_queue_full;
```

or

```
procedure print_run_queue_brief;
```

or

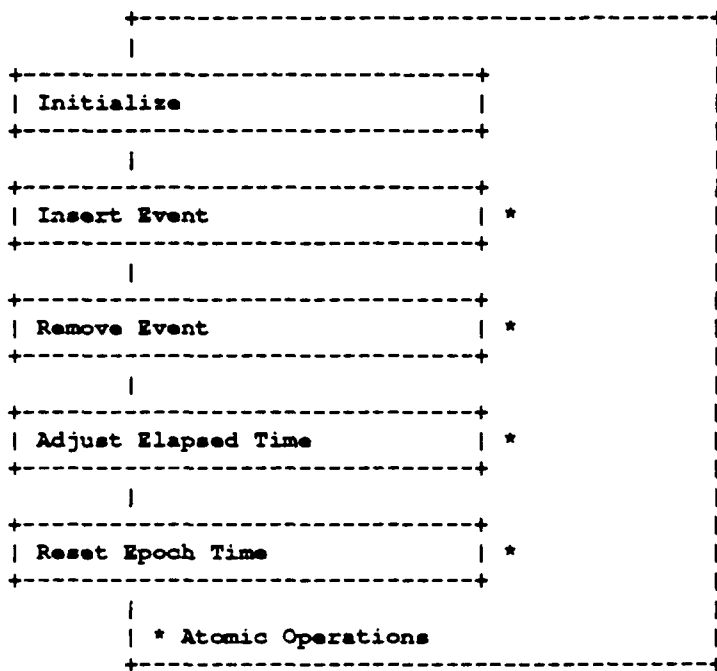
```
procedure print_current_running_process;
```

12.10.2. Sample output

```
!SCH_DEBUG: Dumping run queue (briefly)
!Process: TK_PROCESS
PTB: $$$ BEGIN SCHEDULE ATTRIBUTES $$$
PTB: state => SUSPENDED
PTB: priority => 1
PTB: preemption => ENABLED
PTB: block_time.high => 0
PTB: block_time.low => 0
PTB: unblock_time.high => 0
PTB: unblock_time.low => 12459720
PTB: $$$$ END SCHEDULE ATTRIBUTES $$$$
!Process: Time_burner
PTB: $$$ BEGIN SCHEDULE ATTRIBUTES $$$
PTB: state => SUSPENDED
PTB: priority => 11
PTB: preemption => DISABLED
PTB: block_time.high => 0
PTB: block_time.low => 0
```

```
PTB: unblock_time.high => 0
PTB: unblock_time.low => 0
PTB: $$$$ END SCHEDULE ATTRIBUTES $$$$
!SCH_DEBUG: End of Dump
```

13. Time Keeper



Time_keeper encapsulates all of the time related events:

- Alarms
- Claim timeouts
- Enable_time_slice timeouts
- Receive_message timeouts
- Send_message_and_wait timeouts
- Synchronize timeouts
- Wait timeouts

It is implemented using a timer and a pending event queue, ordered by absolute time of event occurrence (i.e., the next event to occur is at the head of the time event queue).

Each process may have at most two events pending simultaneously:

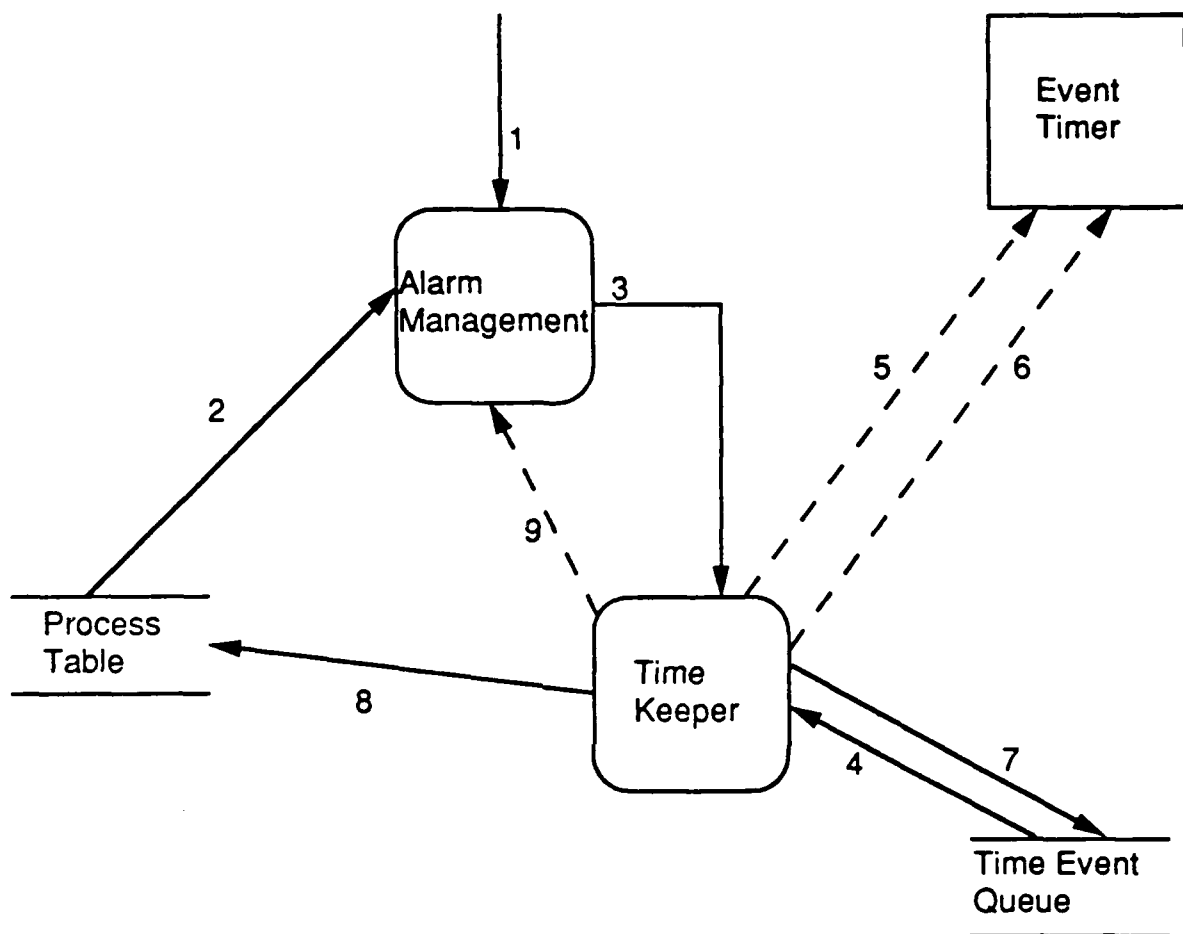
- alarm
- one of: claim timeout, receive_message timeout, send_message_and_wait timeout, synchronize timeout, and wait timeout

The event timer is:

- a countdown timer loaded with the number of ticks to delay until interrupt
- the timer counts down to 0 then generates an interrupt
- the maximum count down value is 32 seconds
- events greater than 32 seconds are divided into 32 second chunks (thus if an

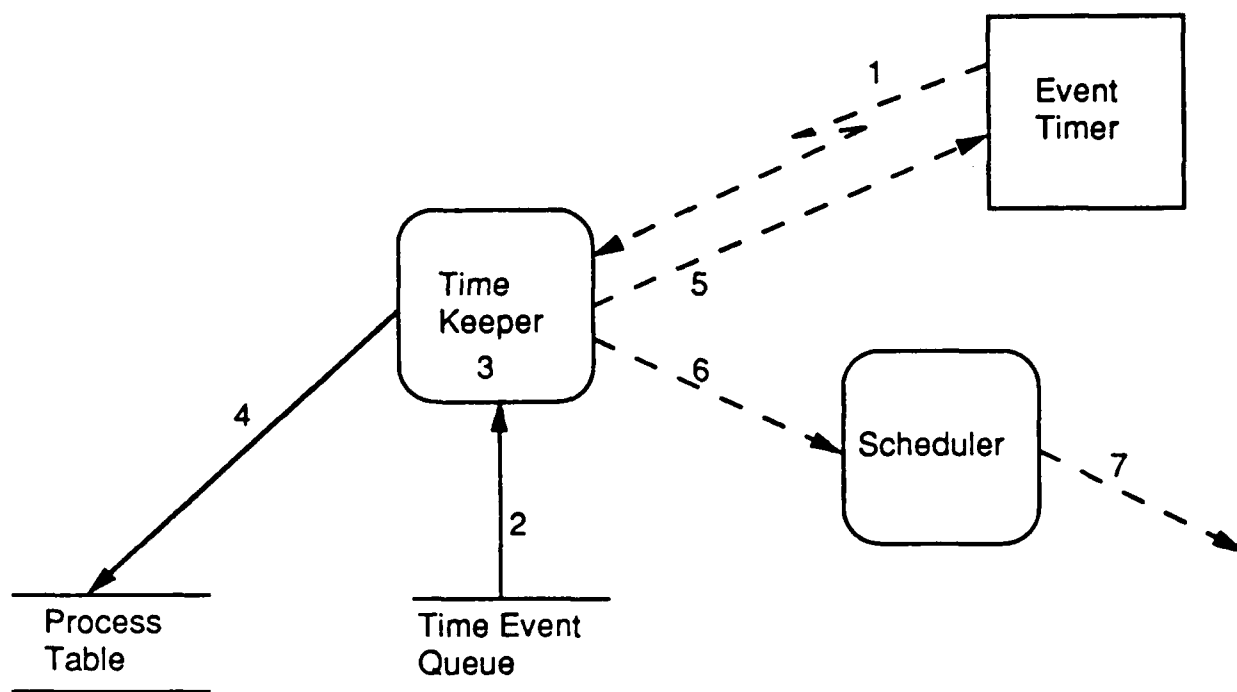
event is set to occur in 60 seconds, two interrupts occur: one at 32 seconds and another one ~28 seconds later)

In Figure 24 illustrates the situation where the inserted event replaces the event currently at the head of the time event queue. While Figure 25 traces the actions that occur when the event actually occurs.



1. The application issues a *set_alarm (one_second)*.
2. *Alarm_management* checks the current alarm status of the process (based on information in the process table) .
3. *Insert* an alarm event into the *time_keeper*.
4. Read the time event queue and determine that the alarm event should be the next event to occur.
5. The current event timer is canceled.
6. The event timer for the alarm is set.
7. The alarm event is enqueued in the time event queue.
8. The process table is updated to reflect the existence of the alarm event for the process.
9. Control returns to the *set_alarm*.

Figure 24: Setting an Alarm Event



1. The alarm timeout expires and the timer generates an interrupt.
2. The *event_interrupt_handler* fields the interrupt and examines the event at the head of the *time_event_queue*.
3. Since the event's time has passed, alarm event processing is performed.
4. The event's occurrence is logged in the *process_table*.
5. The event timer is configured for the next event in the time event queue (if such an exists).
6. The alarmed process is removed for the run queue and reinserted at its alarm resumption priority.
7. Scheduler resumes the alarmed process with the *alarm_expired* exception pending.

Figure 25: Event Expiration

13.1. Initialize

The initial allocation of the time event queue accounts for all possible events generated by all processes known to this node. The initial allocation is computed as follows:

$2 * (\text{maximum number of processes on this processor}) + 1$

13.1.1. Interface

`Initialize`

13.1.2. PDL

Create the time event queue
Install the event_interrupt_handler for the event timer

13.2. Insert_event

13.2.1. Interface

`insert_event` (new event,
 type of event,
 time of event,
 pid of associate process,
 pointer to enqueued event)

13.2.2. PDL

If the event's time has already passed or is zero then
 Process_event_immediately
 Return a null pointer (since the event wasn't enqueued)
Else
 Begin atomic
 If the time event queue is empty then
 Set the timer for new event
 Else
 If new event expires before the event at the
 time event queue head then
 Cancel the timer for the current pending event
 Set the timer for new event
 End if
 End if
 Enqueue the new event in the event queue
 Set pending_activity for the process reflecting event type
End atomic
Return pointer to enqueued event
End if

13.3. Remove_event

13.3.1. Interface

`remove_event` (identifier of event to remove)

13.3.2. PDL

```
Begin atomic
Delete the event
Grab the event at the head of the time event queue
If the deleted event was to expire before the event now at the
  head of the time event queue then
  Cancel the timer pending on the deleted event
  Set a timer for the event at the time event queue head
End if
Reset the pending_attributes flag for the process associated with
  the event
End atomic
```

13.4. Adjust_elapsed_time

13.4.1. Interface

`adjust_elapsed_time` (elapsed time adjustment)

13.4.2. PDL

```
Cancel any pending event timer
Begin atomic
  While there are more events in the time event queue loop
    Dequeue the next event
    Perform the adjustment
  End loop
  Set an event timer to expire immediately...thus processing all
  the events whose time has passed as a result of the adjustment
End atomic
```

13.5. Reset Epoch Time

13.5.1. Interface

`reset_epoch_time` (new time of day)

13.5.2. PDL

```
Cancel timeout for current pending event
Begin atomic
Compute increment = current time of day - new time of day
While there are more events in the time event queue loop
  Dequeue the next event
  Case time class of event
    When elapsed =>
      Add the increment to time in event
      Enqueue the modified event
    When epoch =>
      Enqueue the unmodified event
  End case
End loop
Adjust_epoch_time inside the Clock
Set an event timer to expire immediately...thus processing all
the events whose time has passed as a result of the adjustment
End atomic
```

13.6. Event_interrupt_handler

13.6.1. Interface

N/A

13.6.2. PDL

```
Acknowledge the interrupt
Read the current time
While the time event queue has events yet to process loop
  Dequeue the event at the head of the time event queue
  If the event has expired then
    Process_event
  Else
    Enqueue the unprocessed event
    Set the timer to expire at the time indicated by the event
    at the head of the queue
  Exit the loop
End if
End loop
```

13.7. Process Event

The processing for the expiration of a slice event is needed to allow the next process at this priority of the current_running_process to run. Removing and immediately reinserting a process has the effect of moving the process to the end of the run queue at its priority level.

13.7.1. Interface

process_event (event to process)

13.7.2. PDL

Case on the type of event to process

When an alarm has expired =>

 If the process is not blocked then

 Remove_process from the run queue

 End if

 Reset alarm pending parameters

 Insert_process into run queue at the alarm resumption priority

 Release any allocated datagram buffers

 Cancel any other pending activity (as below)

When a receive times out =>

 Reset the pending activity parameters

 Setup the timeout exception for propagation

 Place the process back in the run queue

When a semaphore claim operation times out =>

 Reset the pending activity parameters

 Setup the timeout exception for propagation

 Place the process back in the run queue

When a wait operation times out =>

 Reset the pending activity parameters

 Place the process back in the run queue

When an acknowledged send operation times out =>

 Reset the pending activity parameters

 Remove the message from the receiver's message queue

 Send a "NAK" message to the sender

 Delete datagram

 Free datagram

When time slice expires =>

 Reset the slice event parameters

 If the sliced process is running then

 Remove_process from the run queue

 Insert_process back into the run queue

 End if

End case

13.8. process_event_immediately

The processing for the expiration of a slice event is needed to allow the next process at this priority of the current_running_process to run. Removing and immediately reinserting a process has the effect of moving the process to the end of the run queue at its priority level.

13.8.1. Interface

`process_event_immediately (event to process)`

13.8.2. PDL

Case on the type of event to process

When an alarm has expired =>

 If the process is not blocked then

 Remove_process from the run queue

 End if

 Reset alarm pending parameters

 Insert_process back run queue at the alarm resumption priority

 Release any allocated datagram buffers

 Cancel any other pending activity (as below)

 Schedule

When a receive times out =>

 Reset the pending activity parameters

 Setup the timeout exception for propagation

When a semaphore claim operation times out =>

 Reset the pending activity parameters

 Setup the timeout exception for propagation

When a wait operation times out =>

 Reset the pending activity parameters

 Place the process back in the run queue

When an acknowledged send operation times out =>

 Reset the pending activity parameters

 Remove the message from the receiver's message queue

 Send a "NAK" message to the sender

 Delete datagram

 Free datagram

When time slice expires =>

 Reset the slice event parameters

 If the sliced process is running then

 Remove_process from the run queue

 Insert_process back into the run queue

 End if

End case

13.9. Time Event Queue

The time event queue is the doubly linked structure that maintains the list of pending events.

The characteristics of the time event queue are:

- Ordered by expiration time, with each event in epoch time format
- Insert places event in its correct position in the queue (this is a high frequency operation)
- Random deletion of queue objects occurs frequently as unblocking events occur and the associated timeouts are canceled

- The structure is periodically reorganized (whenever the user modifies the local processor clock)
- Each entry in the queue is a record structure containing:
 - Kind of event: alarm timeout, claim timeout, receive timeout, send message and wait timeout, timeslice, or wait timeout
 - Time class: epoch or elapsed
 - Expiration time
 - process identifier

Together, all these facets yield the structure shown in Figure 26.

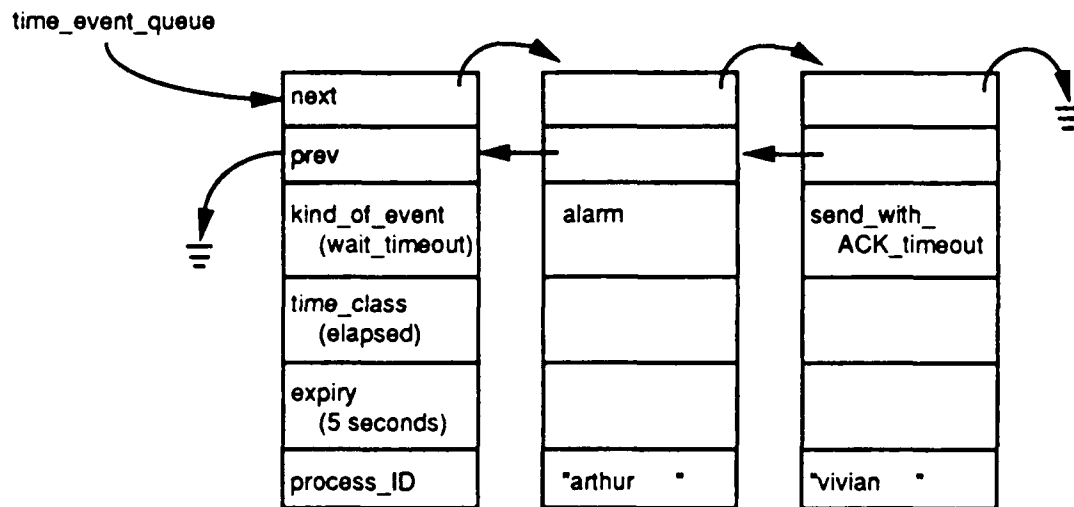
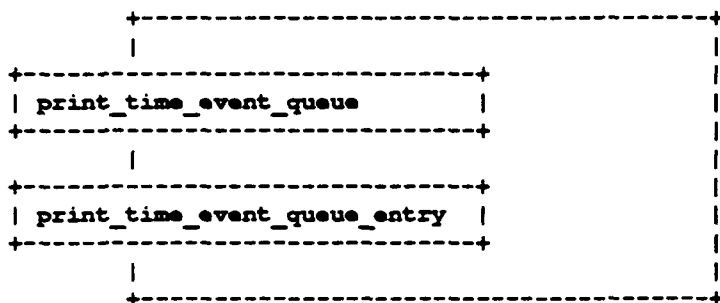


Figure 26: Time Event Queue

13.10. Package time_keeper_debug



This package is nested within the time_keeper package and allows for the diagnostic printing of the entire time event queue or individual entries within the time event queue.

13.10.1. Interface

`print_time_event_queue`

and

`print_time_event_queue_entry (event identifier)`

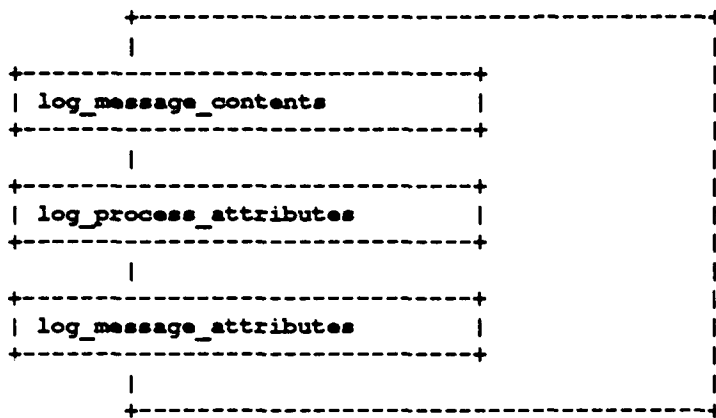
13.10.2. Sample Output

```

TK: ##### BEGIN DUMP OF TIME EVENT QUEUE #####
TK: *****
TK:  kind_of_event => WAIT_TIMEOUT
TK:  time_class   => ELAPSED
TK:  expiry.high  => 0
TK:  expiry.low   => 60445590
TK:  process name => TK_PROCESS
TK: *****
TK: ##### END DUMP OF TIME EVENT QUEUE #####

```

14. Tool Logger



This package is responsible for collecting the monitored attributes, formatting the data into a tool interface message (shown in Table 4), and linking the message into the message queue of the appropriate tool process. To accomplish the logging activity, this package is called from strategic points in the Kernel, namely:

- CM.send_message and CM.receive_message: to log message attributes and message contents.
- SCH.schedule, SCH.schedule_jh, and PAM.kill: to log process attributes.

If for any reason, a logging cannot take place, no tool interface message is formatted and no exception is generated (the request is simply ignored).

14.1. Log_process_attributes

The time is always measured as the first action immediately after it has been verified that logging should happen.

14.1.1. Interface

```
procedure log_process_attributes
  (process whose attributes are to be logged)
```

14.1.2. PDL

```
If it is ok to log the needed attributes then
  Log the time the tool message formatting began
  Format the tool message header
  Collect the process attributes
  Copy the process attributes to the tool message
  Enqueue the tool message on the tool process's message queue
  If the tool process is pending on a message then
    Insert the tool process into the Scheduler
  End if
End if
```

14.2. Log_message_attributes

The time is always measured as the first action immediately after it has been verified that logging should happen.

14.2.1. Interface

```
Procedure log_message_attributes
  (process whose attributes are to be logged,
   sender of the message being logged,
   receiver of the message being logged,
   tag of the message being logged,
   length of the message being logged)
```

14.2.2. PDL

```
If it is ok to log the needed attributes then
  Log the time the tool message formatting began
  Format the tool message header
  Collect the message attributes
  Copy the message attributes to the tool message
  Enqueue the tool message on the tool process's message queue
  If the tool process is pending on a message then
    Insert the tool process into the Scheduler
  End if
End if
```

14.3. Log_message_contents

The time is always measured as the first action immediately after it has been verified that logging should happen.

14.3.1. Interface

```
Procedure log_message_contents
  (process whose attributes are to be logged,
   length of message being logged,
   text of message being logged)
```

14.3.2. PDL

```
If it is ok to log the needed attributes then
  Format the tool message header
  Copy the message contents to the tool message
  Enqueue the tool message on the tool process's message queue
  If the tool process is pending on a message then
    Insert the tool process into the Scheduler
  End if
End if
```

14.4. prepare_to_log

This is an internal procedure. It is always executed as the first activity of any logging activity.

14.4.1. Interface

```
Procedure prepare_to_log (process whose attributes are to be logged,
                          what process information is being collected,
                          boolean telling the logger to proceed,
                          datagram to hold the tool message)
```

14.4.2. PDL

```
If the tool interface is enabled for this process      and
   there is a tool process selected for this attribute and
   there is room in the message of the tool process    and
   there is a datagram available                        then
  it is ok to log the attribute information
End if
```


IV. Communication Subsystem

This Part describes the interfaces, algorithms, and data structures that implement the network.

1. Communication Overview

All messages sent by the application and by the Kernel are stored in Kernel buffers, called "datagram buffers." The datagram buffers are maintained in the shared memory common to the Kproc and Nproc. The datagram buffers are initially allocated from the shared memory by the Nproc. Following this, the datagram buffers are readable and writable by both processors. To satisfy the requirements of the *Kernel Facilities Definition*, the following programming assumptions have been used:

1. The `send_message` Kernel primitive must be non-blocking (Section 19.1.1 of the *Kernel Facilities Definition*). The network may be busy at the time an application process wishes to send a message. To prevent the process from blocking, the application buffer must be copied immediately into a Kernel datagram buffer.
2. When a message is received by a node, the process to which the message is addressed might not be pending on a `receive_message` Kernel primitive (Section 19.1.3 of the *Kernel Facilities Definition*). Consequently, the incoming message must be stored in a Kernel buffer, and later copied into the application buffer.
- 3.

All messages whose destination process is on a remote processor must be copied twice – once on the originating processor from the application buffer to a Kernel buffer, and again at the remote node from a Kernel buffer to an application buffer.⁴ The flow of data are diagrammed in Figure 27. Certain optimizations reduce the number of times that data is copied when both the sending and receiving process are located on the same processor.

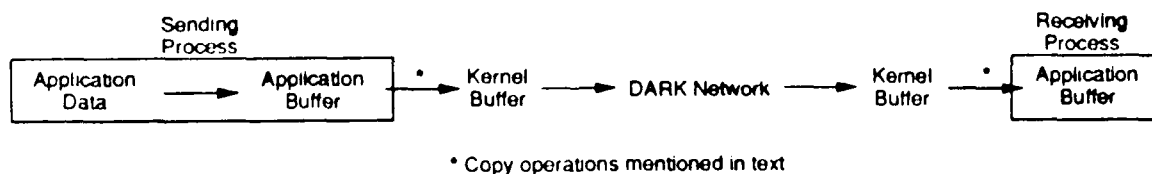


Figure 27: Data Flow Through the Kernel and Network

Once a message has been copied into a datagram buffer by the Kernel and enqueued into the Nproc output queue, the Nproc assumes all responsibility for transferring the message to the remote node.

The remainder of this chapter covers the internal Kernel abstractions necessary to

⁴The messages must be placed in an application buffer by the application prior to the call to the `send_message` or `send_message_and_wait` Kernel primitives, and in a Kernel buffer as the message is initially read off the network, prior to notifying the Kproc of an incoming message. These transactions are not considered in the count of copy operations.

implement the Kproc/Nproc communication and interaction. In particular, Sections 8.1 and 8.2 outline and diagram the transmission of a message from one Kproc to another via the network.

1.1. Design Decisions

A number of design decisions have been made concerning the implementation of the communications algorithms. These are:

1. It is assumed that the Kproc and Nproc have some mechanism of synchronization via semaphores. Two mechanisms of processor synchronization are available:
 - a. Using the MC68020 CAS instruction to set semaphores used by the two processors. This mechanism is the one currently in use, although it is machine-specific.
 - b. A hardware-independent algorithm written entirely in Ada, derived from the algorithms in [Raynal 86]. This algorithm has been tested and executes correctly, although it is slightly slower than the machine-specific version. This version is not being used, although it is supplied as commentary in the body of LLH.⁵

These semaphores will be used to lock resources shared by the Kproc and Nproc, such as free lists and message queues (datagram buffers are stored in message queues; the queues must be locked before any enqueueing or dequeueing can be performed).

2. There is a single set of datagram buffers available to both the Nproc and the Kproc, and it is allocated in the megabyte of shared memory common to both processors. This set of buffers is used both for sending messages from the Kproc to the network, and by the Nproc as it receives messages from the network.
3. The set of datagram buffers comprises three pools of buffers (described in greater detail in Section 2.3). Each pool has its own free list. Whenever the Kernel allocates a buffer, it looks in the free list of the appropriate size.
4. Only the actual number of bytes in the application message (plus the necessary Kernel overhead) are transmitted on the network, even if a buffer larger than the application message is used.
5. Each message originating on a node (a Kproc/Nproc pair) is sent completely before transmission of another message originating on the same node is begun.
6. Messages that are passing *through* a node are interleaved with any message that may be originating on that node. This does not present a problem, since each 32 bits of data that are transmitted have the sender and receiver encoded in the top 16 bits. The node that is "downstream" of the

⁵The short name LLH is used as an Ada code abbreviation for the package `low_level_hardware`. For a complete list of all the package short names that are used in this document, please refer to Appendix H.

sending node is able to identify the sender and recipient of each 32-bit packet of a datagram.

7. Local optimizations are performed for local transmission of messages (Section 10.1.29 of the *Kernel Facilities Definition*). The application program need not know the location of the destination of the message; however, the Kproc speeds message transmission when it is known that the source and destination processes are both resident on the same processor.
8. Implementation of the buffers is via doubly linked lists. Allocation from the free list may be arbitrary (i.e., it does not matter which free pool is used, so long as the buffers in it are of the appropriate size), although the send and receive queues are absolutely First-In First-Out (FIFO). There are *no* message priorities (Section 19 of the *Kernel Facilities Definition*).
9. The algorithms used for datagram buffer allocation, enqueueing, dequeueing, and deallocation are the same on both the Kproc and the Nproc. Wherever needed, semaphores are used to guarantee the atomicity of operations.

Although this design is specified for a single Kproc and a single Nproc comprising each processor node, the algorithms that are described below are sufficiently flexible to allow for multiple Kprocs to be associated with a single Nproc.

2. Data Structures

The following data structures are used to implement data transfer using the Kproc and Nproc model that we have established.

2.1. Datagram Data Structures

A datagram is the Ada type used by the Kernel to keep track of messages. Datagrams are allocated from the free lists with the subroutine `alloc_dg` (Section 4.6), enqueued into the receive and transmit queues with the `enqueue` subroutine (Section 4.2), removed from these queues with the `dequeue` subroutine (Section 4.3), and returned to the free lists with the `free_dg` subroutine (Section 4.7). Other subroutines support the ability to get the first element from a queue without dequeuing it (`get_first`, described in Section 4.4), and to delete an arbitrary element from a queue (`delete`, described in Section 4.5).

Datagrams are never referenced directly, but are instead always accessed through datagram pointers. For consistency, the datagram that is sent from one process is identical to the datagram that is received by another. The Nproc knows nothing about the format of messages other than their size, source, and destination. The datagram contents and access mechanisms are described below.

2.2. Datagram_Pointer

The type `datagram_pointer` is defined as follows:

```
type datagram_pointer is access datagram;
```

A `datagram_pointer` is simply a pointer to a datagram. All references to datagrams within the Kernel are made with this access type.

2.3. Datagram_Class

Datagrams are divided into four distinct classes:

```
type datagram_class is (small, large, kernel, queue_head);
```

The different datagram classes are used for a number of reasons:

1. A significant space savings can be effected by allocating a small-sized datagram to the application when a small message needs to be transmitted, and only allocating a large datagram when larger application messages need to be sent.
2. Although variant records would be the most natural way to accomplish this, the code that our compilers generated was abominably inefficient. Therefore, it was decided to use `unchecked_conversion` to coerce the required number of datagram pointer types into a single type called

`datagram_pointer`. To distinguish between the different sizes of datagrams associated with the pointers, a field called `class` of type `datagram_class` is present in each datagram.

3. Since variant records are *not* being used, rather than have an additional type of structure for queue headers, a class called `queue_head` is used to distinguish "real" datagrams from datagram queue headers. A queue header accesses a datagram that contains no data buffer (or rather, has a buffer of zero size), and is used simply as the access mechanism for a queue. The number of queues in use on a processor is determined at runtime as processes are created – there is one queue for every process, in addition to the free queues and input and output queues.

The data buffer sizes associated with the various datagram classes (plus some additional space for Kernel encapsulation of datagrams) are:

<code>large</code>	<code>CG.maximum_message_size</code> bytes (rounded up to an integral number of 16-bit words).
<code>small</code>	One tenth of <code>CG.maximum_message_size</code> bytes (rounded to an integral number of 16-bit words). If this fraction is smaller than 32 bytes, the smaller buffer pool is not used. If this fraction is larger than 128 bytes, the value of 128 bytes is used.
<code>kernel</code>	<code>NC.net_entry'size</code> bytes (rounded to an integral number of 16-bit words). This buffer size is used to ensure that an adequately sized buffer is available to initialize the DARK network, even if the application specifies that <code>CG.maximum_message_size</code> is zero bytes.
<code>queue_head</code>	No data buffer is allocated for this class of datagram, because it is only used as a queue header, and not for data transmission.

2.4. Datagram

A datagram is divided into a local optimization component (which contains information for processing locally optimized messages), a header component (which describes the contents of the data, contains the various maintenance information, etc.) and a data component (which contains the actual body of the application or Kernel message). The definitions of these two components are:

```

type datagram_header is
  record
    next :          datagram_pointer;
    prev :          datagram_pointer;
    class :         datagram_class;
    buffer_size :   buffer_range;
    msg_count :     hw_long_natural;
    semaphore :     hw_long_integer;
    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;

type copy_action_types is (no_copy_necessary,
                           copy_from_datagram_buffer,
                           copy_from_senders_buffer);

type local_optimization_record is
  record
    do_optimization:  boolean := false;
    copy_action:      copy_action_types := no_copy_necessary;
    copy_from_address: hw_address;
  end record ;

type datagram is
  record
    local:  local_optimization_record;
    header : datagram_header;
    buffer : data_buffer(1 .. CG.message_length_type'last);
  end record;

```

The Local component is used only when the receiver and sender are on the same processor. There are three pieces of information referenced by Local:

do_optimization	A flag used to indicate the message has been locally sent and should be processed appropriately.
copy_action	Indicates one of three actions the Kernel should take when copying the message to the receiver's buffer. The action is affected by the level of optimization that has taken place and reflects the condition of the receiver when the sender issued the call to send. No_Copy_Necessary is set when the receiver was waiting and the message was copied directly from the sender's buffer to the receiver's buffer. A copy in and out of the datagram buffer has been avoided. Copy_From_Senders_Buffer is set when the message was not able to be copied into the receiver's buffer and the sender issued a

Send_Message_And_Wait. This represents the next best level of optimization. The copy is still made directly into the receiver's buffer from the sender's buffer, but only after the receiver is ready. **Copy_From_Datagram_Buffer** is set when it is not possible to copy immediately into the receiver's buffer and the sender issued a **Send_Message**. In this case, the sender could not be blocked and the message had to be copied into the buffer of the datagram. At some point when the receiver is ready, the Kernel will copy the message into the receiver's buffer from the datagram.

copy_from_address

Address of sender's buffer when the **Copy_Action** is **Copy_From_Senders_Buffer**. At the point when the receiver is ready to receive the optimized message the Kernel uses this address to copy to the receiver's buffer from the blocked sender's buffer.

At best, during local optimization the transmission overhead of the network can be avoided and, in some cases, copying into the buffer of the datagram.

The description of the various fields within the header component are as follows:

next	This field contains the pointer to the next datagram in the queue and is used to maintain the doubly linked lists. This field and a number of fields following are not transmitted across the network. They are used only on the node that actually "owns" the datagram and are not considered part of the message that is shipped between processors.
prev	This field contains the pointer to the previous datagram in the queue and is used to maintain the doubly linked lists. This field is not transmitted across the network and is maintained locally by each processor node.
buffer_size	This field is only used in queue headers and is ignored in datagrams. It contains the size of the data buffer <i>available</i> to the application in this datagram (and <i>not</i> the number of bytes in the application message). This field is not transmitted across the network and is maintained locally by each processor node.
msg_count	This field is only significant in queue headers and contains the number of messages presently in the queue. This field is not transmitted across the network and is maintained locally by each processor node.
semaphore	This field is used by both queue headers and datagrams, but in different ways. • In the queue headers, this field contains the interprocessor semaphore for locking the queue for exclusive use. It contains either LLH.free (indicating that the queue is available for use) or LLH.busy (indicating that the queue is presently busy, and is being modified). In queue headers, this field is manipulated with the LLH.P and LLH.V subroutines, and ensures that the Kproc and Nproc do not manipulate the queues simultaneously. • In individual datagrams, this field contains either LLH.free (indicating that the datagram is free from association with

any queue), or `LLH.busy` (indicating that the datagram is currently in a queue). In the former case, any attempt to delete the datagram from a queue will be ignored. In the latter case, any attempt to re-enqueue the datagram will be ignored. In datagrams, this field is manipulated by simply setting the appropriate value while in an atomic region.

This field is not transmitted across the network and is maintained locally by each processor node.

<code>message_length</code>	The number of bytes in the message text, as supplied by the application. This value will always be less than or equal to the <code>buffer_size</code> field. This field, and all following fields are transmitted across the network, and are considered part of the message that is sent and received.
<code>operation</code>	This field contains one value from the enumerated type <code>DGG.kernel_operation</code> for both Kernel to Kernel (e.g., <code>DGG.kernel_message</code> or <code>DGG.init_protocol_message</code>) or application (e.g., <code>DGG.blind_send</code> or <code>DGG.acknowledged_send</code>) messages, depending on the type and originator of the message. This item is generated by the Kernel.
<code>remote_timeout</code>	This field is only used when a <code>send_message_and_wait</code> is transmitted. The value specified in this field is the elapsed time as calculated on the <i>sending</i> process.
<code>sender</code>	The <code>process_index</code> of the process that is sending this message. In the case of <code>DGG.nak_process_dead</code> and related messages, the sender field will be what would have been the <code>process_index</code> of the process that is being marked as dead. Because the sender field contains both a process number and a processor number, the sender field uniquely identifies a process within the DARK network.
<code>receiver</code>	The <code>process_index</code> of the process that is supposed to receive the datagram. Because the receiver field contains both a process number and a processor number, the receiver field uniquely identifies a process within the DARK network.
<code>message_tag</code>	In the case of an application-generated message, this field contains an application-supplied message identifier that is passed to the application. In the case of a Kernel-generated message, this field contains a value from the enumerated type <code>DGG.kernel_tag</code> (i.e., <code>ack</code> , <code>nak</code> , <code>kill_process</code> , etc.), which determines the action to be taken upon receipt of the datagram.
<code>message_id</code>	A sequence number assigned by the Kernel for use with <code>send_message_and_wait</code> and the associated Kernel-to-Kernel reply.
<code>checksum</code>	A checksum of the Kernel datagram message. This checksum is supplied by the transmitting Nproc and is verified by the receiving Nproc. Presently, the checksum is not calculated, and is always supplied as 0 – this field is supplied for future network enhancements.

The array bounds of the data buffer field are defined to be 1 .. `CG.message_length_type`'last. No datagram is ever actually allocated with a

data buffer of this size - this range is included to allow for datagrams of size `CG.maximum_message_size` and smaller (as described in Section 2.3) to be allocated and coerced, using `unchecked_conversion`, into this type. Because of the very large upper bound on the array range of `data_buffer`, no boundary check exceptions are ever raised (and it is assumed that the subroutines in `datagram_management` are sufficiently robust for this not to be a problem).

3. Semaphores and Atomic Regions

The Kproc and Nproc subprograms for datagram manipulation use both interprocessor semaphores and atomic regions to protect Kernel data structures. Interprocessor semaphores are manipulated with the subroutines `LLH.P` and `LLH.V` (these subroutines implement a classical Dijkstra P/V semaphore system). Atomic regions are maintained by the subroutines `LLH.begin_atomic` and `LLH.end_atomic`. Each mechanism has a different purpose.

The semaphores used in the message queue headers are designed to prevent simultaneous access of the datagram queues by both the Kproc and the Nproc. Since the amount of time required to access the queue headers is short, `LLH.P` is a blocking (i.e., busy/wait) operation – that is, the processor that calls `LLH.P` loops until the semaphore becomes available.

Atomic operations are used in conjunction with semaphores to prevent queue operations from being interrupted by the actions of an interrupt service routine operating on the same processor. In this way, only a single access to the queue may be made on a single processor.

In all the subroutines described in Section 4, both atomic regions *and* semaphores are used. The primary reason for this is best illustrated with the following two scenarios:

Scenario 1:

- a. While executing a `BIO.send_process_datagram`, the Kproc enqueues a datagram being sent locally to a process receive queue.
- b. At the same time, the Nproc receives a datagram from a different processor node targeted for the same local process. The Nproc interrupts the Kproc.
- c. The Kproc interrupt service subroutine enqueues the incoming datagram into the process receive queue.

If atomic regions are not used to surround the enqueue operations, the Kproc interrupt service routine (step c) could begin executing while the main-line Kproc subroutine (step a) was still accessing the queue. Either the interrupt service routine will block forever (since the semaphore guarding the queue is presently claimed by the main-line Kproc code), or the interrupt service routine would corrupt the queue currently being manipulated by the main line code (if the semaphore only precludes access by the Nproc).

Scenario 2:

- a. The Kproc, executing a `BIO.send_kernel_datagram`, dequeues a datagram from the free queue so that it may build a Kernel message.
- b. At the same time, the Nproc receives a packet from an incoming datagram. It also attempts to dequeue a datagram from the free queue.

If interprocessor semaphores are not used to surround the dequeue operations, both the Kproc and Nproc could conceivably be accessing the same free queue at the same time, each corrupting the accesses of the other.

Other scenarios can easily be envisioned that couple the conflicting actions of all of the datagram manipulation subroutines described in Section 4. It is necessary to use both atomic regions (to prevent datagram corruption via an interrupt service routine accessing the same datagram queue) and interprocessor semaphores (to prevent corruption by the other processor accessing the same datagram queue) to lock datagram queues for subsequent manipulation. Only by using both mechanisms on the Kproc can the integrity of the datagram queues be guaranteed.

On the Nproc, it is not necessary to use both mechanisms; interprocessor semaphores suffice. In the Nproc, some datagram queues are accessed only by interrupt service routines, and others by only main-line (i.e., non-interrupt level) code. Because of this, atomic sections are not needed (since interrupt service routines are themselves not interruptible). See Section 7 for more details.

4. Datagram Management

The following subroutines are used to manage the datagrams within the Kernel. These subroutines are used to allocate datagrams from free queues, enqueue them into input and output queues, and return them to the free queues when the Kernel is done with them. The application program never sees these subroutines, nor need it ever be aware that datagrams are used for message transmission.

4.1. new_queue

The `new_queue` subroutine is used to create a new datagram queue header. This subroutine must be called before any enqueue operations can be performed.

The queue is initialized with a `msg_count` field of 0 (i.e., no messages), a `class` field of `queue_head` (i.e., this datagram only serves as a queue header, and is never allocated with `alloc_dg` or `dequeue`, Sections 4.6 and 4.3), and a `buffer_size` field of 0 (i.e., no data buffer associated with the datagram). All datagram queues are maintained as doubly linked lists, for reasons described in Section 4.5, and are created with this subroutine.

Since the collection of datagrams that the Nproc allocates is in the shared memory region, the datagram queues associated with them must also be in shared memory. The Kproc, on the other hand, allocates datagram queues in its own local memory (to implement the per-process input queues). Because of this difference, `new_queue` tests to see whether it is being called from the Kproc or the Nproc.

4.1.1. Interface

```
function new_queue
    return DGG.datagram_pointer;
```

4.1.2. PDL

```
if is_Kproc then
    allocate datagram from local heap
else
    allocate datagram from shared memory heap
end if

next := self reference -- Circular list
prev := self reference -- Circular list
class := queue_head
buffer_size := 0
msg_count := 0
semaphore := free
```

4.2. enqueue

The enqueue subroutine is used by the Kernel to place a datagram at the tail of a queue. The enqueue subroutine, in conjunction with the dequeue subroutine (Section 4.3), provides a FIFO queueing system for datagrams.

The enqueue subroutine places the specified datagram at the tail of the specified queue. The msg_count field in the queue head is incremented by 1 for each datagram enqueued.

4.2.1. Interface

```
procedure enqueue (  
    dg : in DGG.datagram_pointer;  
    queue : in DGG.datagram_pointer  
);
```

4.2.2. PDL

```
begin atomic region  
    lock queue semaphore  
    if datagram is not busy (i.e., not in a queue) then  
        link datagram into tail of queue  
        mark datagram as busy (i.e., in a queue)  
        queue.msg_count := queue.msg_count + 1  
    else  
        null  
    end if  
    unlock queue semaphore  
end atomic region
```

4.3. dequeue

The dequeue subroutine is used by the Kernel to remove the first datagram from a queue and return it to the caller. The dequeue subroutine, in conjunction with the enqueue subroutine (Section 4.2), provides a FIFO queueing system for datagrams.

The dequeue subroutine removes the first datagram (i.e., the head of the queue) from the specified queue. The msg_count field in the queue header is decremented by 1 for each datagram dequeued. If no datagrams are available, dequeue returns null.

4.3.1. Interface

```
function dequeue (  
    queue: in DGG.datagram_pointer  
    ) return DGG.datagram_pointer;
```

4.3.2. PDL

```
begin atomic region
lock queue semaphore
if queue.msg_count > 0 then
    unlink first datagram in queue
    mark datagram as free (i.e., not in a queue)
    queue.msg_count := queue.msg_count - 1
    value_to_return := datagram
else
    value_to_return := null
end if
unlock queue semaphore
end atomic region
return value_to_return
```

4.4. get_first

The `get_first` subroutine is used to return a pointer to the first datagram in a specified queue without actually dequeuing it. It is used by the Kernel `receive_message` and `purge_message_queue` subroutines.

The `get_first` subroutine returns a pointer to the first datagram in a queue, but does not delete the datagram from the queue. This subroutine is used in conjunction with `delete` instead of `dequeue` to eliminate some race conditions. These conditions are documented in Appendix C.

4.4.1. Interface

```
function get_first (
    queue : in DGG.datagram_pointer
) return DGG.datagram_pointer;
```

4.4.2. PDL

```
begin atomic region
lock queue semaphore
if queue.msg_count > 0 then
    copy pointer to first datagram
    value_to_return := pointer
else
    value_to_return := null
end if
unlock queue semaphore
end atomic region
return value_to_return
```


4.5. delete

The `delete` subroutine is used to remove a specified datagram from a named queue. It is used by the Kernel to remove a specific datagram from a queue prior to releasing it to the free pool. A primary use of this subroutine is to delete a datagram (from anywhere in the queue) whose timeout has expired. Although messages are enqueued and dequeued in the process queues in FIFO order, timeouts do not necessarily occur in the same order. This subroutine is provided to enable the deletion of a timed-out message from the middle of the queue.

The `delete` subroutine removes the specified datagram from the specified queue. It operates on the assumption that the datagram is in the queue. No checking is done to ensure that this is the case; checking requires an $O(n)$ algorithm, while safe use of a non-checking algorithm requires an $O(1)$ algorithm. Should the test condition be false, the datagram queue will be damaged and subsequent operations on the queue will be jeopardized. This subroutine is used in conjunction with `get_first` instead of `dequeue` to eliminate race conditions. These conditions are documented in Appendix C.

4.5.1. Interface

```
procedure delete (  
    dg : in DGG.datagram_pointer;  
    queue : in DGG.datagram_pointer  
);
```

4.5.2. PDL

```
begin atomic region  
    lock queue semaphore  
    if datagram is busy (i.e., in a queue)  
        unlink specified datagram  
        mark datagram as free (i.e., not in a queue)  
    else  
        null  
    end if  
    unlock queue semaphore  
end atomic region
```

4.6. alloc_dg

The `alloc_dg` subroutine is called by the Kernel when it needs to allocate a new datagram for message transmission. Note that this subroutine differs from both `dequeue` and `get_first`, which return a datagram from a specific queue. `Alloc_dg` searches the available free queues and returns the smallest available datagram that fits the size parameter requirements.

The `alloc_dg` subroutine dequeues a datagram with a data buffer of at least the specified

size from the first available free queue. If no datagrams are available, it returns null. However, if the requested size is larger than the available buffers (a situation that *should not happen*, due to Ada compile-time or runtime range checking), it raises the exception `illegal_datagram_size`.

4.6.1. Interface

```
function alloc_dg (
    size : in DGG.buffer_range
) return DGG.datagram_pointer;
```

4.6.2. PDL

```
if size > large_queue.size and size > kernel_queue.size then
    raise illegal_datagram_size
end if;
--
-- The reason for using a comb structure for the conditional
-- statements (instead of nested elsifs) is that although the
-- application may request a small datagram buffer, there may
-- not be a buffer available - thus we must check to see if a
-- datagram has been allocated after each step, and try again
-- with a different buffer pool if not.
--
if size <= small_queue.size then
    dg := dequeue from small queue
    dg.do_optimization := false;
end if;
if dg = null and size <= large_queue.size then
    dg := dequeue from large queue
    dg.do_optimization := false;
end if;
if dg = null and size <= kernel_queue.size then
    dg := dequeue from kernel_queue
end if;
return dg;
```

4.7. free_dg

The `free_dg` subroutine is used by the Kernel communication subroutines when they have finished with a datagram. This subroutine finds the appropriate free queue in which to place the datagram. `Free_dg` must be called when a datagram is no longer needed; if not, the pool of available datagrams will eventually be exhausted, and subsequent calls to `alloc_dg` will be unable to allocate a datagram.

The `free_dg` subroutine enqueues the specified datagram into the appropriate free queue. Since all datagrams that the application can allocate should have been obtained via `alloc_dg`, the first three statements in the case statement should always be executed. However, in case the application attempts to release an illegal datagram type (i.e., one of type `queue_head`), this subroutine raises the exception `illegal_datagram_class`.

4.7.1. Interface

```
procedure free_dg (  
    dg : in DGG.datagram_pointer  
);
```

4.7.2. PDL

```
case dg.class  
    when small =>      enqueue(dg, small queue)  
    when large =>      enqueue(dg, large queue)  
    when kernel =>     enqueue(dg, kernel queue)  
    when queue_head => raise illegal_datagram_class  
end case
```

5. Packet Communication

All datagrams are broken up into 16-bit chunks (transmitted within 32-bit packets) for transmission around the network. Because of the asynchronous nature of the communication mechanism, there is no guarantee that a datagram arrives in its entirety before another datagram is transmitted from another node. While the transmission of an individual packet is an atomic operation, transmission of a datagram is not. This means that a given node may receive interleaved packets from multiple nodes. To ensure that all packets arrive at their correct destination (and that datagrams are correctly reconstructed at the destination node), each packet also contains the sender and receiver address.

A packet looks like the following illustration:

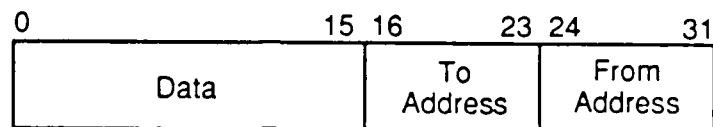


Figure 28: Packet Layout

From-addresses and to-addresses may fall in the range of 16#00# through 16#FE#. The value 16#FF# is reserved for internal use during initialization only, and may not be used as a valid network address (see Section 1.2 for details on how to configure the hardware network address on the Kproc and Nproc). The from- and to-addresses in the packets correspond to the `physical_address` field in the NCT, which is of type `NG.bus_address`.

6. Kproc/Nproc Interface

The interface between the Kproc and Nproc is asymmetrical. The Kproc is advised of changes in the input queue by interrupts from the Nproc, while the Nproc polls the output queue for changes in status. All data transfer is through the memory shared between the Kproc and Nproc, and this in turn is effected by the subroutines `enqueue`, `dequeue`, etc.

6.1. Shared Memory

The shared memory on a processor node resides at the addresses described in Section 3 and Section 10, Table 40. The first 256 bytes of this shared memory are reserved for shared variables, while the remaining bytes are used for the Ada heap for the Nproc. In this latter area, the Nproc allocates the datagram buffers and places them in the various free queues.⁶

There are two shared variables used by the Kproc/Nproc interface. These are `DGM.input_queue` and `DGM.output_queue`, and they reside at the addresses defined by `MEM.input_queue_address` and `MEM.output_queue_address`, respectively.

Other shared memory variables are used internally by package `DGM`, but these are not used by the Kproc/Nproc interface. The specific addresses and their uses can also be found in the package `memory_addresses`.

6.2. Enqueueing Messages for Transmission

When the Kproc wishes to transmit a message to another processor, it enqueues the datagram containing the message into the `output_queue`. The Nproc main loop code (Section 7.2) continually polls this queue. As soon as a datagram is present in the queue, the Nproc commences transmitting it over the network. The datagram must be correctly constructed (i.e., the sender and receiver fields must be initialized, as must the `message_length` and `operation` fields – see Section 2.4 for details on each of these fields). The subroutines `BIO.send_kernel_datagram` and `BIO_send_process_datagram` are typically used to construct these datagrams.

⁶The Ada heap for the Nproc is moved to the region of shared memory by a set of special linker directives. This causes the Ada runtime to allocate memory from shared memory instead of the normal location.

6.3. Receiving Incoming Messages

When the Nproc receives a complete datagram that is addressed to the current node, it enqueues the datagram into the `input_queue`, and generates an interprocessor interrupt via `IPI.generate_kn_interrupt`. The Kproc must have an interrupt handler to service this interrupt. When the Kproc responds to the interrupt, all it does is dequeue the datagram from the `input_queue` and act according to the message contained in it. The Kproc loops and dequeues as many datagrams as are available, since it may be the case that multiple datagrams have been enqueued before the Kproc has a chance to respond to the interrupt. The dequeue subroutine returns `null` if no datagrams are available in the `input_queue` (Section 4.3); this can be used as a test to see if the specified queue is empty.⁷

⁷The subroutine `BIO.initialize` is used in the Kproc to bind the interrupt handler contained in `BIO.receive_receive_datagram_interrupt_handler` (this subroutine is not exported in the spec), which will in turn call the appropriate receive subroutines.

7. Nproc Communication Routines

The Nproc operates as a purely interrupt-driven communication medium, with the non-interrupt cycles being used to poll the output buffer for completed datagrams (and thence to send them out onto the network). The Nproc communication subroutines are broken up into four major parts:

1. Initialization. This part of the Nproc code is used to set up the communication ports, allocate the shared datagram buffers, etc.
2. Main loop. The main loop is the polling (i.e., non-interrupt driven) part of the Nproc code. It is used to scan for datagrams that are to be transmitted onto the network.
3. Receiver interrupt service. This subroutine services interrupts as packets arrive at this node. Packets are then tagged as:
 - Addressed to this node (in which case they are reassembled as datagrams),
 - Addressed to another node (in which case they are simply passed through to the next node downstream), or
 - "Rogue packets" (which, due to an unknown – and unchecked – error condition, are incorrectly addressed and are discarded).
4. Transmitter interrupt service. This subroutine services interrupts whenever a packet may be transmitted from this node. Packets are classified as either:
 - "Originate" packets (which are from datagrams composed on the current node), or
 - "Thru" packets (which have been received by this node, but which are addressed to another node in the network).

The transmitter interrupt service subroutine sends a mixture of originate and thru packets so as to fairly distribute network traffic.

The individual subroutines are discussed in greater detail in the following sections.

7.1. Initialization

The initialization subroutine is used both to set up the contents of the Nproc and to initialize the parallel I/O board (see Section 2 for details on the parallel I/O hardware). The subroutine is also responsible for binding and enabling the interrupt handlers (whose job it is to communicate with the parallel port), and for getting the network into a known startup state. It does this by transmitting a "magic cookie" packet and waiting for the "magic cookie" to appear on its receive port. Since the ability to send a packet around the network is essential (that is, since the closure of the Nproc communications ring must be guaranteed for any network communication to be possible), the Nproc will wait as long as necessary for the "magic cookie" to be received.

This method works both when the network is starting up (in which case each Nproc transmits the "magic cookie" to the node downstream of it), and when a single node reboots when the network is already running (in which case the newly started Nproc transmits the "magic cookie," and every other Nproc passes it through (since it is not addressed to it), until the "magic cookie" finds its way completely around the ring to the sender).

The contents of the "magic cookie" packet is defined to be the 16#00FF_DEAD#. This corresponds to a sender address of 16#00# and a receiver address of 16#FF#, the latter address being "illegal" and thus recognized as a special case by the Nproc software.

7.1.1. PDL

```

DGM.Nproc_initialize
PIO.initialize_recv
PIO.initialize_xmit
send "magic cookie" on output port
loop
    exit when input port = "magic cookie"
end loop

KIM.bind_interrupt_handler(parallel input vector)
KIM.enable(parallel input vector)
KIM.bind_interrupt_handler(parallel output vector)
KIM.enable(parallel output vector)

PIO.enable_recv_interrupt
IPI.enable_kn_interrupt

```

7.2. Main Loop

The main loop code of the Nproc simply checks to see if any datagrams are present in the output queue. If there are, and if no datagram is currently being processed, the Nproc begins processing the datagram by calling the transmitter prime subroutine (see Section 7.6 for details on this subroutine). No other work is done by the Nproc except for processing incoming datagrams in the interrupt service subroutines.

7.2.1. PDL

```

loop
    if current output datagram = null then
        current output datagram := dequeue from output queue
        if current output datagram /= null then
            calculate checksum
            calculate count of words to be transmitted
            mark current output datagram as "active"
            call transmitter prime
        end if
    end if
end loop

```


7.3. Calculate Checksum Routine

In the present implementation, this subroutine is null, and returns a value of 0. In future implementations, this subroutine can be replaced by a true checksum or cyclic redundancy check (CRC) subroutine to verify the validity of the datagram contents. In the current configuration, where all datagram packets are transmitted with a full handshake protocol, it was not seen as necessary to implement this subroutine.

7.4. Receive Packet Interrupt Service Routine

The receive packet interrupt service subroutine is used to capture packets from the parallel port and process them according to their address. Packets fall into three categories:

1. Packets that are addressed *to* this processor node
2. Packets that are addressed *from* this processor node
3. Packets that do not fall into either above category

Packets of the first category are collected into datagrams according to their sender. Packets of the second category are discarded, since a packet sent by a node which has not been picked up by a receiver is addressed to an illegal node address.⁸ Packets of the third category are addressed to a different node and are passed through this node (with the intent of their eventually reaching their destination).

Since a datagram is transmitted in its entirety by a node, before the next datagram can be started, it is guaranteed that each incoming datagram is contiguous *relative to its sender*. Although datagrams from multiple senders may arrive at this node in an interleaved fashion, each datagram from a single sender is contiguous. That is to say, each datagram is transmitted in its entirety by a node before another datagram is started. However, to allow for equitable network traffic management, a node will intersperse "thru" packets and "originate" packets. Because of this, once an initial packet is received from a sender, it is assumed that all following packets from that same sender will be part of the same datagram (up to the declared size of the datagram), although other packets may arrive from other nodes in between. Thus, the first packet of a datagram contains the size of the datagram that is to be received.

If the packet is addressed to the current node, we determine what the current status of the *sender* is with respect to this node. It can be in one of three modes (with a number of possible transitions):

1. *idle* – Currently, there is no partial datagram being received by this node that

⁸This mechanism may change in future implementations of the Kernel. In the current implementation, datagrams that make it around the network without being "claimed" are in error and discarded. Future implementations may make use of more complex error recovery schemes. This implementation, however, takes the simpler approach.

was sent by the specified remote node. In this case, we allocate a datagram buffer to contain the incoming message (recall that the first packet from a node contains the size of the datagram to follow, and that we can allocate a datagram buffer based on the first packet received from a node).

If we are able to allocate the datagram buffer, the status is set to *active*, and all subsequent packets for the datagram from that node are written into the datagram buffer. If no datagram buffer of the appropriate size is available, the mode is set to *discard*, and all subsequent packets that comprise that datagram are discarded.

2. *discard* - All incoming packets from the specified node are discarded until the current incoming datagram is complete. This mode will be used only rarely, as it is expected that the datagram traffic will not be so high that the Nproc will run out of buffers. Note that this mode reflects a lack of datagrams and is not the same as the process message queue overflow state in the *Kernel Facilities Definition*, Section 10.1.22.

When the current incoming datagram is complete (the size of the datagram was transmitted as the first packet of the datagram), the mode will be set to *idle*, so that subsequent incoming packets from that node will again signal the start of a datagram.

3. *active* - All incoming packets from the specified node are placed into the incoming datagram buffer associated with that node (recall that datagrams are transmitted contiguously with respect to a single node).

When the current incoming datagram is complete (the size of the datagram was transmitted as the first packet of the datagram), the mode will be set to *idle*, so that subsequent incoming packets from that node will again signal the start of a datagram. Additionally, the Nproc interrupts the Kproc, advising it that an incoming datagram has been received.

If a packet is destined for a different processor, it is simply loaded into a "thru buffer" for transmission to the next node in the ring (the thru buffer is emptied by the transmit interrupt service subroutine, Section 7.5). If the thru buffer becomes full as a result of enqueueing a packet, receive interrupts are disabled to prevent buffer overrun. It is anticipated that this condition will never happen, since the Nproc does nothing other than process datagrams and packets (and thus would never be loaded enough for the thru buffer to fill). If, however, the node is so overloaded that the thru buffer fills up, we allow a temporary shut down (the receiver interrupts are subsequently reenabled in the transmit interrupt subroutine). If the thru buffer was empty before the packet was enqueued, this means that the thru buffer now has something in it, and the transmitter needs to be primed. If this is the case, the transmitter prime subroutine is called to start transmitting the contents of the thru buffer (see Section 7.6 for details on the prime subroutine).

7.4.1. PDL

```

while PIO.recv_buffer_full loop
  PIO.acknowledge_recv_interrupt
  if incoming packet addressed to this node then
    case incoming status[from address] is
      when idle => -- No datagram active
        calculate size of incoming message
        allocate incoming datagram buffer
        if no buffers available then
          set status := discard
        else
          set status := active
          save incoming packet into buffer
        end if
      when discard => -- Throw out mode
        throw out packet
      when active => -- Save mode
        save incoming packet
        if packet is end of datagram
          calculate checksum
          if checksum error then
            discard incoming datagram
          elsif datagram is sync message
            act on sync message
          else
            enqueue into Kproc receive queue
            set status := idle
            IPI.generate_kn_interrupt
          end if
        end if
      end case
    elsif incoming packet addressed from this node or
      incoming packet addresses are illegal
      throw packet out
    else
      put packet into thru buffer
      case thru_buffer_status is
        when is_full =>
          PIO.disable_recv_interrupt
        when was_empty =>
          xmit_prime
        when others =>
          null
      end case
    end if
  end loop

```

7.5. Transmit Packet Interrupt Service Routine

The transmit interrupt service subroutine is used to transmit packets to other nodes. Packets fall into two categories:

1. Packets originating on this node. These packets are dequeued from the Kproc output queue by the main loop code (Section 7.2).
2. Packets originating on other nodes, passing through this node. These packets are received by the receive packet interrupt handler (Section 7.4) and are enqueued into the thru packet buffer.

The transmit interrupt handler sends both types of packets out onto the network in as fair a distribution as possible (to prevent one node from monopolizing the network). When both types of packets are available to be sent by the Nproc, the transmit interrupt subroutine sends one "originate" packet for every $(NC.number\_of\_nodes+1)/2$ "thru" packets. This keeps any one node from overloading the network when other nodes need to transmit. If only one kind of packet is available, no rationing is performed, and all packets of that type are transmitted.

When thru packets are transmitted, one of three conditions can exist:

1. The thru buffer was previously full, and sending a packet created space in the queue. In this case, receiver interrupts are re-enabled, allowing more incoming packets to arrive (see Section 7.4).
2. The thru buffer is empty, having been exhausted by the last transmission. In this case, if there are no originate packets to be sent, transmitter interrupts are turned off, since no packets remain to be sent. Packets may later become available in the receive interrupt subroutine (Section 7.4) or from the main loop code (Section 7.2), in which case transmit interrupts are re-enabled.
3. The thru buffer is neither full nor empty, in which case nothing is done with either interrupt mechanism.

7.5.1. PDL

```
loop
  if thru buffer not empty and thru count < maximum then
    PIO.send_packet(thru packet)
    if thru status = was_full then
      process incoming packet
      PIO.enable_recv_interrupt
    elsif thru status = is_empty and
      current output datagram = null then
      PIO.disable_xmit_interrupt
    else
      null
    end if
  return -- we sent something, so quit
```

```

elseif current output datagram state = active then
    PIO.send_packet(originate packet)
    if current output datagram finished then
        DGM.free_dg(current output datagram)
        current output datagram state := idle
        if thru buffer empty then
            PIO.disable_xmit_interrupt
            return
        end if
    end if
    reset thru count := 0
    return -- we sent something, so quit
else
    reset thru count := 0
    if thru buffer empty then
        PIO.disable_xmit_interrupt
        return -- nothing to send, so quit
    else
        null -- something to send, continue loop
    end if
end if
end loop

```

7.6. Transmitter Prime

The transmitter prime subroutine is used to prime the transmitter. It is called when the main loop starts sending a new outgoing datagram, or when the receiver interrupt subroutine sends the first thru packet. Both subroutines use it to start the transmitter cycling, so that as each packet is transmitted, an interrupt is generated to get the next packet.

The transmitter on the parallel I/O board issues a transmitter interrupt when it is able to transmit to the parallel port (i.e., when the transmitter data port is empty). This is different from the receiver side, which interrupts when it *has received* data on the parallel port (i.e., when the receiver data buffer is full). Priming the transmitter pump, therefore, is simply a matter of enabling transmitter interrupts. The parallel I/O board issues an interrupt as soon as it can transmit, which causes the transmitter interrupt subroutine to be called. This in turn transmits the first available word from the output datagram.

The transmitter may already be enabled (if there are thru packets still to be transmitted), in which case:

1. The transmitter may not interrupt immediately. This is not a concern, since it will interrupt soon enough, and when it does, the transmitter interrupt subroutine arbitrates between originate and thru packets,
2. The transmitter is already enabled. This also is of no concern, since enabling an already enabled interrupt is legal.

7.6.1. PDL

`PIO.enable_xmit_interrupt`

8. Message Transfer Thread Examples

The following two sections outline the transfer of a message from one Kproc to another via the DARK network. The first section describes the message transfer in words, calling out each Kernel routine that is used to move data from one location to another. The second section views this transfer from a higher level, pictorially representing the data flow from the application on one Kproc to another.

8.1. Detailed Thread Description

This section presents a simple scenario of one Kernel process transmitting a message to another Kernel process residing on another processor. The receiving process is not pending on a call to the Kernel primitive `receive_message` at the time the message is sent. For readability purposes, the sending Kproc actions are in regular type, the sending Nproc actions are in bold type, those of the receiving Kproc are in italics, and those of the receiving Nproc are in bold italics.

1. User process issues a `CM.send_message` Kernel call.
2. Local Kernel allocates datagram buffer of appropriate size from free list by using `DGM.alloc_dg`. If none is available, control is returned to the application (datagram model does not require error reporting).
3. Local Kernel builds datagram header and copies application message into datagram buffer using `BIO.send_process_datagram`.
4. Local Kernel places datagram buffer into `DGM.output_queue` with `DGM.enqueue`.
5. **Local Nproc detects buffer in queue when `DGM.dequeue` returns a valid datagram (Nproc simply idle loops when there are no messages to send).**
6. **Local Nproc transmits datagram contents onto network using established low-level protocol.**
7. **Local Nproc calls `DGM.free_dg` to deallocate the datagram buffer.**
8. ***Remote Nproc detects "new datagram begins" packet. The first packet of a datagram contains the length of the datagram to follow.***
9. ***Remote Nproc calls `DGM.alloc_dg` to allocate a datagram buffer of the appropriate size from free list. If none is available, all remaining bytes from this incoming datagram are discarded.***
10. ***Remote Nproc fills datagram buffer packet by packet as received off the network. Note that multiple incoming datagrams may be processed simultaneously, although only one will be coming from any one processor at a time.***
11. ***When the incoming datagram is complete, the remote Nproc calls `DGM.enqueue` to enqueue the datagram into `DGM.input_queue`.***
12. ***Remote Nproc calls `IPI.generate_kn_interrupt` to interrupt the associated remote Kproc.***

13. *Remote Kproc BIO.receive_datagram_interrupt_handler is entered.*
14. *Interrupt service calls DGM.dequeue to remove the datagram from DGM.input_queue.*
15. *Datagram contents are decoded, and a pointer to the datagram is enqueued onto the receive process message queue. If the process receive message queue is full, the error action selected by the queue_overwrite_rule is performed.*
16. *Loop back to step 14 until no more datagram pointers are in Nproc input queue.*
17. *At some later time, remote application issues CM.receive_message Kernel call.*
18. *Remote Kernel calls DGM.get_first to get a pointer to first datagram in process message queue and copies data into application buffer.*
19. *Remote Kernel calls DGM.delete to remove the datagram from the application's incoming message queue.*

8.2. Graphic Representation of Thread

This section presents another simple scenario of one Kernel process transmitting a message to a second Kernel process residing on a different Kproc. Except where absolutely essential, the specific routines used within the Kernel are not called out. Refer to Section 8.1 for these details.

In Figure 29, the sending process calls `CM.send_message` to cause a message to be transmitted to another process. To preserve the integrity of the data in the message, the Kernel allocates a datagram in the shared memory pool and copies the application message into it. The Kernel also maintains datagram bookkeeping information in the datagram header, including the linked list pointers, application message length, sender, receiver, etc.

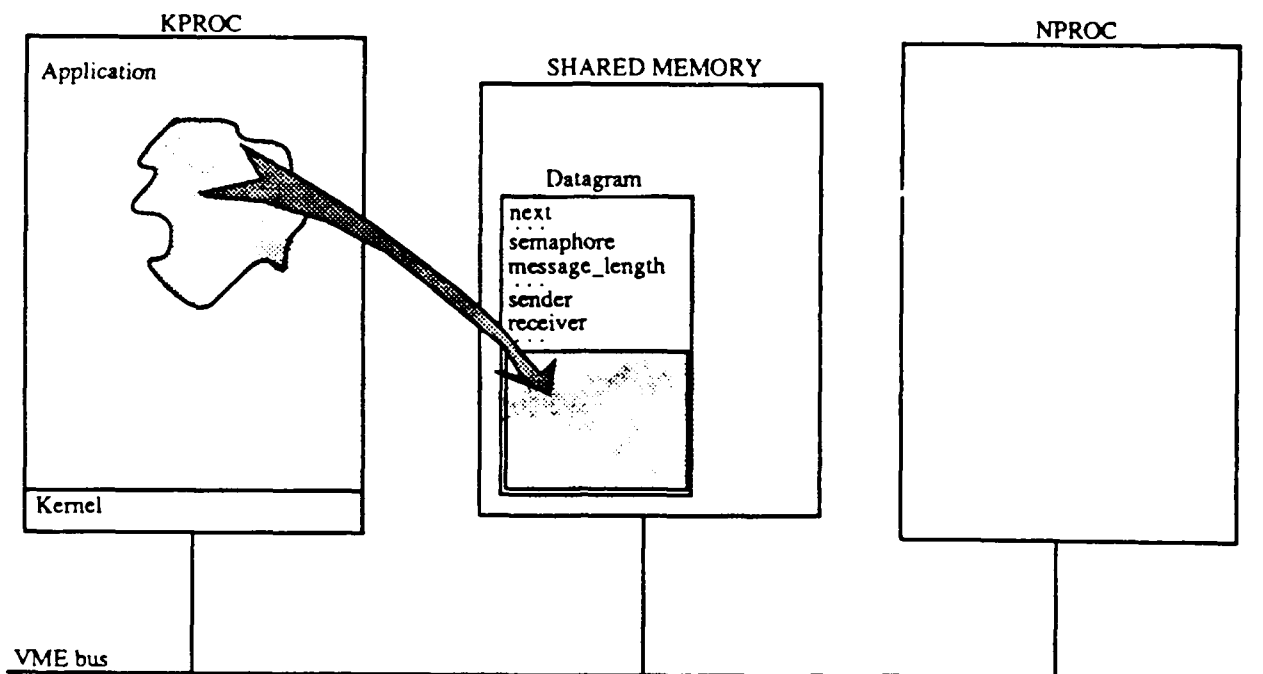


Figure 29: Send_Message – Application Message to Datagram

As can be seen in the figure above, the Kproc, Nproc, and shared memory are all co-resident on the VME bus, and both the Kproc and Nproc can address the shared memory with the same virtual addresses.

Once the application message has been copied into the datagram, and once the bookkeeping information has been initialized, the datagram is enqueued into the output queue so that the Nproc can transmit the datagram onto the network. As shown in Figure 30, the output message queue is maintained as a circular doubly linked list. Each datagram header contains pointers to both the next and previous datagrams in the output queue (in fact, all datagram lists – input queues, output queues, and free lists – are maintained as circular doubly linked lists).

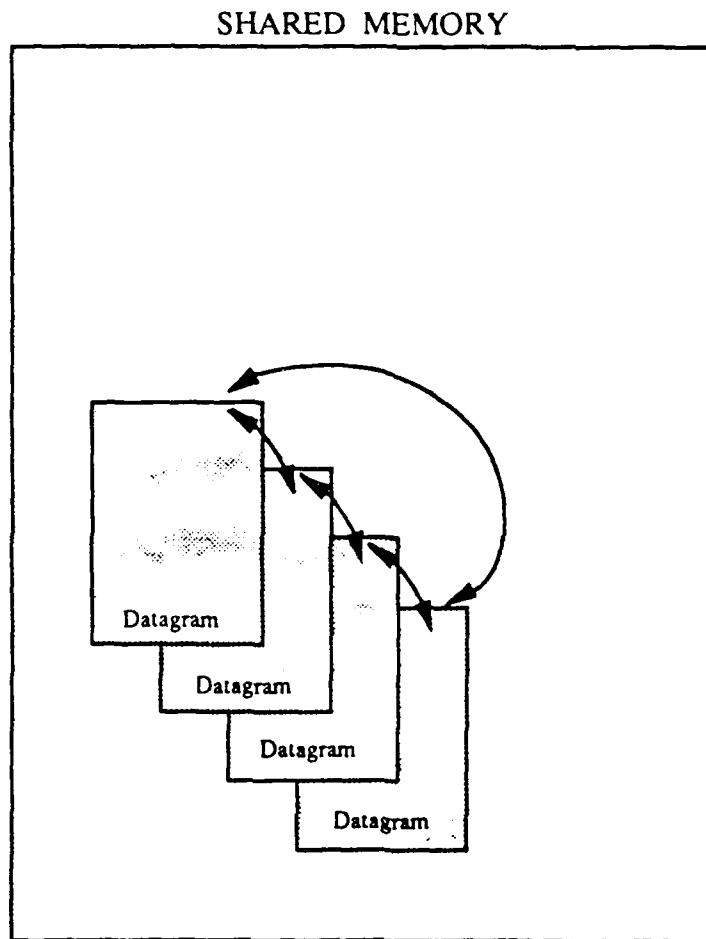


Figure 30: Output Message Queue

When the Nproc detects the presence of a datagram in the output queue (and when no other transmission action is being performed), it transmits the datagram to the network. The Nproc does so by splitting the datagram into a series of 32-bit packets and transmitting them in sequence. As shown in Figure 31, the datagram header is not transmitted to the network. Instead, data are extracted from the datagram body in 16-bit chunks and bundled with the 16-bit sender and receiver address to form a 32-bit packet (see Figure 28 in Section 5 for more details).

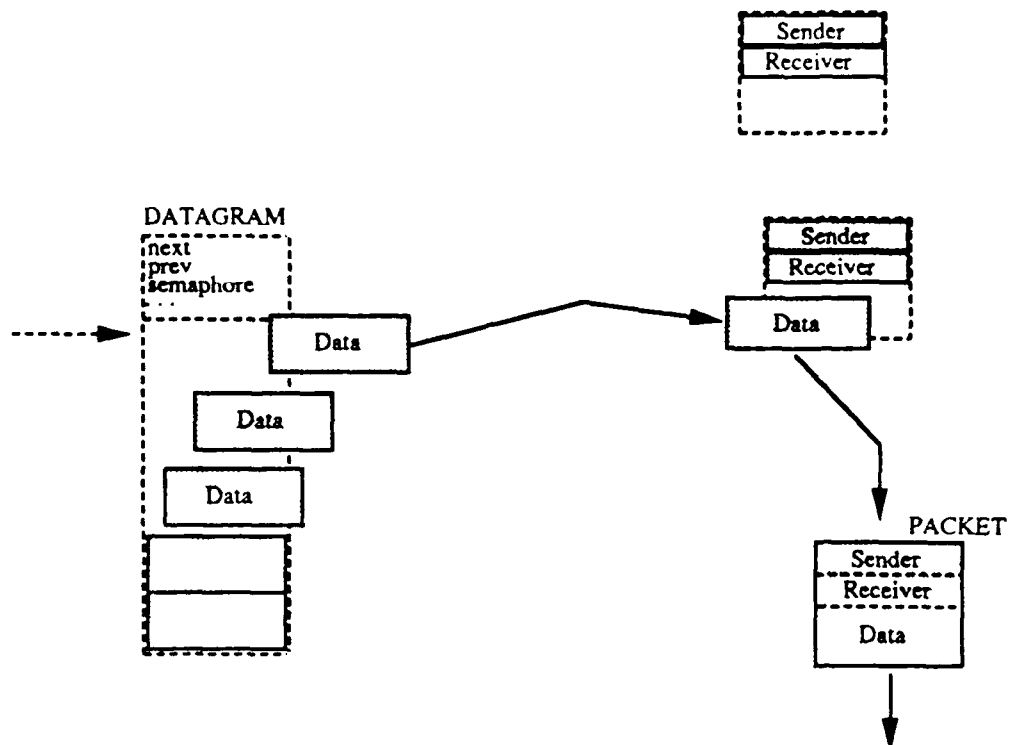


Figure 31: Datagram to Packet Data Flow

As described in Section 7, packets can arrive at an Nproc at the same time that the Nproc is transmitting packets. These arriving packets can either be addressed to the processor node or they can be packets that are to be passed through this node to another processor node. Figure 32 illustrates this multiple pathway communication scheme.

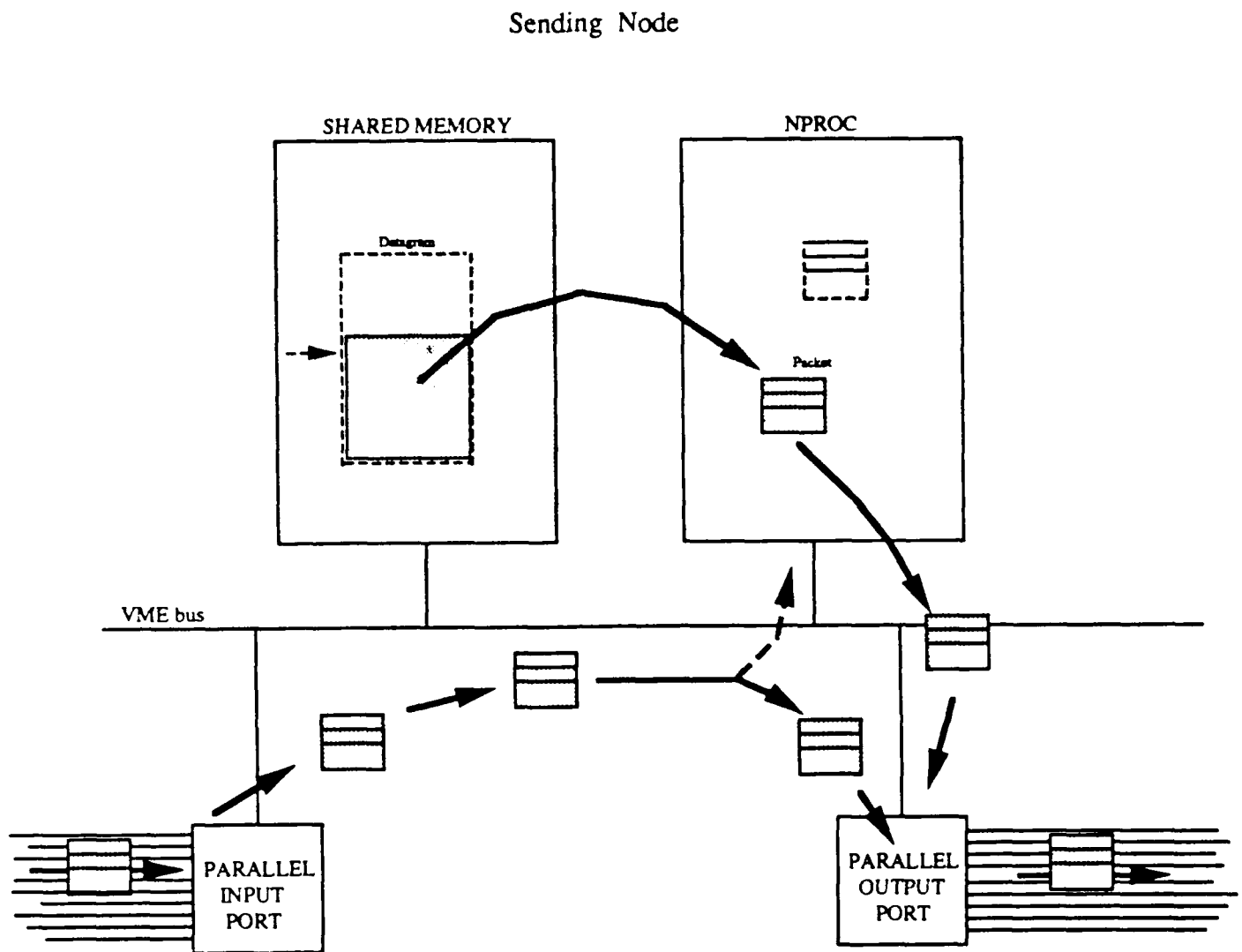


Figure 32: Packet Traffic onto Network

Packets arrive at the parallel input port and are processed by the Nproc (either for "thru" transmission or to be stored by this processor node). While this is happening, the Nproc is also extracting data from the datagram body, bundling it into packets, and transmitting it onto the network.

As mentioned previously, packets that arrive at a node are either addressed to that node or are addressed to a different node. As seen in Figure 33, "thru" packets are simply retransmitted on to the network. Those packets that are addressed to the current processor node are bundled into a datagram in the shared memory pool. Section 7.4 describes this procedure in detail.

Receiving Node

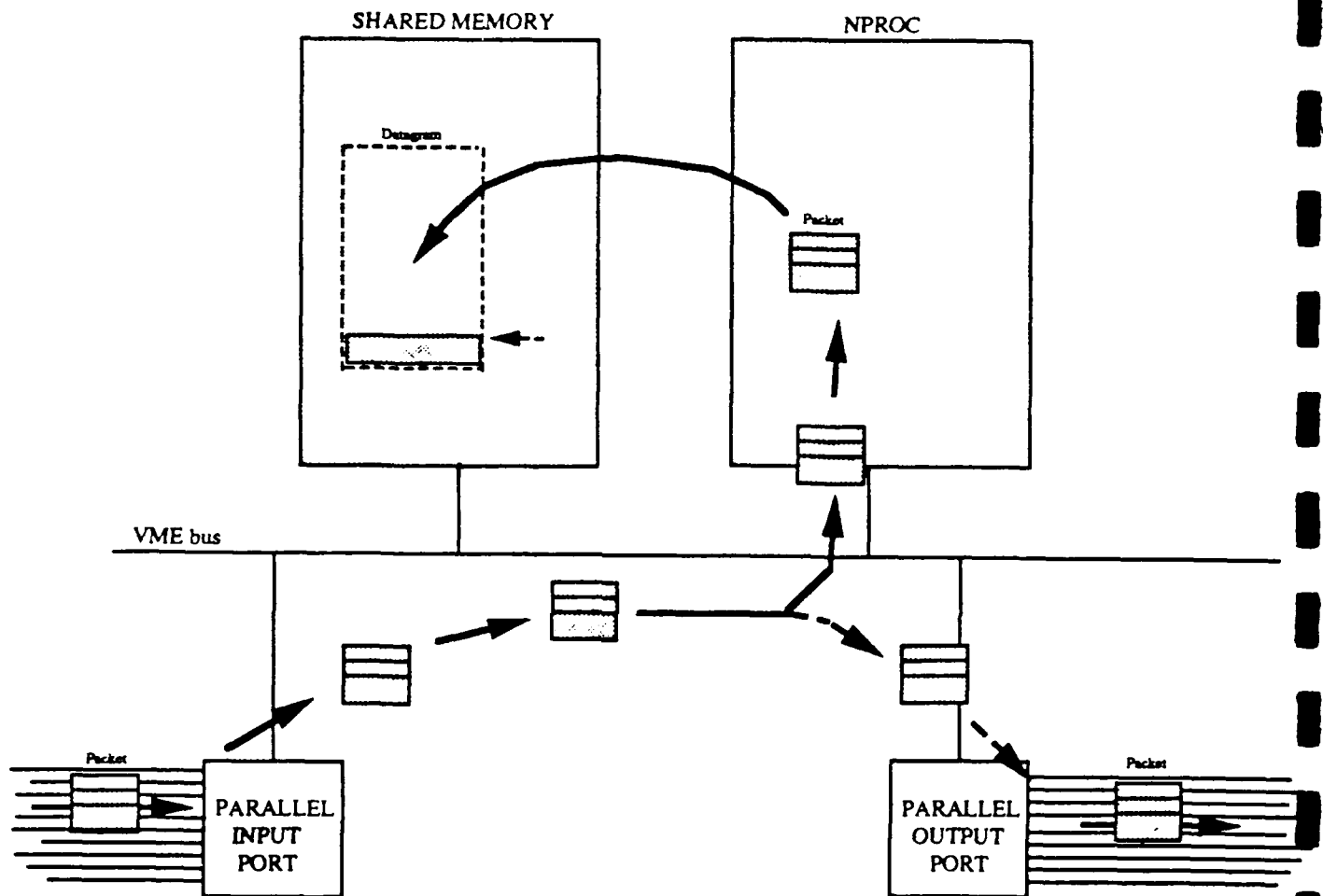


Figure 33: Packet Traffic off Network

Once a datagram has been completely received, the Nproc places it into the input queue of the associated Kproc and generates an interprocessor interrupt (Section 7.4.1 details the code needed to perform these actions).

The Kproc responds to the interrupt and moves the datagram from the general input buffer to the specific application process buffer. The application can then issue a call to `CM.receive_message` to copy the contents of the datagram into the application buffer. Figure 34 illustrates this latter process. The Kernel copies the body of the datagram into the application buffer and discards the datagram. Some of the information in the datagram header is passed back to the application (i.e., the sender, message tag, and message length), while other information is simply used for Kernel bookkeeping e.g., the linked list pointers, checksum).

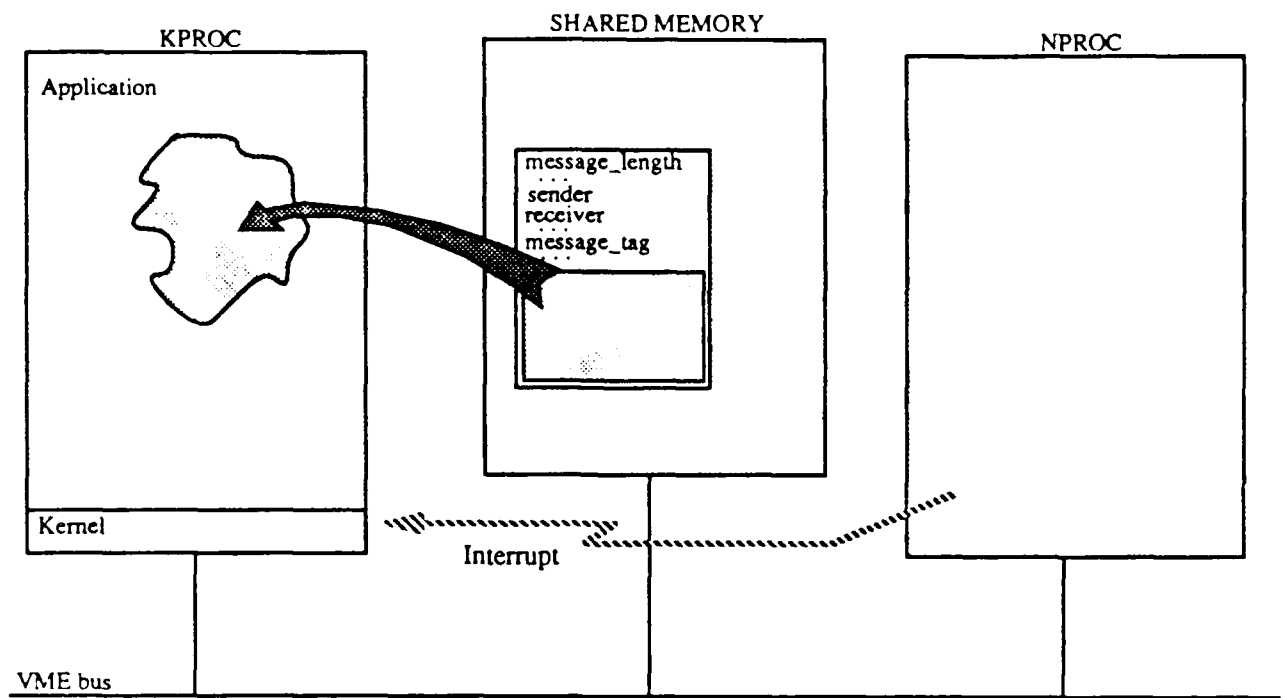
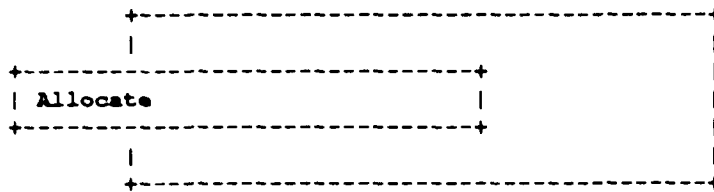


Figure 34: `Receive_Message` – Datagram to Application Message

V. General Utilities

1. Low_level_storage_manager



This package is used to allocate blocks of bytes.

1.1. Allocate

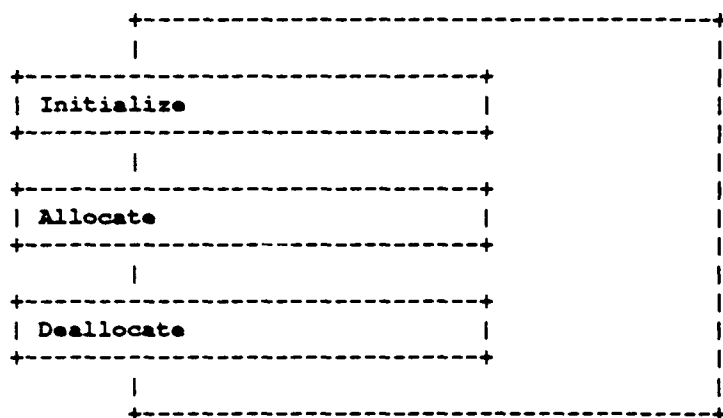
1.1.1. Interface

Allocate (number of bytes to reserve)

1.1.2. PDL

Allocate a byte array of the desired size
Return the address of the allocated area

2. Storage Manager



This package manages all allocation and deallocation of dynamic storage. It does this by encapsulating the equivalent Ada calls. This package exists only to provide the capability to change to a different allocation mechanism should the standard Ada allocator prove to cause problems.

2.1. Allocate

2.1.1. Interface

`Allocate return object pointer`

2.1.2. PDL

`return new object`

2.2. Deallocate

2.2.1. Interface

`Deallocate (object pointer)`

2.2.2. PDL

`Free (object pointer)`

2.3. Initialize

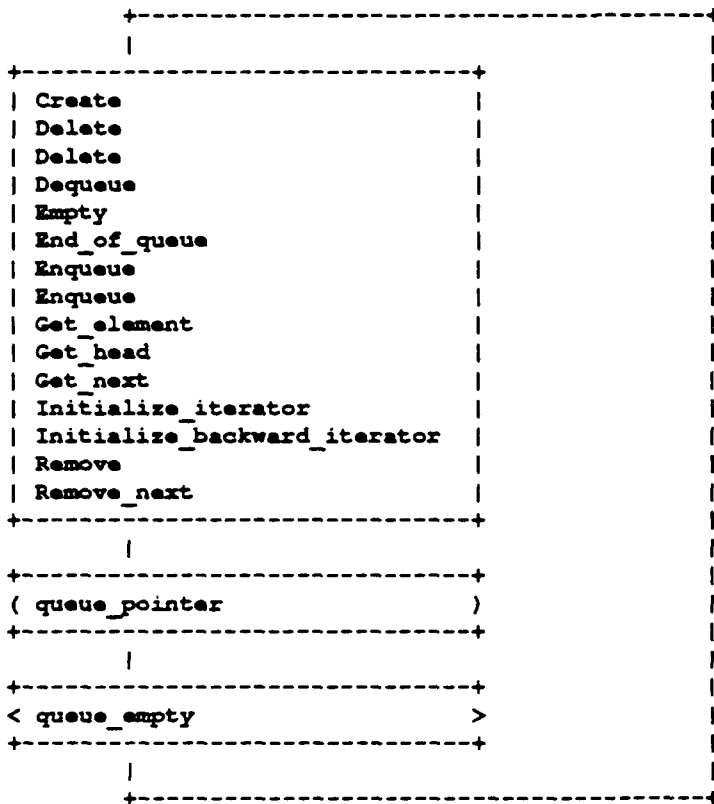
2.3.1. Interface

`Initialize`

2.3.2. PDL

`Null`

3. Queue Manager

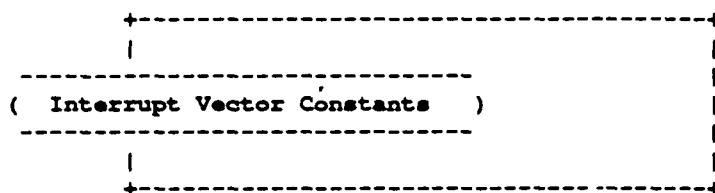


This package builds and manages an ordered queue for the instantiated type.

This package is capable of being used in two modes: normal and fast. In normal mode, all the operations search the queue for the desired element. In fast mode, a pointer to the element is generated on insertion and used for quick retrieval and removal.

VI. Target-Specific Utilities

1. Interrupt Names



This package contains a set of named constants for use in setting up and handling interrupts. These constants represent known vector numbers that uniquely identify an interrupt.

The devices associated with the vectors and a discussion of interrupts in general can be found in Part VIII.

1.1. Interrupt Vector Constants

There are several interrupt vector constants for each of the following devices:

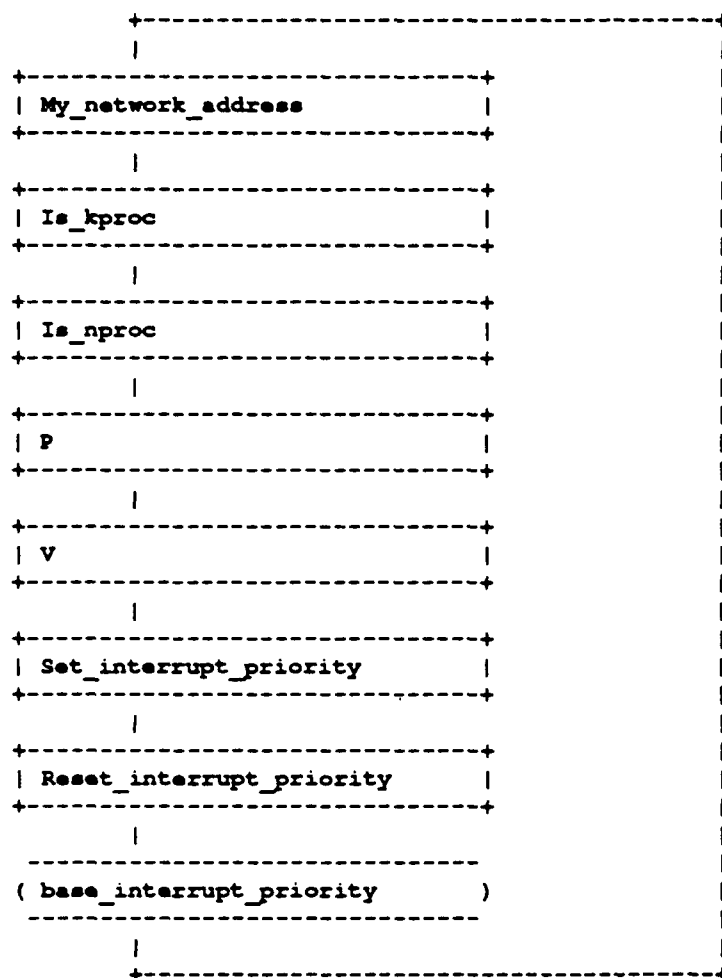
1. Parallel I/O (PIO)
2. 8-bit timer (Timer)
3. 24-bit timers (Timer)
4. Serial I/O (SIO)
5. Real-time clock
6. Interprocessor interrupter

The constant names have the following form:

<device name>_<device section>_vector

Where <device name> is one of the short names in the above paragraph, and <device section> is a unique designation given to the different sections of a device. For example, there are four 24-bit timers - A, B, C, and D; thus, A Timer's interrupt vector constant is Timer_A_Vector.

2. Low_level_hardware



2.1. My_network_address

2.1.1. Interface

`my_network_address` return network address of caller

2.1.2. PDL

return value from hardware specific address for network address

2.2. Is_kproc

2.2.1. Interface

is_kproc return boolean

2.2.2. PDL

return value from hardware specific address for Kproc/Nproc flag

2.3. Is_nproc

2.3.1. Interface

is_nproc return boolean

2.3.2. PDL

return value from hardware specific address for Kproc/Nproc flag

2.4. P

2.4.1. Interface

P

2.4.2. PDL

Use 68020 "CAS" instruction to claim a Dijkstra style semaphore. Semaphore is shared between Kproc and Nproc.

2.5. V

2.5.1. Interface

V

2.5.2. PDL

Release a Dijkstra style semaphore. Semaphore is shared between Kproc and Nproc.

2.6. Set_interrupt_priority

Note, HIGH_INTERRUPT_PRIORITY (used below) has an actual value of 6, this shuts out all interrupts except the real-time clock interrupt.

2.6.1. Interface

`set_interrupt_priority return old interrupt priority`

2.6.2. PDL

Trap to Ada Runtime with request to set process's priority to
HIGH_INTERRUPT_PRIORITY
Return previous process priority
1

2.7. Reset_interrupt_priority

Note, in normal usage, the parameter to reset_interrupt_priority will be the result returned by the corresponding call of set_interrupt_priority that opened the atomic region. Exceptionally, when an atomic region is to be unconditionally closed, an unmatched call to reset_interrupt_priority appears, taking as a parameter the value BASE_INTERRUPT_PRIORITY exported by this package.

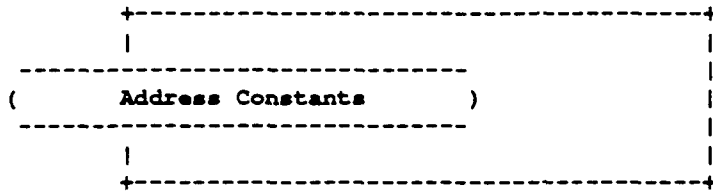
2.7.1. Interface

`reset_interrupt_priority (old interrupt priority)`

2.7.2. PDL

Trap to Ada Runtime with request to set process's priority to
the old interrupt priority
The previous process priority is discarded

3. Memory Addresses



This package contains only named address constants. These constants are used specify addresses of variables that must be placed at a particular memory location.

In particular, these constants define the location of the different message queues that reside in shared memory. For further information on the queues and shared memory refer to Part IV and Part VIII.

3.1. Address Constants

The address constants found in this package are for the small, large, and Kernel datagram queues, and input message queue and output message queue.

The constants names have the following form:

<queue name>_address

Where <queue name> is a string that identifies the queue name.

4. MVME133A Definitions

```

+-----+
|
|-----|
| ( Hardware Address Constants ) |
|-----|
|
|-----|
| ( Type MFP_Registers ) |
|-----|
|
|-----|
| ( Type MSR_Register ) |
|-----|
|
|-----|
| ( Type MSR_Bits ) |
|-----|
|
+-----+
```

This package contains named address constants and representation specifications for the MVME133A Mono Board Computer. This board has several devices that are used during the operation of DARK, such as MC68020 Microprocessor, and MC68901 Multi-Function Peripheral (MFP). Refer to Part VIII for additional information.

The MVME133A also has the Z8530 Serial Communication Controller (SCC), but it is not referenced or supported in this package. Refer to Chapter 6 for more information on this device.

4.1. Hardware Address Constants

The hardware address constants identify the locations of the MFP and Module Status Register.

4.2. Type MFP_Registers

Ada representation specification clauses are used to define the type and layout of the MFP register set.

There is an access type and constant declared for referencing the MFP. The access constant points to the MFP register set and allows for easy device access through the Ada language.

4.3. Type MSR_Register and MSR_Bits

The MSR is defined with an Ada representation specification clause. It defines the layout and type of each field in the register. Two representations are provided for use in different type access to the register.

An access type of the MSR_Register record type is declared for accessing the structure.

5. MZ8305 Definitions

```
+-----+
|
|-----|
| ( Hardware Address Constants ) |
|-----|
|
|-----|
| ( Type PIO_Control )          |
|-----|
|
|-----|
| ( Type Timer_Control )        |
|-----|
|
+-----+
```

This package contains named address constants and representation specifications for the MC68230 PI/T device on the Mizar MZ8305 Parallel Interface and Timer hardware board. Refer to Part VIII.

5.1. Hardware Address Constants

The hardware address constants identify the locations of the base of the two Parallel I/O and Timer device register banks.

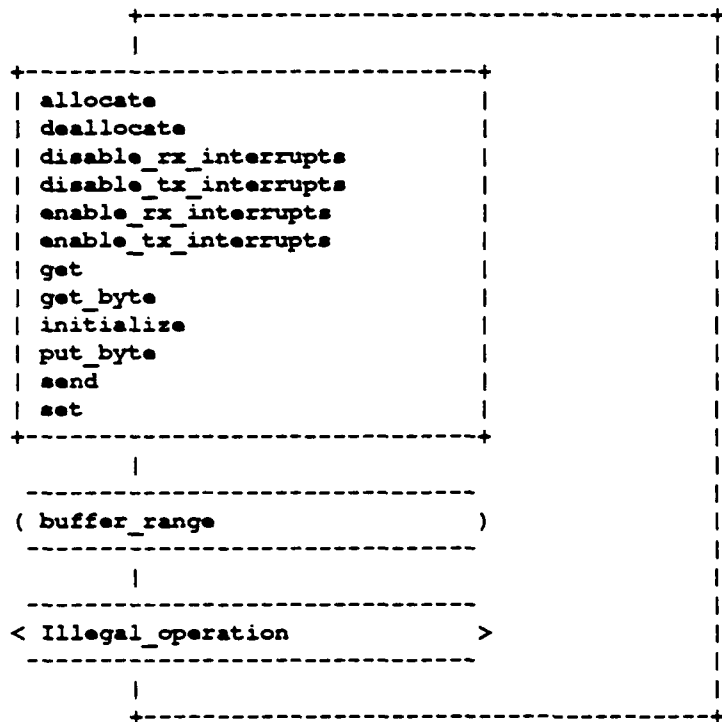
5.2. Type PIO_Control And Timer_Conrol

Ada representation specification clauses are use to define the type and layout of each register in the device register bank. The layout of the individual registers map directly to those defined in the user's manual for the PI/T device.

The register bank includes registers for the parallel I/O and timer; however, they are declared in two separate representation specifications, since the two sections are functionally different and are always reference separately.

In addition, there are access types and constants declared for the parallel I/O and timer register sets. The access constants point to each of the register sets for easy device reference.

6. SCC_porta



The Serial Communications Controller (SCC) hardware is a general purpose I/O device. A description of the hardware can be found in Chapter VIII. This package is used to support the *synchronize* primitive discussed in Chapter 13 and, as such, is not a general purpose serial i/o package.

SCC Port A is used to provide an independent time synchronization mechanism using standard, commonly available hardware. Each Kproc is connected via SCC Port A to form the sync bus, dedicated exclusively to implementing the *synchronize* primitive on a non-interfering basis. The software is constructed such that only one node, call the bus master, is allowed to transmit at any given instant. Normally, the sync bus is in an idle state where no one is transmitting or receiving. Mastership of the bus is achieved by successfully allocating the device. When some process has successfully allocated the sync bus (i.e., has become bus master) all other nodes on the bus are automatically transitioned into slave mode, waiting for the *time_is_now* message from the bus master. The detailed layout of the sync messages are contained in Appendix B, Table 2.

Some additional points to note about the sync bus software:

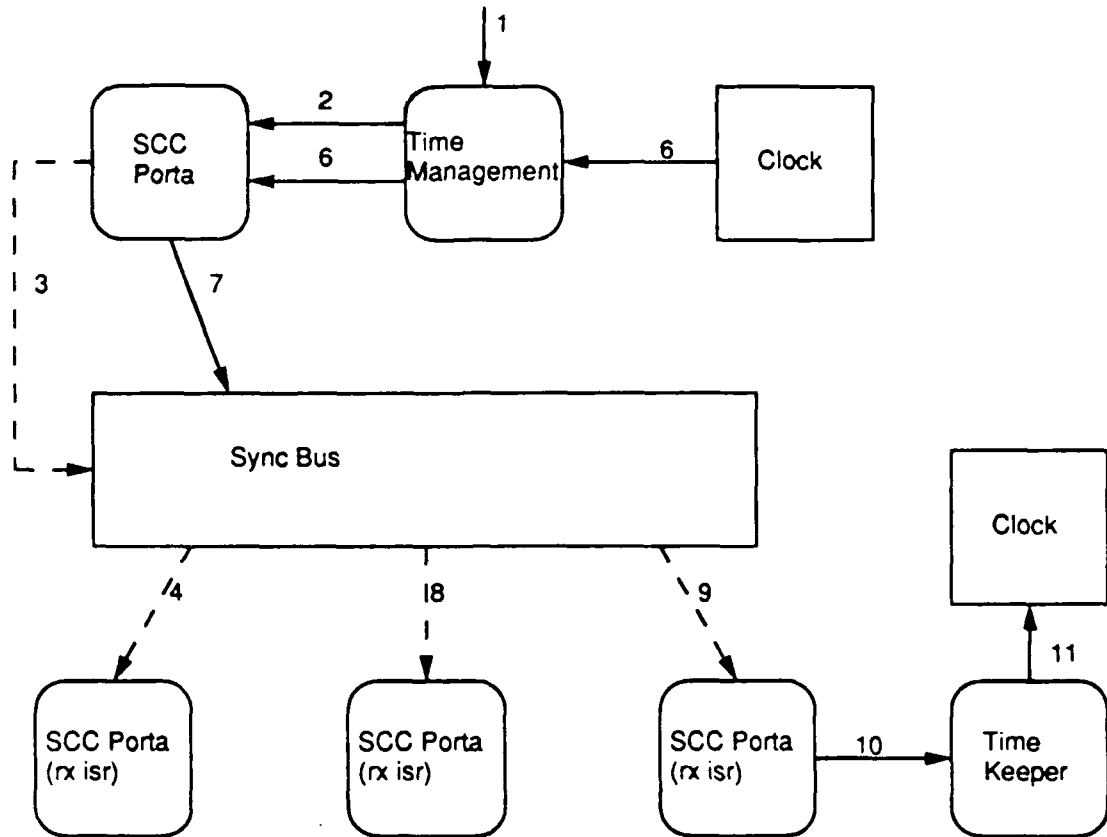
- The data are sent via the transmitter interrupt service routine (tx_isr). Sending data via the serial port functions as follows:
 1. The transmitter is primed with the first byte of data and transmitter interrupts are enabled.
 2. When a byte has been transmitted, a transmitter interrupt is generated on the sender.

3. The tx_isr sends the next byte in the message.
4. When the last byte in the message is sent, the tx_isr disables the transmitter interrupts.

The data is received via the receiver interrupt service routine (rx_isr). The rx_isr operates similarly to the tx_isr:

1. An interrupt is generated after a byte of data is stored in the receive data buffer.
2. The rx_isr stores the data in an incoming message buffer.
3. When the last byte of the message arrives, the rx_isr resets the epoch time. Note that, the actual event interrupt that processes the expired events won't occur until all of the rx_isr processing is complete, because it has a higher (and must always have) interrupt priority than the event timer.

Figure 35 illustrates the functioning of the sync bus described above.



1. Application calls *synchronize*.
2. *Synchronize* attempts to allocate the sync bus (thus acquiring bus mastership).
3. If no other node is currently using the sync bus, a *prepare_to_sync* message is transmitted.
4. The *prepare_to_sync* message is received by all other nodes, transitioning them into slave mode.
5. The current time of day is obtained,
6. Send first byte of *time_is_now* message.
7. The tx isr receives an interrupt after each byte of the message is sent. It completes sending the *time_is_now* message asynchronously.
8. All slave nodes receive bytes 1 through 7 of the *time_is_now* message.
9. Last byte (byte 8) of *time_is_now* message received.
10. *Adjust_epoch_time* is invoked to update the clock.
11. The local processor clock is reset to reflect the new time.

Figure 35: Sync Processing

6.1. Allocate

6.1.1. Interface

Allocate return boolean

6.1.2. PDL

Case sync mode is

When master =>

Disallow the process to allocate the bus, since some other process
(on this node) has a synchronization in progress

When slave =>

Disallow the process to allocate the bus, since some other process
(on another node) has a synchronization in progress

When idle => only in this state is the bus (presumably) available

Disable both rx & tx interrupts

Assert bus mastership

Send out the prepare_to_sync message

Wait for the data to be sent to all the nodes

Read back the sync character

If errors were detected in the prepare_to_sync message then a
collision occurred and this node must back off and wait

Transition back to slave mode and allow data to arrive

Wait for a specified time before reattempting to acquire

mastership (this node will either win mastership or

receive a sync byte from the winning node during this time)

Else

Transition to master mode

Inform the caller of successful allocation

End if

End case

6.2. Deallocate

6.2.1. Interface

deallocate

6.2.2. PDL

Case sync mode is

When idle or slave =>

null

When master =>

Set sync mode to idle

Reset sync bus to slave mode

Enable_rx_interrupts

End case

6.3. Disable_rx_interrupts

Since the enable/disable bits for both the Rx and Tx interrupts are stored in the same byte, software flags are maintained that indicate the current state of each interrupt.

6.3.1. Interface

```
disable_rx_interrupts
```

6.3.2. PDL

```
Case tx_enabled is
  When true =>
    Disable rx interrupt and enable tx interrupt
  When false =>
    Disable rx interrupt and disable tx interrupt
End case
Set rx_enabled to false
```

6.4. Disable_tx_interrupts

Since the enable/disable bits for both the Rx and Tx interrupts are stored in the same byte, software flags are maintained that indicate the current state of each interrupt.

6.4.1. Interface

```
disable_tx_interrupts
```

6.4.2. PDL

```
Case rx_enabled is
  When true =>
    Enable rx interrupt and disable tx interrupt
  When false =>
    Disable rx interrupt and disable tx interrupt
End case
Set tx_enabled to false
```

6.5. Enable_rx_interrupts

Since the enable/disable bits for both the Rx and Tx interrupts are stored in the same byte, software flags are maintained that indicate the current state of each interrupt.

6.5.1. Interface

`enable_rx_interrupts`

6.5.2. PDL

```
Case tx_enabled is
  When true =>
    Enable rx interrupt and enable tx interrupt
  When false =>
    Enable rx interrupt and disable tx interrupt
End case
Set rx_enabled to true
```

6.6. Enable_tx_interrupts

Since the enable/disable bits for both the Rx and Tx interrupts are stored in the same byte, software flags are maintained that indicate the current state of each interrupt.

6.6.1. Interface

`enable_tx_interrupts`

6.6.2. PDL

```
Case rx_enabled is
  When true =>
    enable rx interrupt and enable tx interrupt
  When false =>
    Disable rx interrupt and enable tx interrupt
End case
Set tx_enabled to true
```

6.7. Get

6.7.1. Interface

```
Get (register to read)
  return byte read from register
```

6.7.2. PDL

```
Select register to read
Return data in register
```

6.8. Get_byte

6.8.1. Interface

```
get_byte (data byte,  
          data valid)
```

6.8.2. PDL

```
If data is available  
  Read the data  
  Set the data valid to true  
  Reset any pending tx interrupts  
  Reset any pending rx interrupts  
Else  
  Set the data valid to false  
End if
```

6.9. Initialize

6.9.1. Interface

```
Initialize
```

6.9.2. PDL

```
Bind the receiver interrupt handler  
Enable the receiver interrupt  
Bind the transmitter interrupt handler  
Enable the transmitter interrupt  
Clear the rx data register (it is quad buffered)  
Set all the command registers  
Allow (in hardware) rx interrupts to occur  
Disallow (in hardware) tx interrupts to occur
```

6.10. Put_byte

6.10.1. Interface

```
put_byte (data byte)
```

6.10.2. PDL

```
Wait for a space in the tx data buffer to become available  
Place the data in the tx data buffer
```


6.11. Send

6.11.1. Interface

send (buffer address, data count)

6.11.2. PDL

Case on sync mode is

When master =>

 Disable rx interrupts (since tx_isr will read the loop back bytes)

 Copy the message into local storage

 Enable Tx interrupts

 Prime the tx buffer with the first byte of the message

When slave =>

 Reject the operation

When idle =>

 Reject the operation

End case

6.12. Set

6.12.1. Interface

set (register
 data)

6.12.2. PDL

Setup register to write

Write data to register

6.13. Rx_isr

6.13.1. Interface

N/A

6.13.2. PDL

If receive data errors are detected =>

 return, i.e., ignore the interrupt (the sender will
 retransmit the bad data)

End if

Acknowledge the interrupt

Case on sync_mode

when idle => node can safely transition to slave mode
 (no prepare_to_sync message has arrived)

```

    read the sync protocol byte
    transition to slave mode (i.e., listen-only mode)
    reset the received message counter
when slave =>
    get the next byte of message
    if this is the last byte of the message then
        reset the epoch time
        transition back into idle mode
    end if;
when master =>
    get the offending byte to clear the rx pending conditions
end case;

```

6.14. Tx_isr

6.14.1. Interface

N/A

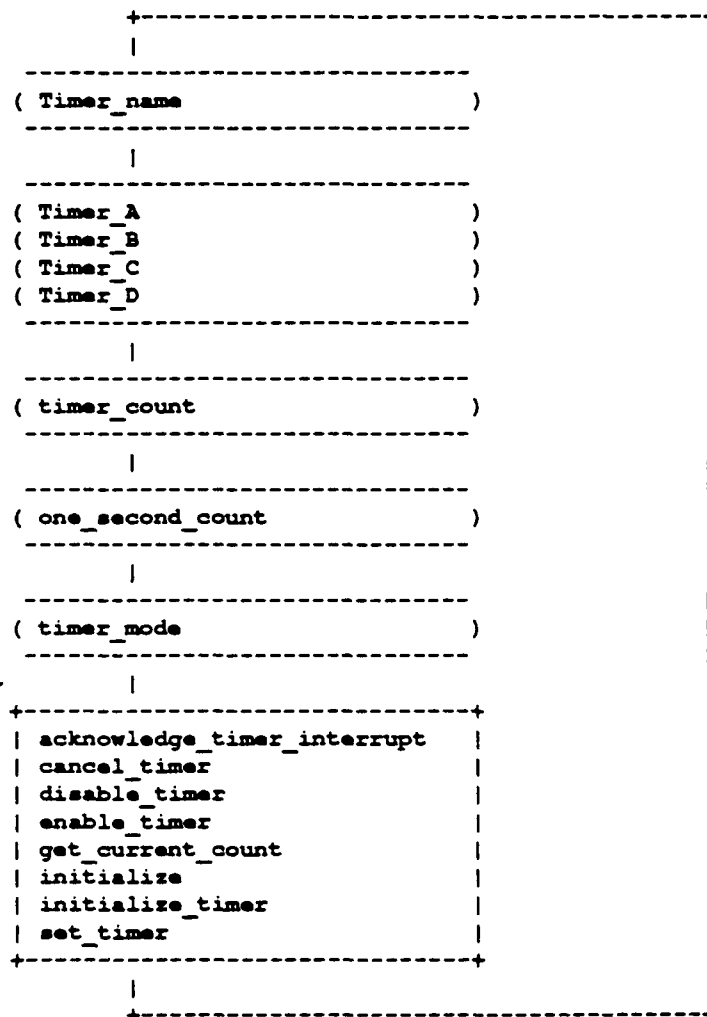
6.14.2. PDL

```

If errors were detected during transmission then
    Resend the bad character
Else
    Read back the last character sent (thus resetting any
        pending conditions)
    if the current byte count < the length of the message then
        Send the next byte of the message
    Else, the message is completely sent...
        Reset the output buffer parameters
        Wait for all the characters to be transmitted
        Read back the last character
        Disable and reset Tx interrupts
        Release bus mastership

```

7. Timer_controller



A description of the actual timer hardware is found in Part VIII. From a software perspective, the timers are simple devices that are loaded with a value and told to start. When the timers have decremented the initial value to zero, an interrupt is generated. The timers operation in two modes:

1. `Single_shot`: where the initial value is decremented to zero and then the timer is reloaded with the timer's maximum value and counting continues. This maximum count allows the user enough time to stop the timer before another interrupt is generated.
2. `Automatic`: where the initial value is decremented to zero and then the timer is reloaded with the initial value. This allows the user to generate an interrupt at a predictable rate.

The Kernel uses the `single_shot` mode for managing the time event queue (discussed in Chapter 13) and the `automatic` mode for the real time clock (discussed in Chapter 2).

7.1. acknowledge_timer_interrupt

7.1.1. Interface

`acknowledge_timer_interrupt(name of timer)`

7.1.2. PDL

`Reset timer interrupt`
`Reenable timer interrupt`

7.2. cancel_timer

7.2.1. Interface

`cancel_timer (name of timer)`

7.2.2. PDL

`Disable the timer`

7.3. Disable_timer

7.3.1. Interface

`disable_timer (name of timer)`

7.3.2. PDL

`Disable timer interrupt`
`Stop timer from counting`

7.4. Enable_timer

7.4.1. Interface

`enable_timer(name of timer,`
`timer mode)`

7.4.2. PDL

`Set timer to begin counting down in given mode`
`Enable timer to interrupt`

7.5. Get_current_count

Reading the current count does not stop the timer from counting or interfere with its processing.

7.5.1. Interface

```
get_current_count (name of timer) return timer count
```

7.5.2. PDL

```
Read current value of timer count (1)
Read current value of timer count (2)
Adjust result to allow for overflow/wrap around between
  read (1) and read(2)
Return adjust result
```

7.6. initialize

7.6.1. Interface

```
initialize (timer vector
           address of timer interrupt handler)
```

7.6.2. PDL

```
Bind the interrupt handler
Enable the interrupt handler
```

7.7. initialize_timer

7.7.1. Interface

```
initialize_timer (name of timer,
                 timer count,
                 timer vector)
```

7.7.2. PDL

```
Set timer to interrupt at timer vector
Set timer to count down from timer count
```

7.8. set_timer

7.8.1. Interface

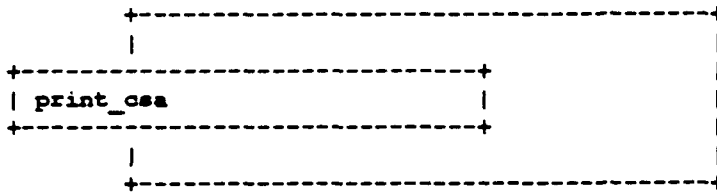
```
set_timer (timer name,  
           timer vector  
           time of interrupt)
```

7.8.2. PDL

```
If the timer value > the maximum countdown time then  
    Set the timer for its maximum countdown time  
Else  
    Convert the countdown time to timer ticks (each tick is 2 microsec)  
End if  
Enable the timer
```

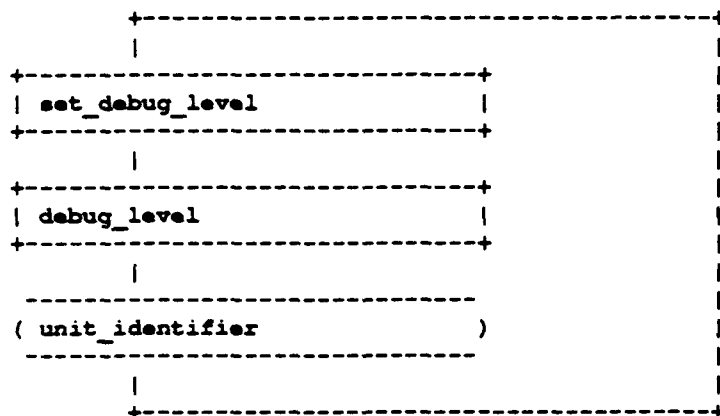

VII. Debug Utilities

1. CSA_debug



This utility prints the contents of the context save area associated with a particular process. It is 68020-specific.

2. Debug



This package contains an enumerated type of the short names of all the Kernel packages. Associated with each package is a debug level. Taken together, they are used to control debug out in the corresponding package. Currently, all debug output is controlled at the package level, but any unit name can be added to this package and used for a finer level of control.

Any new packages added to the system should be added to this list. In reality, any unit name can be added to the package and used to control debug output.

The debug levels are simple integers used to control the debugging output embedded in the Kernel packages. The exact meaning of any particular value depends on the package, but in general, there are two points:

1. The larger the number the larger the amount of debug data generated.
2. A value of zero turns off all debug output by a package.

For example, suppose that one wanted to turn on the debugging in the package `processor_management` (short name: `rm`). This would look like:

```
set_debug_level (rm, 100);
```

Subsequently, in the body of `processor_management` one would find the debug output like:

```
If debug_level(rm) >= 99 then
    text_io.put_line ("some truly important data...");
End if;
```

2.1. Set_debug_level

2.1.1. Interface

```
set_debug_level (unit short name,  
                debug level)
```

2.1.2. PDL

Set the current debug level for the unit to the specified value

2.2. debug_level

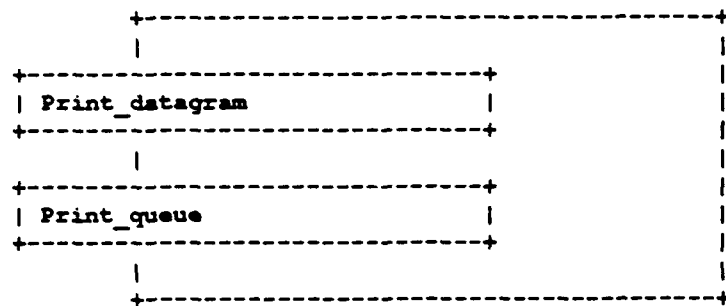
2.2.1. Interface

```
debug_level (unit short name) return integer
```

2.2.2. PDL

Return the current debug level for the name unit

3. dgg_debug



This package prints the contents of a single datagram or an entire queue of datagrams. Datagrams can be dumped in two ways:

1. Header information only
2. Header information and message text (as a block of byte data)

3.1. Print_datagram

3.1.1. Interface

```
print_datagram (pointer to datagram to print
                print whole datagram flag)
```

3.1.2. PDL

```
print contents of datagram header
if dump_whole_datagram then
    print contents of entire datagram buffer
else
    print first 6 bytes of datagram buffer
end if
```

3.1.3. Sample output

```
Datagram located at 16#10F528#
next => 16#1006F0#   prev => 16#1006F0#
class => SMALL
buffer_size =>      64
msg_count =>         0
semaphore =>         0
message_length =>    4
operation => BLIND_SEND
remote_timeout => microsec (high 32 bits) =>      0
                  microsec (low 32 bits) =>      0
sender =>    0 / -32766
receiver =>  1 / -32765
message_tag =>      0
message_id =>      0
checksum => 16#0#
buffer => 102 101 101
```

3.2. Print_queue

3.2.1. Interface

```
print_queue (pointer to datagram queue to print
            print whole datagram flag)
```

3.2.2. PDL

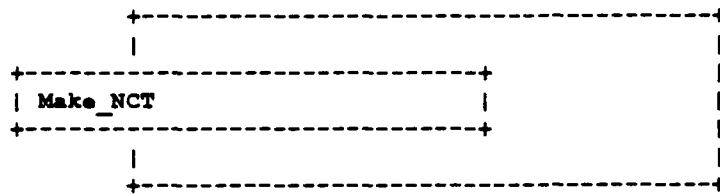
```
For every datagram in queue loop
    If dump_whole_datagram then
        Call print_datagram(dump_whole_buffer => true)
    Else
        Print selected header information
```

```
End if
End loop
```

3.2.3. Sample output

```
Datagram queue located at 16#1006F0#
== 2 datagrams in queue ==
Datagram located at 16#10F528#
  next => 16#10F430#   prev => 16#1006F0#
  class => SMALL
  sender =>    0 / -32766
  receiver =>   1 / -32765
  message_length =>      4
  operation => BLIND_SEND
  message_tag =>      0
  message_id =>      0
Datagram located at 16#10F430#
  next => 16#1006F0#   prev => 16#10F528#
  class => SMALL
  sender =>    0 / -32766
  receiver =>   1 / -32765
  message_length =>      7
  operation => BLIND_SEND
  message_tag =>      2
  message_id =>      0
```

4. Make NCT



This is procedure prompts the caller to select one of the available network configurations. It then fills in the NCT appropriately and returns the selected configuration to the caller. It is setup to work with the DARK hardware described in Chapter VIII.

4.1. Make_nct

4.1.1. Interface

make_nct (character indicating selected configuration)

4.1.2. Output

make_nct: The available network configurations are...

make_nct: 0> stand alone node 0

make_nct: 1> stand alone node 1

make_nct: 2> stand alone node 2

make_nct: 3> stand alone node 3

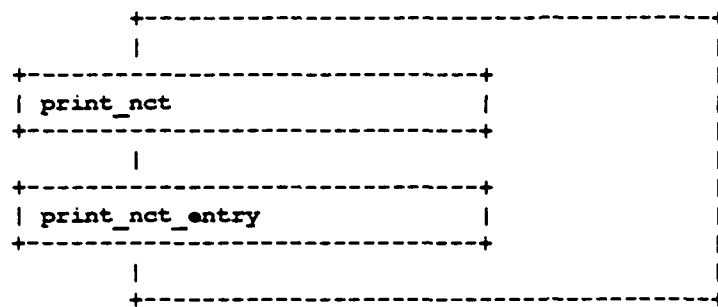
make_nct: a> 2 nodes (0/1/x/x) 0 master

make_nct: b> 2 nodes (2/3/x/x) 2 master

make_nct: c> 4 nodes (0/1/2/3) 0 master

make_nct: Enter your network configuration:

5. NCT_debug



This package prints all or part of the NCT.

5.1. Print_nct

5.1.1. Interface

Print_nct

5.1.2. PDL

```
For each entry in the NCT
  Print_nct_entry (the next NCT entry index)
End loop
```

5.2. print_nct_entry

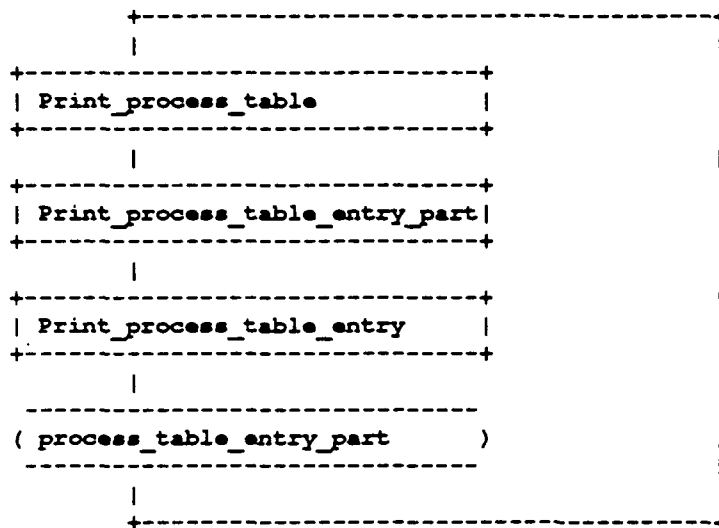
5.2.1. Interface

print_nct_entry (index of entry into NCT)

5.2.2. Sample Output

```
NCT:  **** BEGIN DUMP OF NCT ENTRY ****
NCT:  logical_name           => Kproc_a
NCT:  physical_address       => 0
NCT:  kernel_device          => TRUE
NCT:  needed_to_run          => TRUE
NCT:  allocated_process_id    => NULL
NCT:  initialization_order     => 0
NCT:  initialization_complete => FALSE
NCT:  ***** END DUMP OF NCT ENTRY *****
```

6. PTB_debug



Local procedures are defined to print each sub-record of a process table entry:

- process_attributes
- schedule_attributes
- communication_attributes
- pending_activity_attributes
- send_w_ack_attributes
- semaphore_attributes
- tool_interface_attributes

Each of these internal procedures takes a process identifier as an input parameter and dumps the appropriate section of the process table.

6.1. print_process_table

This procedure prints the entire process table entry for each process.

6.1.1. Interface

```
print_process_table
```

6.1.2. PDL

```
For each entry in the process table loop
    print_process_table_entry (process identifier)
end loop
```

6.2. print_process_table

This procedure prints only a selected part of the process table entry for each process.

6.2.1. Interface

```
print_process_table (process table entry part)
```

6.2.2. PDL

```
For each entry in the process table loop
    print_process_table_entry (process identifier,
                               process table entry part)
end loop
```

6.3. print_process_table_entry_part

6.3.1. Interface

```
print_process_table_entry_part (process identifier,
                                process table entry part)
```

6.3.2. PDL

```
Using the appropriate local procedure, print that part of
the process table for the selected process.
```

6.3.3. Sample Output

See print_process_table_entry below.

6.4. print_process_table_entry

6.4.1. Interface

print_process_table_entry (process identifier)

6.4.2. Sample Output

```
PTB:  **** BEGIN DUMP OF PROCESS TABLE ENTRY ****
PTB:  $$$ BEGIN PROCESS ATTRIBUTES $$$
PTB:  logical_name           =>  foo_1
PTB:  kind_of_process        =>  KERNEL_PROCESS
PTB:  process_init_status.declared =>  FALSE
PTB:  process_init_status.created =>  FALSE
PTB:  process_index.node_number  =>      0
PTB:  process_index.process_number =>      0
PTB:  code_address           =>      -1
PTB:  stack_low_address      =>      -1
PTB:  stack_high_address     =>      -1
PTB:  context_saved          =>  VIA_CALL
PTB:  program_counter        =>      0
PTB:  status_register        =>      0
PTB:  data registers
      [d0] =>      0
      [d1] =>      0
      [d2] =>      0
      [d3] =>      0
      [d4] =>      0
      [d5] =>      0
      [d6] =>      0
      [d7] =>      0
PTB:  address registers
      [a0] =>      0
      [a1] =>      0
      [a2] =>      0
      [a3] =>      0
      [a4] =>      0
      [a5] =>      0
      [a6] =>      0
      [a7] =>      0
PTB:  floating_point_coprocessor =>
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
      0      0      0      0
```

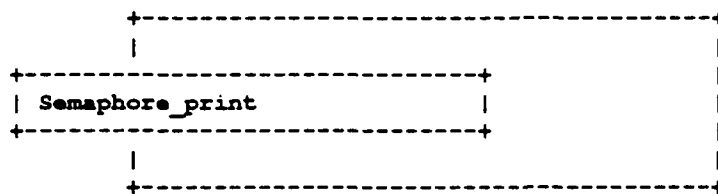
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

```

PTB: $$$$ END PROCESS ATTRIBUTES $$$$
PTB: $$$ BEGIN SCHEDULE ATTRIBUTES $$$
PTB: state => RUNNING
PTB: priority => 1
PTB: preemption => DISABLED
PTB: block_time.day => 0
PTB: block_time.second => 0.00000
PTB: unblock_time.day => 0
PTB: unblock_time.second => 0.00000
PTB: $$$$ END SCHEDULE ATTRIBUTES $$$$
PTB: $$$ BEGIN COMMUNICATION ATTRIBUTES $$$
PTB: next_available_message_ID => 0
PTB: maximum_message_queue_size => 0
PTB: message_queue => -1
PTB: current_send_buffer => -1
PTB: queue_overwrite_rule => DROP_NEWEST_MESSAGE
PTB: message_queue_overflow => FALSE
PTB: $$$$ END COMMUNICATION ATTRIBUTES $$$$
PTB: $$$ BEGIN PENDING ACTIVITY ATTRIBUTES $$$
PTB: pending_activity => NOTHING_PENDING
PTB: pending_event_ID => queue is empty
PTB: current_pending_message => -1
PTB: alarm_event_ID => queue is empty
PTB: alarm_resumption_priority => 0
PTB: exception_name => NO_EXCEPTION
PTB: $$$$ END PENDING ACTIVITY ATTRIBUTES $$$$
PTB: $$$ BEGIN SEND W ACK ATTRIBUTES $$$
PTB: event_ID => queue is empty
PTB: message => -1
PTB: queue => -1
PTB: $$$ BEGIN SEMAPHORE ATTRIBUTES $$$
!!! TBD for now !!!
PTB: $$$$ END SEMAPHORE ATTRIBUTES $$$$
PTB: $$$ BEGIN TOOL INTERFACE ATTRIBUTES $$$
!!! null for now !!!
PTB: $$$$ END TOOL INTERFACE ATTRIBUTES $$$$
PTB: ***** END DUMP OF PROCESS TABLE ENTRY *****

```

7. semaphore_debug



7.1. semaphore_print

7.1.1. Interface

`semaphore_print ( semaphore to print)`

7.1.2. PDL

Announce self with message and print opening delimiter
Print address of semaphore head
Print number of queued processes

For each queued process in the ordered enqueued loop
 Print the logical name of the process
End loop

For each semaphore previously claimed by the owning process
 in reverse claim order loop
 Print address of semaphore head
End loop

Print closing delimiter

7.1.3. Sample Output

```
* Contents of Semaphore                ffd1f0

[ number_of_waiting_processes =>      3

  Enqueued process => process_1
  Enqueued process => process_2
  Enqueued process => foo

  Previously claimed sema => ddc0c7
  Previously claimed sema => efc010
  Previously claimed sema => ff0143

]
```


VIII. 68020 Hardware Configuration

This chapter describes the DARK hardware testbed, in particular the MC68020 target hardware and its configuration.

1. Target Processor Board

The DARK testbed has four processor nodes distributed across a network. Each node comprises two processors: the Nproc and the Kproc. This chapter describes these processors in more detail. The Motorola MVME133A board is used to implement both the Kproc and Nproc.

1.1. MVME133A Board

The MVME133A is a single VME board computer module with many features required by embedded system applications. Refer to the *MVME133A-20 VME module 32-Bit Monoboard Microcomputer User's Manual* for further details on this board.

This list summarizes the MVME133A features:

- 20MHz MC68020 32-bit microprocessor.
- 20MHz MC68881 Floating Point Coprocessor (FPC).
- 1 megabyte (MB) of shared dynamic RAM, 32-bit wide.
- 32-bit address and data bus VME master (A32/D32), and compatibility with A32/D16, A24/D32, and A24/D16. 4-gigabyte address space.
- Sockets for ROM, PROM, or EPROM chips, 256 kilobytes (KB) max.
- 1 user-programmable, 8-bit timer.
- Time-of-day clock, 100ms resolution.
- 1 asynchronous serial port, and two async/sync serial ports.
- VME bus controller functions, master or slave.
- VME bus interrupter.
- VME bus interrupt-handler logic.

1.1.1. Local Memory

There is a total of 1 MB of Dynamic Random Access Memory (DRAM) on the MVME133A board. It is shared so either the MC68020 or the VME bus master can access on-board memory, but not at the same time. This memory is used to store the object code, but can also be accessed from the VME bus.

Each MC68020 access to on-board DRAM requires four clock cycles (200ns, three minimum plus a wait state). However, during the Read-Modify-Write (RMW) sequence it may take more.

Local memory can have a different range of addresses when accessed locally or from the VME bus. Memory is always accessed locally with addresses between 16#00000# and 16#1FFFF#. However, this same 1 MB address space can be accessed relative to a different base address that is set with jumpers.

1.1.2. Floating Point Coprocessor

The Motorola MC68881 Floating Point Coprocessor (FPC) is a full implementation of the IEEE standard for binary floating-point arithmetic. It provides a logical extension to the MC68020 microprocessor and operates at the same frequency, 20MHz.

1.1.3. Real-Time Clock

The Real-Time Clock (RTC) on the MVME133A is an MM58274 chip. It provides a time-keeping function from tenths of seconds to tens of years. It can generate interrupts to the MC68020 at regular intervals with a 0.1-second resolution.

1.1.4. Serial Debug Port

The Debug port is provided by a Motorola MC68901 chip. The connector for this port is located on the front panel of the MVME133A board. It is an RS-232-C compatible port, configured for Data Communications Equipment (DCE) only. It is called the Debug port because when the debug ROMs are used, it is programmed to be used for interactive debugging. No other port may be configured for this purpose. The debug port may operate at all the standard baud rates between 110 and 19,200.

1.1.5. Serial Ports A and B

The MVME133A uses the Z8530 Serial Communications Controller (SCC) chip to implement its two multi-protocol serial ports, which provide multi-function support for handling a large variety of serial communications protocols. The Z8530 may be programmed to follow standard formats such as byte-oriented synchronous, bit-oriented synchronous, and asynchronous. Port A of the Z8530 is connected to on-board RS-485/422 drivers and receivers. Port B of the Z8530 is connected to on-board RS-232C drivers and receivers. The baud-rate clock for both channels may be obtained from several on-board sources.

Port A, with RS-485/422 drivers, is routed to the P2 connector. It may be configured by software to be either master or slave and half or full duplex.

Port B, with RS-232C drivers, is also routed to the P2 connector. It may be configured either as DTE or DCE, by setting jumpers.

1.1.6. Timers

The MVME133A has four on-board timers. Three of them are not available to the programmer, because they are assigned to do other functions; debug port baud rate generator, tick timer, and watchdog timer. All of the timers are part of the MC68901 chip, which also provides a serial port (debug port), and general purpose I/O pins used for status and control. The only timer available to programmer is eight bits wide and can be used for any purpose.

1.1.7. Interrupts

The MVME133A board provides logic for interrupt handling and an interrupter. The interrupt handler gives the on-board MC68020 the ability to sense and respond to all on-board and off-board (VME bus) interrupts. The board may be jumpered to enable or disable any combination of the seven interrupt request lines.

The interrupter can generate interrupts on interrupt request level 3.

1.1.8. ROM, PROM, EPROM, and EEPROM Sockets

The MVME133A has four IC sockets that are organized into two banks. These sockets provide up to 256 KB of extra ROM. The sockets hold the debug ROMs or any other user-programmed ROMs.

1.1.9. VME System Controller

The system controller on the MVME133A implements a level 3 VME bus arbiter, VME bus requester, and Interrupt ACKnowledge (IACK) daisy-chain driver. The arbiter and IACK daisy-chain functions are designed to meet the VME bus specification.

The VME bus requester is used to obtain and relinquish master control of the VME bus. It can request VME bus master control on any one of the four request levels depending on how it is jumpered. It requests master control of the VME bus any time the MVME133A is not the current VME bus master and the MC68020 indicates it requires access to the VME bus.

The control function arbitrates all VME bus requests so that only one requester of the bus actually gets control. The local microprocessor has to compete evenly with all other devices to get master control of the bus.

The IACK daisy-chain function is a mechanism to acknowledge interrupt request in some orderly fashion. Due to the way the daisy-chain works, the physical position of VME boards in the chassis is significant.

1.1.10. P1 And P2 Connector

The MVME133A board attaches to the VME backplane at the connectors P1 and P2. P1 couples most of the required VME bus signals to and from the backplane. The P2 connector couples all of the optional VME bus signals and the signals for serial ports A and B.

1.2. Kernel Processor Board Configuration

This section lists all of the jumpers on the MVME133A board and indicates how they are set for the board identified to be the Kernel processor. More detail on the function and location of each jumper may be obtained from the MVME133A user's manual.

- J1 – System Controller Enable Jumper – set to disable the system controller function.
- J2 – Onboard RAM offset address select header – consists of four jumpers set for an offset address of 16#200000#.
- J3, J4 – VME bus requester level select headers – consists of six and three jumpers, respectively. These have been left in their factory settings to couple all four bus grant lines through the board and establish a level 3 arbiter.
- J5 – RMW cycle type select jumper – has been left in its factory setting; the processor must obtain master control of the VME bus to execute read-modify-write accesses.
- J6, J7 – ROM/PROM/EPROM size headers – consist of three jumpers each. These have been left in their factory settings and have been configured for the Debug 133A ROMS.
- J8 – Global timeout jumper – no jumper installed. This jumper is only important for boards that have been set up as system controllers.
- J9 – Reset switch jumper – set to enable the reset momentary button on the front panel.
- J10 – Abort switch jumper – set to enable the abort momentary button on the front panel.
- J11 – VMEbus interrupter jumper – set to enable the interrupter logic. Specific instructions were given not to change this from the factory setting.
- J12 – VMEbus interrupt handler header – consists of seven jumpers. These jumpers are set to permit the Kproc board to handle interrupt requests on IRQ7,IRQ6,IRQ3,and IRQ1 lines.
- J13 – Serial port B configuration header – consists of 11 jumpers. These have been left in their factory settings for port B to operate as a DCE device.
- J15 – Software readable header – consists of five jumpers. They are set differently for each board. J15 can be read from software, and it is used to indicate the physical number of the board.
- J16 – Serial ports RTXCx source select header – consists of two jumpers. These have been left in their factory settings.
- J17 – VMEbus data width select jumper – may be placed in one of several positions. It is set to indicate either a 16-bit or 32-bit, or both 16-bit and 32-bit data path during VME I/O accesses. It has been left in the factory setting to allow both 16-bit and 32-bit width data.
- J18 – VMEbus address size select Jumper – is similar to J17, except it is set to choose the size of the address on the VME bus, either 32-bit or 24-bits wide. It is set for 32-bit wide addresses.
- E1, E2 – Cache disable test points – may be wire-wrapped together to disable the MC68020 cache. They are normally left untouched, but during performance analysis, the cache is disabled to use the logic analyzer.

1.3. Network Processor Board Configuration

This section covers the jumper settings for the Nproc board. Since most of the Nproc jumpers are set the same as the Kproc jumpers, only the ones that differ are listed. More detail may be obtained on the function of each jumper by referencing the *MVME133A Users Manual*.

- J1 – System controller enable jumper – set to enable the system controller function.
- J8 – Global timeout jumper – set to enable the global timer. This jumper is only important for boards that have been set up as system controllers.
- J12 – VMEbus interrupt handler header – consists of seven jumpers. These jumpers are set to permit the Kproc board to handle interrupt requests on IRQ5, IRQ4, IRQ2, and IRQ1 lines.
- J15 – Software readable header – consists of five jumpers set differently for each board.

2. Parallel Interface

The network in the DARK testbed consists of segments of 32 parallel lines connecting adjacent Nproc. The parallel interface used to connect the Nproc to each segment of the network is the Mizar MZ8305 Quad Parallel Port module. Two of these modules are required per node.

2.1. MZ8305 Board

The MZ8305 is a single height VME bus compatible board. It connects to P1 of the VME bus backplane (upper connector). At the heart of the MZ8305 board are two Motorola 68230 Parallel Interface/Timer (PI/T) chips, designated #1 and #2. The board has a total of 32 bits of buffered parallel I/O and two programmable timers. Refer to the *MZ8305 Quad Parallel Port Module User's Manual* for more details on this board.

All on-board addresses are mapped to a 16-bit I/O address space. The base address can be jumpered to any 256-byte boundary. All of the registers on the MZ8305 are contained in the two PI/Ts. Each PI/T has 32 8-bit registers.

There are four interrupt sources on the MZ8305. Each can be jumpered to drive any one of the VME bus 7 interrupt request levels. However, no two sources can drive one line.

2.2. Parallel Interface/Timer

Both 68230 PI/T chips have three 8-bit ports, designated ports A, B, and C; one (port C) is used for controlling associated interrupts, buffering logic, and handshake lines; and two (ports A and B) are used for parallel I/O. The parallel ports can be programmed for input, output, or bi-directional I/O. There are two handshake lines per I/O port, designated H1, H2, H3, and H4. The PI/T can be programmed to generate interrupts when data are received at port A or B.

Each PI/T also has a 24-bit programmable timer, which may be programmed to generate interrupts periodically, or after a specified period of time. The timer input clock may come from an external source, through one of the parallel I/O connectors, or from on-board circuitry at one of the following frequencies: 500, 250, 125, or 67.5 KHz (2, 4, 8, 14.8 μ sec periods, respectively).

2.2.1. Parallel I/O Connector

The MZ8305 board has two 50-pin connectors, designated J1 and J2. These connectors provide access to the I/O and handshake lines, the timer input clock, and timer output for each PI/T.

2.3. Input Port Parallel Board Configuration

This section covers the jumpers on the input parallel port board. Many of the jumper settings are the same on both the input port and output port boards. The function of each jumper is explained in more detail in the hardware reference, *MZ8305 Parallel Interface Board User's Manual*. The following is a list of all the jumper blocks and their settings for this board:

- K08 – Address select – All jumpers are left installed so that the base addresses for PI/T #1 and #2 are 16#FFFF0040# and 16#FFFF0000#, respectively.
- K09 – Interrupt request – The jumpers are wire wrapped so that P1 is connected to IRQ4, P2 is connected to IRQ5, T1 is connected to IRQ7, and T2 is connected to IRQ6.
- K05 – Interrupt ACK for PIO – Jumpers are installed at A02 and A01 for PIRQ1 and at A02 for PIRQ2.
- K06 – Interrupt ACK for timers – A jumper is installed only at A01 for T_{out2}.
- K04, K10 – T_{out1}/T_{out2} enable – Both these jumpers are installed.
- K01 – T_{out1}/T_{out2} and buffer control – Only four jumpers are installed. They connect pin 3 to 13, 6 to 16, 7 to 17, and 10 to 20.
- K03 – T_{in1}/T_{in2} input enable – Both jumpers are removed.
- K07 – Timer frequencies – One jumper is placed so that T_{in1} and T_{in2} connect, and a second jumper is installed to connect T_{in2} to 500KHz.
- K02 – H2/H4 direction – All jumpers are installed so that the handshake lines operate in the out direction.

2.4. Output Port Parallel Board Configuration

Only one jumper is set differently on the output port parallel board from those on the Input Port Parallel Board configuration:

- K08 – Address select – One jumper has been removed from the factory setting (A11) so that the base address for PI/T #1 is 16#FFFF0140# and for #2 is 16#FFFF0100#.

3. Shared Memory

This section covers basic information on the shared memory boards. The testbed uses Motorola MVME225-1 shared memory boards for the global memory on each processor node. This board is a full height VME board with 1 MB of DRAM. Refer to the *MVME225-1 1Mb Dynamic Memory Module User's Manual* for more details on this board.

The processor boards are designed so that the local processor always sees its local memory at the same location (starting at 16#00000000#). However, from off-board, the memory appears at a different location. Because of the different address mappings, the two processors are never able to see the same memory location using the same address. This would only be possible with a separate memory board such as the MVME225-1.

The base address of the shared memory is set with jumpers. Since the board doesn't support parity, a jumper must be installed to disable parity checking.

3.1. Shared Memory Board Configuration

The shared memory board has four jumper blocks. This section covers how they have been set to operation in the DARK testbed.

- K4 – Test Connector – Pins 15 and 16 are jumpered to disable parity checking, since the MVME133A processor boards do not check or generate parity information.
- K3 – Address Mode Select – All jumpers are removed to select continuous address space.
- K1, K2 – Address Select – Jumper A20 is removed to set the base address of the memory at 16#100000#.

4. VME Chassis

The DARK testbed uses four Motorola MVME945 chassis to hold all of the processor node hardware, namely two MVME133A CPU boards, shared memory board, and two MZ8305 Quad Parallel Port VME modules, as shown in Figure 36. The MVME945 chassis is designed to house industry standard VME modules, and may be used for desk-top or rack-mounted operation. The VME945 card cage holds up to 12 full height VME boards. The boards are inserted into guides from the front of the chassis. On the rear of the chassis, the P2 connectors can be accessed for connecting I/O cables or setting bus jumpers. Refer to the *MVME945 Chassis User's Manual* for more details on the location and function of the jumpers.

The power supply is approved for 400 watts. It requires one 115 VAC outlet, and supplies the following voltages:

- +5 VDC @ 50A
- +12 VDC @ 10A
- -12 VDC @ 5A

The chassis contains a 12-slot, 32-bit VME bus backplane and provides forced-air cooling for the VME modules. The VME modules are cooled by air drawn in from the bottom of the chassis by two fans, forced past the VME modules, and blown out the top of the chassis.

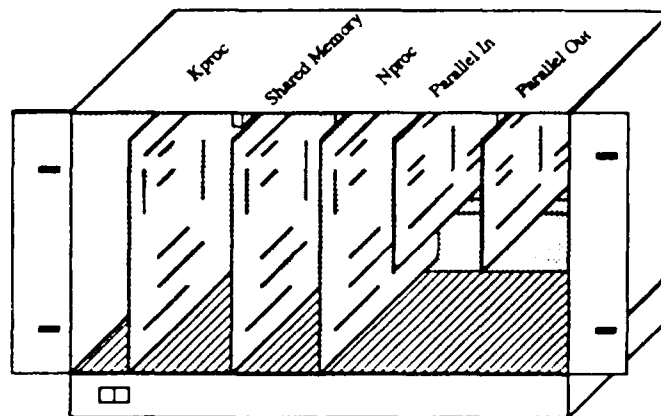


Figure 36: Chassis Hardware

4.1. Backplane Jumper Configuration

Each of the 12 P1 connectors, except slot 1, has two sets of jumpers that have to be configured for any of the boards plugged into the backplane to operate correctly. Jumpers are required in all empty slots and are removed for slots containing boards. The jumpers pass signals through each empty slot and on to the next.

The following is a list of the sets of jumpers:

- Bus Grant Signal headers, J3 through J12 – have four jumpers BG0, BG1, BG2, and BG3. These signals are part of the bus arbitration logic and are necessary to control which device gets control of the bus.
- The IACK Signal headers, J13 through J23 – has only one jumper each. These jumpers are used to form a daisy-chain for acknowledging interrupts.

The five VME boards are assigned to slots in the chassis. The order of the boards determines part of their bus request and interrupt priority. The only absolute requirement is that the system controller be in slot 1. The order in which the boards are inserted into the chassis in the DARK testbed, from left to right:

1. Slot 1 – Nproc MVME133A processor
2. Slot 5 – Kproc MVME133A processor
3. Slot 10 – Shared Memory MVME225-1
4. Slot 11 – Input Port Parallel Interface MZ8305
5. Slot 12 – Output Port Parallel Interface MZ8305

Again, except for slot 1, which does not have any jumpers, all of these slots have their respective jumpers removed, so the signals do not bypass boards.

5. Equipment Rack

The four VME card chassis are mounted in two 19-inch equipment racks. Two doors, located on the front and rear, provide access to the chassis mounted inside. The front door is tinted Plexiglas, while the rear door is sheet metal with ventilation holes. The chassis are mounted horizontally, one on top of the other, as shown in Figure 37. The rear door permits access to the rear of the chassis and the cabling between each processor node. One surge-protected, multi-plug, power strip distributes AC power to two chassis in each rack.

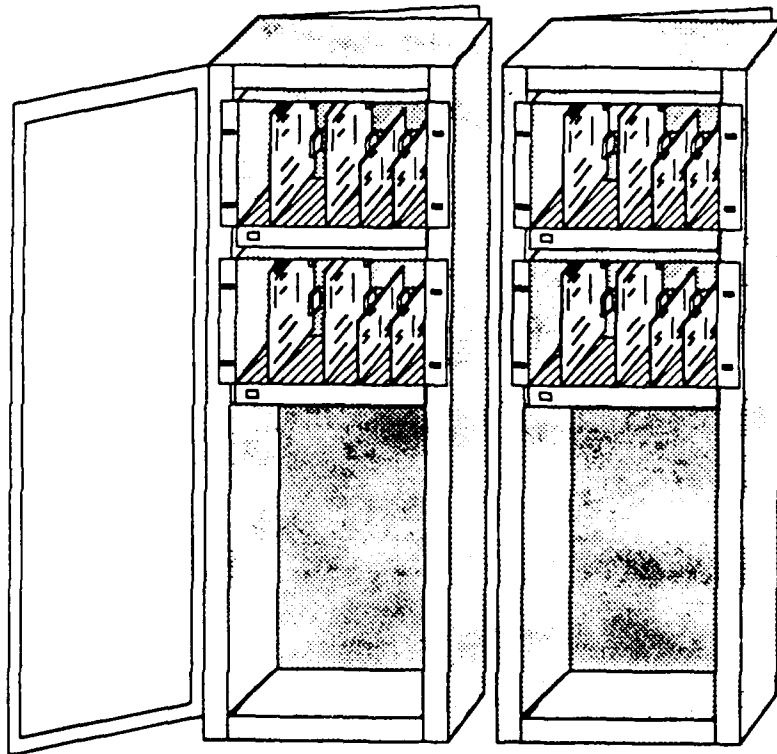


Figure 37: Equipment Rack

6. Host System

This section discusses the host system used to develop and download the Kernel and application.

The host system consists of four μ VAX-II computers clustered together and tied into the SEI local area network. They are called DARKA, DARKB, DARKC, and DARKD. DARKB and DARKC are the only μ VAX-IIs connected to the processor nodes, as shown in Figure 5. The others are only used for code development. Four serial communication lines (debug port and a downloading port for the Kproc and Nproc) run between each node and the DARKC μ VAX-II.

During the development process, the host machines provide support for many activities. The software is created on a project member's workstation, then transferred to one of the hosts, or entered directly on a host. After compilation and linking, the executable images are downloaded using either the Debug ROM downloader, or Telesoft's download facility.

Using both the debug ROM and Telesoft's Ada development system, it is possible to debug software while it is on-line and operating. The debug ROMs provide only basic debug operations, while the Telesoft symbolic debugger is more versatile and able to show how execution is proceeding relative to the Ada source code.

An alternative means of downloading, called VMSLink, is also available. It consists of VMSLink software, which can be run from any host machine in the cluster, and an ethernet board in each chassis.

6.1. Serial I/O Ports

Two out of three ports available on each processor board are used during the development. The debug monitor and TeleGen2 runtime use the debug port, and the TeleGen2 download system uses port B. The host system is not connected to Port A on either the Kproc or Nproc.

In total, there are 16 serial lines connected to the host system. To handle all of these serial lines, DARKC is equipped with two DHQ11 serial cards. There are three cards (two installed in DARKC and one in DARKB) that provide a total of 24 asynchronous serial lines for the host system.

Figure 38 is a cross reference between DARKC host ports and the serial ports on the processor boards in the testbed. For example, CPU 0 is the Kproc on Node 0. The TXA0 port of DARKC is connected to the debug port on CPU 0.

CPU #	Processor	Node Name	Debug Port	Download Port
CPU 0	Kproc	Node 0	TXA0	TXB0
CPU 1	Nproc	Node 0	TXA1	TXB1
CPU 2	Kproc	Node 1	TXA2	TXB2
CPU 3	Nproc	Node 1	TXA3	TXB3
CPU 4	Kproc	Node 2	TXA4	TXB4
CPU 5	Nproc	Node 2	TXA5	TXB5
CPU 6	Kproc	Node 3	TXA6	TXB6
CPU 7	Nproc	Node 3	TXA7	TXB7

Figure 38: VAX Ports to Testbed Ports Cross Reference

7. Test Equipment

This section covers the test equipment used to debug, monitor, and measure performance of the Kernel and the application. Electronic test equipment is used in the DARK Project for several activities, including:

- Measuring performance
- Troubleshooting hardware problems
- Monitoring software efficiency
- Debugging software

During DARK software development, the performance of critical sections of Kernel code are measured, and if not found to be satisfactory, optimized. The test equipment is used as a tool to help fine-tune the Kernel's performance. The final performance measures are taken and used to compare against the required performance measures recorded in the *Kernel Facilities Definition*.

The test equipment is used to help isolate problems when hardware components fail.

The operational behavior of the demonstration application running on DARK will be fully understood only when it has been implemented and tested. The test equipment is used to locate and verify correction of certain efficiency, or timing, problems. The test equipment also has the capability to help trace and debug certain kinds of logic errors.

7.1. Test Equipment Hardware

The DARK team had several different pieces of test equipment available for use, including:

- Tektronics 2223 Analog Oscilloscope
- Gould K115 Logic Analyzer
- Gould Microprocessor disassembly pod
- Gould CLAS 4000 with support for 2 MC68020 boards.
- Tektronics 2432A Digital Storage Scope
- Volt Ohm Meter
- XYComm VME Board Extender

8. Low-Level I/O

The Kproc and Nproc software accesses features of the target hardware through low-level I/O. Low-level I/O is necessary for two reasons. First, the compiler may not provide facilities required to accomplish a needed hardware operation. Second, although such facilities may be provided, they may not satisfy performance requirements of the application or the Kernel.

Low-level I/O is usually necessary to access special status information controlling the operation of devices on the target hardware, or for access memory directly. Status may be needed for such things as determining the state of a device, synchronizing with another device, detecting error conditions, or polling. The target hardware has several devices that must be controlled through low-level I/O by setting or changing the mode of a device, sending interrupts, or setting status in some port or device register.

Low-level I/O is accomplished through read or write operations to registers or memory locations. The MVME133A 68020 processor board is designed so that all device registers are mapped, or assigned, to have addresses in the address space of the MC68020 microprocessor.

Since the software that performs low-level I/O operations is highly dependent on the target hardware, it has been isolated to several Ada packages.

The following devices are part of the MVME133A 68020 microprocessor board and are involved in low-level I/O operations:

- Local ROM banks
- Local dynamic RAM
- Multi-Function peripheral – MFP
- Module Status Register – MSR
- Serial I/O port B – SIOB
- Serial I/O port A – SIOA
- Real-Time Clock – RTC
- Interrupter
- Parallel Interface and timer – PI/T

These devices are listed in the order they appear in the address space of the MVME133A 68020 microprocessor. The devices external to the the MVME133A board (e.g., PI/T) exist at higher addresses than the interrupter.

Each processor board (Nproc and Kproc) in a node has its own devices to control. The Kproc uses low-level I/O to access the shared memory. The Nproc uses low-level I/O to control the four 68230 Parallel Interface/Timer devices on two MZ8305 parallel interface boards and also to access shared memory.

8.1. Software

The Telesoft compiler provides support for low-level I/O in three basic ways: Machine Code Insertion (MCI), address clause, and imported subprograms.

MCI is provided through a package called `Machine_Code`. MCI allows single machine instructions to be inserted in-stream of the Ada source code. The compiler will make sure that the inserted low-level machine instructions are placed in the appropriate place among other machine instructions generated as a result of the high-level Ada statements. The Telesoft MCI supports all of the machine instructions available for the 68020 microprocessor.

The address clause for objects may be used to access hardware memory, registers, or other known locations. Symbolic names may be given to physical memory locations and accessed as if they were variables using the address clause.

The Ada programming language defines a pragma "interface" that allows Ada to interface with subprograms written in other languages, including assembly. The pragma interface statement declares a subprogram that will be imported and used in the Ada source code.

9. Interrupts

The mechanism for initiating and handling interrupts on the VME bus and MC68020 processor is well established and documented; therefore only a few important details of interrupt handling and the arbitrary and specific assignments made for the DARK hardware implementation are covered in this section. Refer to the *MC68020 32-Bit Microprocessor User's Manual* and VMEbus Specification document.

9.1. Interrupt Request Levels

Each interrupt source interrupts on one of seven interrupt request lines (IRQ1-IRQ7). These lines are prioritized, with IRQ7 having the highest request priority. Some of the interrupt sources have been permanently assigned to a particular request level. The rest may be assigned through jumpers (with some limitations). Generally, more than one interrupt source may be assigned to one interrupt request level.

There are jumpers on each processor board to connect each interrupt request line from the VME bus to the boards' interrupt handling logic. Since the function of the Kproc and Nproc differ, the two boards' interrupt request lines are jumpered differently. All on-board interrupts bypass these jumpers.

9.2. Interrupt Vector Numbers

During interrupt processing an interrupt vector number is obtained for each acknowledged request. An interrupt source either provides its own vector number or one is generated for it based on the interrupt request line it issues the request on. The processor uses the vector number to associate an interrupt service routine with the interrupt source. Each interrupt source must have a unique interrupt vector number, unless the service routine handles more than one interrupt source with respect to one processor.

9.3. Interrupt Configuration Summary

Figure 39 summarizes how the different interrupt sources are configured with respect to interrupt request levels and interrupt vectors. If a particular device is shown once when in fact there is more than one, all have been configured the same. For example, in one chassis, there are two MC68901 devices (one on each processor board). Both are set up to have the vector numbers shown. This does not conflict with the rule about unique vectors for each interrupt source, because these are local to a processor board, and the other processor cannot respond to them. On the other hand, the interrupts from the parallel interface boards must be unique across the VME bus, because both processors can potentially respond to these interrupts.

Board Name	Device	Interrupt Source	Interrupt Vector	Interrupt Request Level
Parallel I/O "In" Port				
	MC68230 PI/T #1	PIO	16#42#	IRQ4
		Timer (24A)	16#44#	IRQ7
	MC68230 PI/T #2	PIO	16#4A#	IRQ5
		Timer (24B)	16#4C#	IRQ6
Parallel I/O "Out" Port				
	MC68230 PI/T #1	PIO	16#52#	IRQ4
		Timer (24C)	16#54#	IRQ7
	MC68230 PI/T #2	PIO	16#5A#	IRQ5
		Timer (24D)	16#5C#	IRQ6
Kproc & Nproc				
	Abort Logic		16#1F#	IRQ7
	ACFail Logic		16#1F#	IRQ7
	MC68901 MFP	Timer (8A)	16#6D#	IRQ5
		Timer (8B)	16#68#	IRQ5
		Timer (8C)	16#65#	IRQ5
		Timer (8D)	16#64#	IRQ5
	Z8350 SCC	Port A TX	16#78#	IRQ6
		Port A RX	16#7C#	IRQ6
	MM58274	Real-Time Clock	16#1C#	IRQ4
	Interrupter Logic		16#FF#	IRQ3
EXOS 202				
	Ethernet Driver		Unknown	IRQ6

Figure 39: Interrupt Summary

10. Memory Map

Figure 40 lists all of the major devices and their base address assignments. Each assignment is identified as either hard-wired or jumpered. The hard-wired addresses may not be changed; the jumpered addresses may change.

Addresses are assigned so they do not conflict with each other. All of a processor board devices, except memory, are not visible to the other processor (off the board). This means, for example, that the real-time clock on each processor board can have the same address.

Board Name	Reference	Address Range
Kproc & Nproc	Local Memory	16#00000000# - 16#000FFFFFF#
Memory	Shared Memory	16#00100000# - 16#001FFFFFF# ⁹
Kproc	Nproc Memory	16#00200000# - 16#002FFFFFF#
Nproc	Kproc Memory	16#00300000# - 16#003FFFFFF#
Kproc & Nproc	Module Status Register	16#FFF80000# ¹⁰
Kproc & Nproc	Multifunction Peripheral	16#FFF80001# ¹¹
Kproc & Nproc	Serial Communication Controller	16#FFFA0000# - 16#FFFA0003#
Kproc & Nproc	Real-Time Clock	16#FFFB0000#
Kproc & Nproc	Interrupter Logic	16#FFFB8000#
Parallel I/O "In" Port	PI/T #1	16#FFFF0040# - 16#FFFF006F#
Parallel I/O "In" Port	PI/T #2	16#FFFF0000# - 16#FFFF002F#
Parallel I/O "Out" Port	PI/T #1	16#FFFF0140# - 16#FFFF016F#
Parallel I/O "Out" Port	PI/T #2	16#FFFF0100# - 16#FFFF012F#
EXOS 202	Status Register	16#FFFF8000#

Figure 40: Memory Map

⁹Shared memory is accessed over the VME bus in this address range by both the Nproc and Kproc.

¹⁰This is the first of many even-valued addresses at which this register can be referenced.

¹¹This is the first address of a group of registers.

11. Network Cable

The network data path extending between each node is made of two flat ribbon cables. These cables are specially constructed to handle the handshake lines and terminate unused pins.

All eight pieces of flat ribbon cable (two per node-to-node segment) are constructed the same. Signals H1 and H3 on both ends are crossed to H2 and H4 on the other end, respectively. Signals T_{in} and T_{out} on both ends are not connected. See Figure 41 for a schematic of the cable.

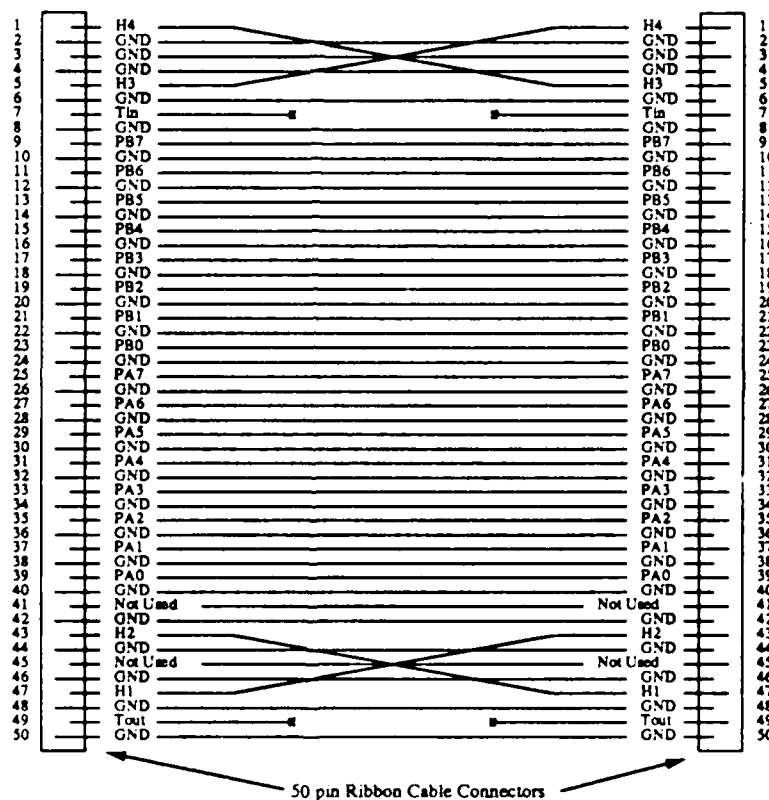


Figure 41: Flat Ribbon Cable Schematic

12. Synchronization Bus

The synchronization bus, or "sync bus" for short, is used to provide a way to broadcast time for the Kernel during time synchronization. All of the Kprocs on the network are connected to the sync bus.

The Kernel software that processes the time synchronization event is interrupt-driven. So, when synchronization occurs, all Kprocs respond immediately with the minimum amount of delay.

12.1. Bus Description

The interface to the sync bus is made through port A on the P2 connector on the Kproc board. Port A is one of two serial ports on the Z8530 SCC chip. Unlike port B, which uses RS-232C line drivers, port A uses RS-485/422 line drivers. This particular line driver permits one master (at a time) and multiple slaves.

The sync bus consists of two wires that carry a differential voltage, asynchronous serial data signal.

Figure 42 is a schematic of the sync bus (also refer to the schematic of the P2 connector in Figure 43). The sync bus schematic shows how the two wires of the bus are connected to the Send Data (SD), and Receive Data (RD) lines of port A's RS-485/422 drivers. Note that the SD and RD lines are connected together at the bus cable. This is so that the master can monitor for data collisions that would result if another Kproc attempts to become master at the same time.

The monitoring is required because the designers of the MVME133A board did not implement all of the RS-485/422 handshake signals, such as RTS, CTS, and DCD. If, in the future, the Kernel is ported to another target and this same approach for time synchronization is used, a more complete implementation of the RS-485/422 handshake signals is recommended.

12.2. Bus Operation

The relative frequency of time synchronization should be low. Normally, when no synchronization is in progress, all the Kprocs are set for slave operation. Port A is set for slave operation when:

- Transmitter output signal is electrically isolated from the SD signal lines.
- Receiver data input is monitoring the RD signal lines.
- Receiver interrupts are enabled.

When there is a call to synchronize time by an application process, the Kernel does several

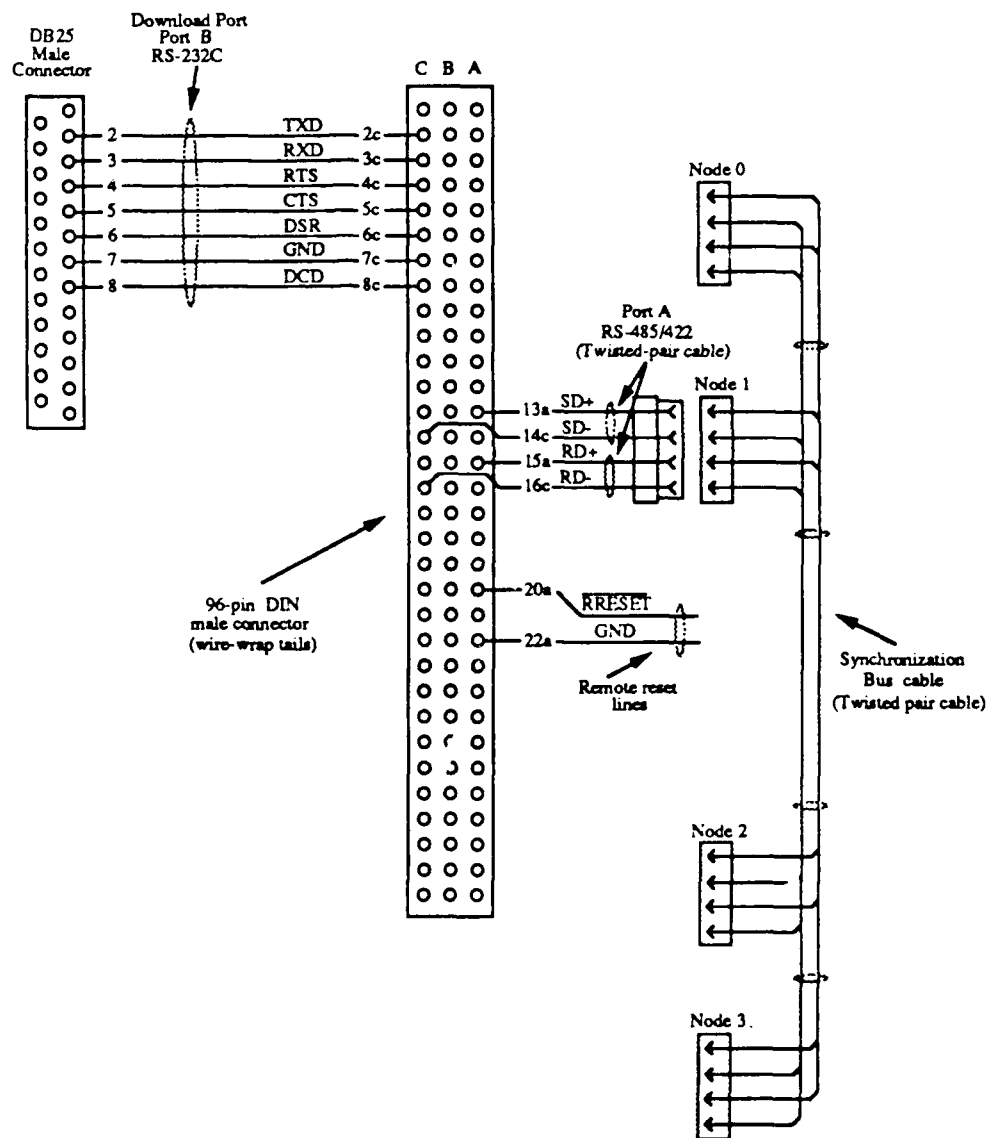


Figure 42: Kproc to Synchronization Bus Interface

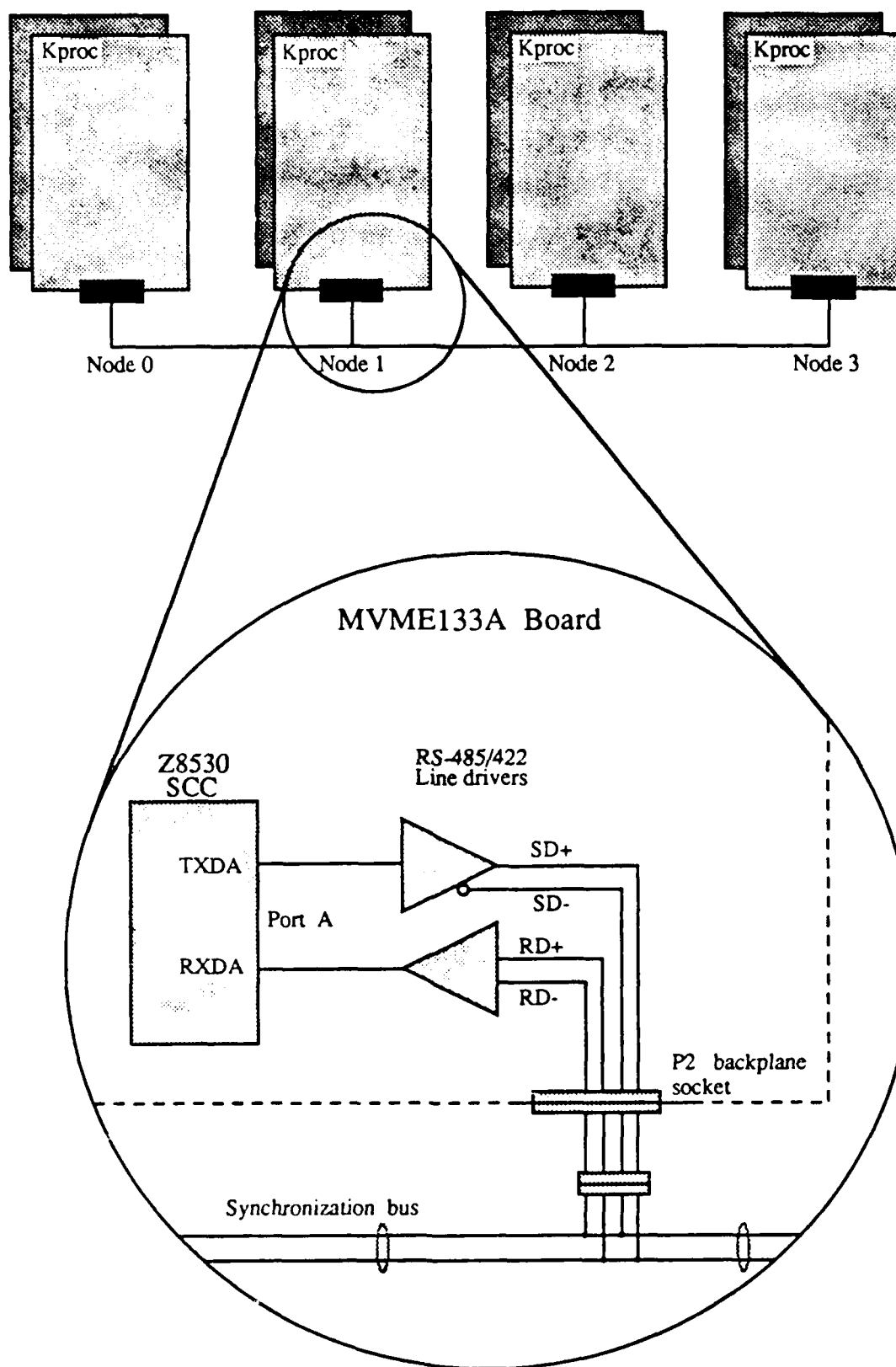


Figure 43: P2 Cable Harness Schematic

operations. One of the last is to assert port A mastership. This must be done before the time information can be broadcast to the other Kprocs. Port A mastership is attained by having:

- Transmitter output signal connected to the SD signal lines.
- Port A receiver interrupts disabled.
- Port A receiver input monitoring the RD signal lines.

A test byte is put onto the sync bus to verify that no other port is asserting itself as master. The receiver is checked to confirm that the byte received is the byte sent. In the meantime, all of the slave ports respond to the test byte by interrupting its associated Kproc, and then preparation begins to receive a new time to synchronize on. If the test byte is not received correctly, appropriate actions are taken.

13. P2 Backplane Connector Wiring

The schematic in Figure 43 is of the connector that is made to plug into all the processor's P2 sockets on the VME backplane of the chassis. This connector and cable harness provide access to several interface signals made available at P2 from the MVME133A board. The interfaces include:

- Port A RS-485/422 serial lines
- Port B RS-232C serial lines
- Remote Reset control lines

Port A is used for connecting to the synchronization bus; Port B is used for the download port and connects to the host system. The remote reset control lines provide a way to remotely perform a reset of the processor.

There are eight identical harnesses, and all of them are identical. The port B lines on the Nproc's are not connected to anything.

IX. TeleSoft Ada Compiler Dependencies

This part lists and explains those aspects of the DARK software artifacts that are dependent on the specific Ada compiler used. To assist with the porting and maintenance process, this part explains the ways in which the DARK software depends on the specific Ada development systems used.

1. Major Dependencies

The DARK software exhibits dependencies on the Ada development system in three major areas:

- Aspects of basic software architecture and design
- Representation and use of basic data types
- Encapsulation of hand-coded assembler

These will be discussed at successively greater levels of detail.

1.1. Software Architecture and Design

Some aspects of the DARK architecture were constructed, in part, to build on known strengths of the Ada system and avoid known weaknesses.

For example, the decision to represent execution-time errors by Ada exceptions was taken, in part, because the compiler handled guarded regions very efficiently. As another example, the DARK software avoids records with discriminants entirely because of demonstrated implementation inefficiencies.

These decisions require review if the software is to be moved to another Ada system. In most cases, we believe we have not made it impossibly difficult for the porter to modify, or even reverse, the most significant architectural decisions.

The DARK architecture is discussed in detail in Chapter 3.

1.2. Basic Data Types

The DARK software is largely embedded systems code, and as such must interface closely to the target machine in many places. It must be able to manipulate basic data types, memory addresses, device registers, and similar low-level target entities. As far as possible, it must do so robustly and clearly, through the medium of the Ada language.

The software therefore uses Ada representations of basic machine objects, defined as data types with necessary operations. These Ada definitions depend on the Ada compiler's own view of the machine, as captured in the package *System*. A different compiler, for a different or even the same machine, may take a different view, and in consequence the DARK definitions might need to be changed.

The required data types are given in detail in Chapter 4. The specific definitions used on each target are given in the appendices.

1.3. Encapsulation of Assembler

The DARK software contains some small modules written by hand in the target assembler code. For example, the DARK scheduler's basic context save and restore operations manipulate target machine state directly, and hence must be written in machine code.

In order for these modules to interface properly with the Ada code in which the rest of the Kernel is written, the compiler conventions must be relied upon in great detail. If a different compiler uses different conventions, the encapsulated assembler will have to be changed.

Note that this is a separate issue from the issue of recoding the modules for a new target. For example, another Ada compiler on the same target might define a different procedural interface, passing parameters in a different way. The called module would remain MC68020 Assembler, running on the same hardware, but would still have to be modified.

The conventions that must be understood and observed are given in detail in Chapter 5. The specific conventions used for each target are given in the appendices.

2. Software Architecture and Design Dependencies

This chapter gives the main compiler dependencies, explains the reasons behind them, and indicates what modifications might be feasible as a consequence of a port or compiler upgrade.

2.1. Code Customization

The most obvious and pervasive compiler-dependent architectural feature of DARK is the manner in which it allows customization of the code by acceptor sites and other users. The bulk of this customization involves the enabling and disabling of error checks.

The DARK model is the following:

- Error-checking code can be included in the Kernel, or excluded from it, at the option of the acceptor site.
- The inclusion of the code is controlled by a conditional statement, wherever possible at the outermost level of nesting within the relevant procedure.
- The condition driving the conditional statement is a generic formal parameter of the package, of a simple scalar type.
- Customization involves instantiating the package with a corresponding generic actual parameter that is an explicit constant.

The DARK model relies on these features of the Ada compiler:

- Generic instantiation is performed by code substitution at compile time.
- Constant actual values are substituted for the formal parameters, and simple constant comparisons will be done at compile time.
- Code guarded by conditional statements that are known at compile time to be false ("dead code") will be removed by the compiler and will not generate any object code. (This is not necessary for correct functioning of the Kernel, but makes it smaller and faster.)

These simple mechanisms are found in many Ada compilers, and they are accepted by the Ada community as a conventional way of achieving "conditional compilation" in Ada. However, should a compiler not support them, the customization cannot be achieved in this way. The brute-force alternative is to edit the Ada source code to remove any unwanted code; to help achieve this, all source code lines dealing with removable error checking are individually commented.

2.2. Representation of Errors

As discussed above, the DARK primitives can be customized to detect and report various execution-time errors. There are also some error conditions that cannot be suppressed, either because they are part of the semantics of the primitive or because continued execution in their presence would be impossible.

The Kernel systematically represents error conditions by user-defined exceptions, and reports them by raising the exception, presumably to be handled by the invoking code.

For this to be feasible, certain compiler features are assumed:

- The execution cost of guarded regions and unraised exceptions is very small, preferably zero.
- The cost of raising and propagating an exception is reasonable; in particular, it is not so great as to prevent timely recovery by the exception handler.
- The exception mechanism can function safely and accurately in the context of a DARK process.

The first two assumptions can be verified by studying the compiler documentation or (at worst) its output. The third assumption must be tested more carefully, with these issues in mind:

- Does the fact that a DARK process executes on its own stack affect exception semantics?
- Will the process "stack plug" effectively prevent exception propagation out of a process?
- Can exceptions propagate correctly out of, and through, any hand-coded assembler subprograms?
- Is the propagation mechanism re-entrant at the DARK process level, so that a process can be sliced during exception propagation?

If, for any reason, it is felt that the exception mechanism should not be used to indicate errors, then the Kernel code could be changed, albeit at some cost, to use status codes instead of exceptions. The information record for a DARK process contains a component that can be set to indicate an error condition. This component is used internally by the Kernel scheduler and context switch routines, but it would be straightforward to add an enquiry function that allowed a process to interrogate its own error status.

The one DARK facility that relies absolutely on the exception mechanism is the *alarm*; no other reliable means exists in Ada for aborting execution of linear code and transferring control to another part of the same process. It might be possible to revise the alarm semantics so that control is transferred to an Ada-labeled statement, but that is both poorer methodology and a less portable solution.

2.3. Module Initialization

The Kernel is structured into several modules in a dependency graph of several levels. Many of these modules contain data structures or device-handling code that requires initialization.

The current implementation tries as far as possible to use explicit initialization procedures called explicitly from the top-level modules. It restricts "automatic" module initialization, done by the statement sequence of the package body, to as few modules as possible. Also, it does not assume any specific order in which automatic initialization will be done.

This strategy should be robust against compiler differences in package elaboration order, and should work correctly without the acceptor site having to change module dependencies or introduce pragma *Elaborate*. However, the developers found some very subtle elaboration order problems, so they cannot assert the problem is completely solved.

However, there are some initialization dependencies between *processors* comprising a DARK target network. In particular, the communication mechanism must be initialized and ready before the rest of the Kernel can begin execution; since on the first target the communication is done by separate processors running separately linked programs, this dependency cannot be captured in the Ada code. In addition, most Kernel internal data structures, and some visible to the application, rely on the Ada facility that allows explicit or default initialization of declared objects.

Correct initialization of the Kernel must be checked as part of the test of the port.

2.4. Chapter 13 Issues

Those parts of the Kernel that manipulate the target machine rely to some extent on the features provided in Chapter 13 of [ALRM 83]. If any of them are absent, it will not be easy to find alternative strategies, since the implementors have used these features only when they believed them essential.

The specific features used, and their purposes, are:

- Size specifications [13.2(a)] to force the compiler to use operations of the correct size for a hardware device register. Failure to do this will cause a hardware memory reference trap.
- Record representation clauses [13.3] to construct Ada objects with the exact layout required by hardware devices. They are also used to specify the layout of objects transferred between processors. If this cannot be done, the device-handling code, and its associated data structures, must be rewritten in assembler.
- Address clauses [13.5] to place objects shared between processors in explicit places in shared memory. If this cannot be done, the same effect might be achievable via the linker. As a last resort, address binding can be done by using access objects set by hand to designate the correct memory addresses.

- The package *System* [13.7] for the basic machine types, as explained in the next chapter.
- The system-dependent named numbers [13.7.1], to obtain the extrema of the basic types.
- The attribute *'Address* [13.7.2], to compute the addresses of both subprograms and data objects. This is essential if the Kernel is to be ported, since, for example, it must be able to take the *'Address* of a subprogram that is to be a process, and the *'Address* of a data object whose contents is an inter-process message.
- The attribute *'Size* [13.7.2] of basic types and of arrays, to allow the Kernel to compute and allocate storage.
- The machine-code insertion facility [13.8], used very sparingly.
- The pragma *Interface*, to allow the Kernel to invoke machine-code subroutines. This is essential.
- The pragma *Inline*, for subprograms whose bodies are small enough. It is especially used for shell procedures that merely call other lower-level or more general procedures. This is not essential, but allows a good compromise between execution efficiency and functional abstraction. The Kernel assumes that, if the pragma is provided, it can be used to inline across compilation unit boundaries.
- The generic *Unchecked_Conversion* [13.10.2] for several low-level purposes, as explained in the next chapter. It is probably essential that this facility work, and work with high efficiency, on all simple types.

There is very little in the way of a contingency plan should some of the above language features be absent. In general, they provide facilities that cannot be obtained in any other way, and that are an essential part of much embedded systems programming.

2.5. Pragmas

The Kernel uses the following standard pragmas:

- pragma *INLINE*. This is not necessary; it is used to gain some extra efficiency by hoisting small routines. The structure of the code assumes that the pragma works across separate compilation boundaries. As a result of a restriction in the compiler used to develop the Kernel, this pragma is never applied to the result of instantiating a generic subprogram. One consequence is that instantiations of *unchecked_conversion* have almost always been hoisted by hand into package specifications.
- pragma *INTERFACE*. This is used to allow the Ada code to call lower level routines written in machine code, as explained in detail in Appendix D.

2.6. Ada Use Subset

As well as making certain assumptions about what the Ada system does provide, the Kernel was designed and written under certain assumptions about what it *need not* provide. In effect, it employs an *application subset* of the language, avoiding constructs that the implementation team believed either unnecessary or possibly inefficient.

These assumptions should not affect a port, since in general they have led to simpler and more straightforward code. They are recorded in full in a project-specific Ada style guide. However, the most significant unused features of Ada are given here:

- The Kernel makes no use of tasking.
- Records with discriminants are not used. This has no visible impact on the application, but has caused some slightly strange coding styles in parts of the Kernel.
- Objects of dynamic size are never declared within subprograms.
- Subprograms are not nested within other subprograms.
- Allocated storage is never deallocated, either explicitly or implicitly. All uses of the Ada allocator could be removed from the Kernel, if it seemed desirable for a port to use a custom storage-management system.
- The **separate** clause is not used. This is to avoid the name management problems that arise with library subunits.

3. Basic Data Types and Operations

This chapter gives the main hardware data types and operations required by DARK and explains how they have been constructed in Ada. It indicates the compiler dependencies involved in this process. These dependencies are encapsulated in the package *hardware_interface*, which is included in Appendix E.

Target-dependent values, that must be computed afresh for each machine, are indicated here by [].

3.1. Sizes of Data Types

The sizes of the basic machine data types are defined as manifest constants:

```
bits_per_byte : constant := [];  
  
byte          : constant := [];  
  
word          : constant := [];  
  
longword      : constant := [];
```

On a typical 32-bit target, these values will be 8, 1, 2, and 4, respectively.

3.2. Untyped Storage

Access to the basic machine storage units is provided by two data types, one defining the smallest addressable unit as an integer type, and the other defining it as a record composed of individual bits. Representation clauses are used to enforce the correct mapping from the Ada level to the hardware:

```
type hw_[unit] is range []..[];  
  
for hw_[unit]'size use [];  
  
type hw_bits[N] is record  
  bit[K] : Boolean;  
  -- repeat for all N bits, in the appropriate order  
end record;  
  
for hw_bits[N] use record  
  []  
end record;
```

On a typical byte-addressed machine, the hardware unit will be an unsigned 8-bit byte, range 0..255. The corresponding record type will then be:

```

type hw_bits8 is record
  bit0 : Boolean;
  bit1 : Boolean;
  bit2 : Boolean;
  bit3 : Boolean;
  bit4 : Boolean;
  bit5 : Boolean;
  bit6 : Boolean;
  bit7 : Boolean;
end record;

```

The value conversion is also defined from the integer type to the record type, using an instantiation of `unchecked_conversion`:

```

function to_hw_bits[N] is
  new unchecked_conversion(hw_[unit], hw_bits[N]);

```

If a variable `V` has been declared as a `hw_[unit]`, the individual bits of its current value can be accessed by the function call: `to_hw_bits[N](V)`.

It is also necessary to be able to convert variables from one type to the other; this is achieved by the usual Ada device of defining two access types and a value conversion between access values. This in effect allows a pointer to an object of one type to be converted into a pointer to an object of the other type:

```

type hw_[unit]_ptr is access hw_[unit];
type hw_bits[N]_ptr is access hw_bits[N];

function to_hw_[unit]_ptr is
  new unchecked_conversion(system.address, hw_[unit]_ptr);

function to_hw_bits[N]_ptr is
  new unchecked_conversion(system.address, hw_bits[N]_ptr);

function to_hw_bits[N]_ptr is
  new unchecked_conversion(hw_[unit]_ptr, hw_bits[N]_ptr);

```

Hence, a given bit of the variable `V` could be modified by:

```

to_hw_bits[N]_ptr(V'address).bitK := new_value_for_bitK;

```

3.3. Integer Types

The basic integer types are defined by giving explicit ranges. Representation clauses are used to enforce the correct mapping from the Ada level to the hardware:

```

type hw_integer is range [] .. [];
for hw_integer'size use [] * bits_per_byte;

type hw_short_integer is range [] .. [];
for hw_short_integer'size use [] * bits_per_byte;

type hw_long_integer is range [] .. [];
for hw_long_integer'size use [] * bits_per_byte;

```

The Kernel assumes that a `hw_integer` is at least 16 bits wide, and a `hw_long_integer` is at least 32 bits wide.

In addition, the important subsets of the basic integer types are defined explicitly. These correspond to the Ada *natural* subset—the non-negative integers within the range—and the *positive* subset—the strictly positive integers within the range:

```

type hw_natural is range 0 .. hw_integer'last;
for hw_natural'size use hw_integer'size;

type hw_positive is range 1 .. hw_integer'last;
for hw_positive'size use hw_integer'size;

type hw_long_natural is range 0 .. hw_long_integer'last;
for hw_long_natural'size use hw_long_integer'size;

type hw_long_positive is range 1 .. hw_long_integer'last;
for hw_long_positive'size use hw_long_integer'size;

```

3.4. Duration

The target representation of the Ada type *duration* is defined; this must capture exactly the representation used by the compiler:

```

type hw_duration is
  new duration range -86_400.0 .. +86_400.0;
for hw_duration'small use [];
for hw_duration'size use [] * bits_per_byte;

```

3.5. Machine Addresses

The Kernel must be able to generate the addresses of subprograms and objects, store them, pass them around, and subsequently use them. It is necessary, therefore, to define an appropriate address type, together with conversions from the Ada type *System.Address*:

```
type hw_address is [];  
  
function to_hw_address is  
  new unchecked_conversion(hw_long_integer, hw_address);  
  
null_hw_address : constant hw_address  
  := to_hw_address(hw_long_integer' ([]));
```

On a conventional von Neumann machine, the type *System.Address* will probably be an integer type, and type *hw_address* can simply be derived from it. The null value should if possible be a value that will cause a hardware trap if an attempt is made to use it as an address; typical null values are 0 and -1.

3.6. Strings

Finally, a suitable string type is defined:

```
type hw_string is [];
```

This will almost always be a type derived from the standard Ada string type.

4. Encapsulation of Assembly Code

This chapter explains the principles behind the design of DARK hand-coded assembler modules and their interface to the Ada code. It outlines the considerations that such a design must address, and the compiler dependencies involved.

4.1. Linkage

First, the compiler must provide a means of invoking assembler subprograms from Ada. This should be done by the standard Ada pragma *Interface*. However, it might also be necessary to use appropriate naming conventions for the subprograms, since the machine assembler and linker might not obey the Ada conventions concerning lexical identifiers. Particular issues to be addressed are:

- Legal characters
- Maximum allowed length
- Case sensitivity
- Possible clashes with compiler-generated names

It is also necessary to be able to inform the Ada library or binder that certain bodies are in assembler, so that it does not complain when they are not found in Ada. This is often done automatically as a consequence of the pragma. Finally, the real bodies must be linked with the Ada code to form the executable image; this may be done by a special command to the Ada linker, or by importing the bodies into the current library.

The assembler code must contain the appropriate cross-reference directives to make any defined symbols known to the Ada linker or debugger.

4.2. Program and Data Sections and Attributes

Any assembler code must be assigned to the correct code section or *Psect*, with the correct attributes. This is usually done by assembler directives. These must be inserted in the assembler code bodies in a manner prescribed by the machine assembler manual, and they must conform to the conventions used by the Ada compiler.

Conventions that the hand coder might be advised to respect include:

- Declare code sections to be execute-only
- Use position-independent code
- Avoid jumping between different subprogram bodies

Similarly, any data objects defined at the assembler level must be allocated in the appropriate data sections, with the correct attributes, just as if the Ada compiler had created them rather than the hand coder. Any conventions for allocating read-only and read-write

data should be respected; for example, if the compiler stores string literals in a read-only data area, so should the hand coder.

4.3. Data Representation

In the Kernel, some objects are accessed by both Ada code and machine code. It is necessary for the machine code to understand the representations of these objects. Objects passed from Ada code down to assembler code include simple integers, addresses, values of type duration, and simple records.

Most data representation issues are captured by the declarations in Chapter 3; for any of the types there introduced, the correct target representation is made explicit and guaranteed by appropriate representation clauses.

Any record types that must be used by the assembler level are defined in terms of the simple hardware types, and their representation also is fixed by representation clauses.

4.4. Access to Ada Objects from Assembly Code

The Kernel has been structured so that no assembler subprogram requires direct access to any Ada object. Where access to such objects is necessary, special subprograms have been introduced that allow the relevant addresses to be passed as parameters, or returned as results, from one level to the other.

For example, one assembler subprogram invokes the Kernel scheduler. This is done by calling an initialization routine that passes as a parameter the address of the scheduler subprogram; this address is saved in a static variable within the assembler module, and the call of the scheduler is performed indirectly through this variable.

4.5. Access to Assembler Objects from Ada Code

The Kernel has been structured so that no Ada code requires direct access to any data object created at the assembler level.

4.6. Procedural Interface

Any assembler subprograms must obey all appropriate parts of the Ada protocol for the procedural interface. This includes:

- Entry protocol
- Exit and return protocol
- Register usage

- Stack manipulation
- Parameter passing
- Exception propagation

The parameter-passing protocol of course depends on the number, type, and mode of the parameters, and so will be different for different subprograms. The Ada conventions for parameters and results must be copied exactly by the assembler code, for all appropriate types and modes.

The issues involved are:

- Mode of transmission—value or reference
- Manner of transmission
- Order of parameters
- How to access **in** parameters
- How to set **out** parameters
- How to return function results
- What extra "hidden" parameters need to be passed

The protocol for saving and restoring registers may depend on how many registers the assembler code subprogram uses. Some assemblers can generate this protocol mechanically, but most cannot, and the hand coder must then take care to save and restore all registers that ought to be saved and restored.

The rest of the protocol will usually consist of standard sequences that will be the same, or almost the same, for all subprograms.

4.7. Exceptions

The assembler subprograms must perform all necessary action to ensure that exceptions are correctly raised and propagated. This includes:

- Raising exceptions where appropriate
- Ensuring that exceptions are propagated out of assembler bodies
- Ensuring that exceptions are propagated *through* assembler bodies
- Informing the Ada runtime that a subprogram contains *no* guarded regions
- Identifying any guarded regions

The current Kernel requires only the first four of these facilities, since no assembler code contains a guarded region.

Appendix A: Data and Control Flow Diagrams

The notation used for data and control flow is a modified form of the notation expounded on by Paul Ward and Stephen Mellor in their books on the design of real-time software [Ward 85]. The notation used is true to the intent of Ward and Mellor's notation. The only variations are:

- use of rectangles with rounded corners for processes
- use of a square for external entities

Aside from these minor cosmetic changes, the data and control flow diagrams used follow the conventions set forth by Ward & Mellor. Figures 44 thru 46 briefly explain the symbols available using this notation.



Figure 44: Store Notation

The data store icon, shown in Figure 44 (a), represents a place where data is held until needed by a process.

The event store icon, shown in Figure 44 (b), represents a place where control signals are held until needed by a process.

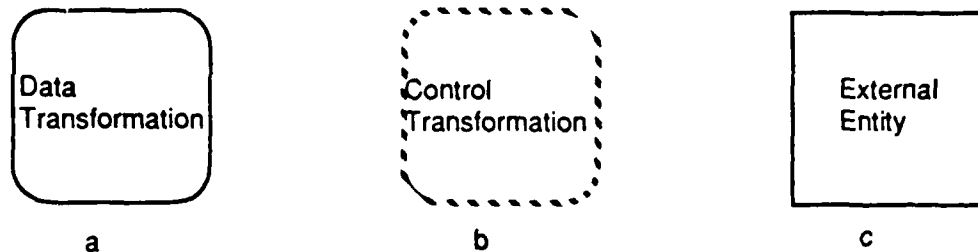


Figure 45: Process Notation

The data transformation icon, shown in Figure 45 (a), represents a process that accepts input data from a data flow(s), control signal(s) from an event flow(s), performs processing on the input data, and transfers the data out over a data flow(s).

The control transformation icon, shown in Figure 45 (b), represents a process that accepts a control signal(s) from an event flow(s), performs processing on the control signal and transfers information out over an event flow(s).

The external entity icon, shown in Figure 45 (c), represents a physical device capable of generating and/or accepting data and control flows.

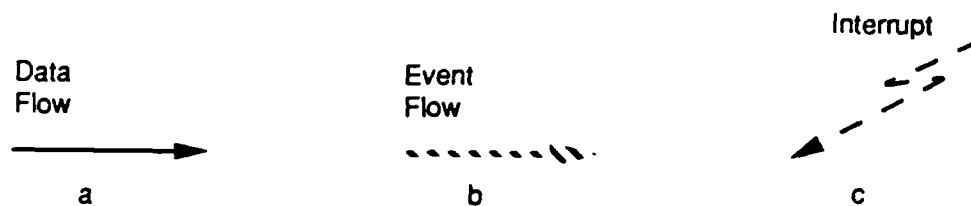


Figure 46: Flow Notation

The data flow symbol, shown in Figure 46 (a), represents the transfer of data from one process to another process or an external entity. This is a discrete transfer, i.e., the data is available until read and then is no longer available via the flow.

The event flow symbol, shown in Figure 46 (b), represents the transfer of a control signal from one process to another process or an external entity. This a discrete transfer, i.e., the signal is available until read and then is no longer available via the flow.

The interrupt symbol, shown in Figure 46 (c), represents transfer of a control from from one design entity to another.

Appendix B: Kernel Interface Control Document

This appendix presents the format of the various Kernel messages described elsewhere in this document. In particular, refer to Part IV, Communication Subsystem, Section 2.4 for the exact definition of each message field.

Operation	Tag	Sender	Receiver	Timeout	Length	Message ID	Message Content
Kernel message	ACK	ACKing process	originating process	NA	0	id of msg being ACKed	NA
Kernel message	NAK	NAKing process	originating process	NA	0	id of msg being NAKed	NA
Kernel message	NAK process dead	dead process	originating process	NA	0	id of msg being NAKed	NA
Kernel message	info process dead	dead process	originating process	NA	0	id of msg being NAKed	NA
Kernel message	kill process	originating process	process to kill	NA	0	NA	NA
Kernel message	init complete	Main Unit	Main Unit	NA	0	NA	NA
Kernel message	process created	Main Unit	Main Unit	NA	length of process name	NA	process name
Kernel message	network failure	any process	any process	NA	0	NA	NA

Table 1: Kernel Message Formats

Operation	Tag	Sender	Receiver	Timeout	Length	Message ID	Message Content
sync protocol	prepare to sync	originating processor	all other processors	NA	1	NA	16#FF#
sync protocol	time is now	originating processor	all other processors	NA	8	NA	current epoch time

Table 2: Synchronization Message Formats

Operation	Tag	Sender	Receiver	Timeout	Length	Message ID	Message Content
init protocol	master ready	Master processor	subordinate processor	NA	0	NA	NA
init protocol	NCT count	subordinate processor	Master processor	NA	4	NA	number of NCT entries
init protocol	NCT entry	subordinate processor	Master processor	NA	size of an NCT record	NA	an NCT record
init protocol	go enclosed	Master processor	subordinate processor	NA	8	NA	Current Epoch time
init protocol	go acknowledge	subordinate processor	Master processor	NA	0	NA	NA

Table 3: Initialization Message Formats

Operation	Tag	Sender	Receiver	Timeout	Length	Message ID	Message Content
blind send	defined by user	Main Unit	tool process	NA	See <i>tool interface package</i>	NA	See <i>tool interface package</i> , attribute definitions

Table 4: Tool Interface Message Formats

Appendix C: Race Conditions

In the Kernel, two types of asynchronous actions exist that can disturb the execution of a process:

1. Interrupts: where an external device takes control of the CPU away from the currently executing process.
2. Process suspension: where the Kernel takes control of the CPU away from a process (according to the rules defined for the Scheduler). This activity is either voluntary (via the invocation of a blocking primitive) or the result of an interrupt changing the state of a higher priority process.

Given that asynchronous activities can occur, there are a number of critical data structures that must be protected. These shared data structures are:

- Process Table (PTB)
 - Context Save Area
 - Schedule attributes
 - Message queue
 - Pending activities attributes
 - Semaphore attributes
 - Tool interface attributes
- Interrupt Table (IT)
- Network Configuration Table (NCT)
- Timeslice Parameters (TSP)

This appendix examines each of these data structures and identifies the potential race conditions and how each is resolved.

The assumptions upon which this analysis is based are:

1. The Kernel is reentrant, i.e., the Kernel, can suspend processes in the middle of primitive invocation processing.
2. The following procedures are atomic:
 - Time_keeper.insert_event
 - Time_keeper.remove_event
 - Scheduler.schedule
 - Context_switcher.switch_processes
 - Context_switcher.resume_process
 - Exception_raiser.raise_exception
 - Datagram_management.enqueue
 - Datagram_management.dequeue
 - Datagram_management.delete

- Datagram_management.get_head

C.1. Process Table - Context Save Area

Situation: Interrupt causing a context switch while the context switcher is executing.

Resolution: Make context switch atomic.

C.2. Process Table - Schedule Attributes

Situation: Alarm and timeout expire for the same process at same instant

- 2 different priorities
- 2 different exceptions

Resolution: Alarm expiration has precedence.

Situation: Executing *set_process_priority* or *set_process_preemption* when alarm expires (but before the call to *Schedule* that updates these values).

Resolution: Propagation of the alarm exception terminates the primitive invocation with the process having the priority specified by the *set_alarm* call.

Situation: Executing *set_process_priority* or *set_process_preemption* when a schedule operation occurs (but before the call to *Schedule* that updates these values).

Resolution: The operation simply takes place when next the process gets scheduled. The actual value updates still occur atomically.

C.3. Process Table - Message Queue

Situation: Executing *receive_message* when receive timeout expires

Resolution: Cancel timeout before beginning the message processing. This means that once a message becomes available and a *receive_message* is started, it will finish without a timeout interruption.

Situation: Updating PTB when an interrupt or process context switch occurs

Resolution: The only multi-component values that must be updated are the:

1. Scheduling attributes (priority, preemption, state)
 - on write, the update is handled via the Scheduler which is an atomic operation
 - on read, these are simple values and can safely be read without locking

2. Message Queue

- on write, all modifications are handled via the atomic operations enqueue, dequeue, get_head, and delete
- no simple reads are ever performed

Situation: Executing in an interrupt handler (i/h) when higher priority interrupt occurs

Resolution: Primitives and user code are reentrant

Situation: Executing *receive_message* when a "kill" message arrives for the receiving process

Resolution:

- Process is immediately yanked from the Scheduler (via *Remove_process*) and never completes the receive operation.
- The message queue is flushed, starting with the message that was being processed at the time "kill" arrived. This can be accomplished because *get_head* returns the message at the head of the queue without actually removing it from the queue. Removal from the queue is done when the buffer space is deallocated from the message queue and returned to the free list.

Situation: Executing *die* or *kill (self)* when a "kill" message arrives for the process

Resolution: Same as above.

Situation: Executing *send_message* or *send_message_and_wait* when "kill" message arrives for the sending process

Resolution:

- Process is immediately yanked from the Scheduler and never completes the send operation
- A CURRENT SEND BUFFER pointer is maintained in the process table. When a "kill" occurs, this buffer is also deallocated and returned to the free list.
- The incoming message queue is also purged (as above).
- The ACK/NAK to a dead process is ignored.

Situation: Executing *send_message* or *send_message_and_wait* when an alarm expires for the sending process

Resolution: The send operation is terminated. The alarm processing section of the clock interrupt handler *Deallocates* the CURRENT SEND BUFFER. No buffers are lost.

C.4. Process Table - Pending Activities Attributes

Situation: Executing `set_alarm` when an alarm expires (race on the ALARM RESUMPTION PRIORITY field)

Resolution: The alarm event is removed before modifications are made to the alarm-related data structures.

Situation: Executing `cancel_alarm` when an alarm expires

Resolution: Propagation of the alarm exception terminates the primitive invocation with the process having the priority specified by the `set_alarm` call (i.e., the alarm expiry has precedence).

Situation: Executing any primitive when an alarm expires

Resolution: Propagation of the alarm exception terminates the primitive invocation with the process having the priority specified by the `set_alarm` call.

NOTE:

1. When an alarm expires no post conditions are guaranteed for any Kernel operation executing at that instant.
2. When an alarm expires during a `receive_message`, the message being processed is still in the queue and is picked up by the next `receive_message` invocation.

C.5. Process Table - Semaphores Attributes

Situation: Executing `claim` when preempted by a higher priority process

Resolution: Manipulation of semaphore wait queue is atomic.

Situation: Executing `Release` when preempted by a higher priority process

Resolution: Manipulation of semaphore wait queue is atomic.

C.6. Table - Tool Interface Attributes

Situation: Executing `begin_collection` when a tool interface message arrives

Resolution: Simple value update (receiving process id). No locking needed.

Situation: Executing `end_collection` when a tool interface message arrives

Resolution: Simple value update (receiving process id). No locking needed.

C.7. Interrupt Table

Situation: Executing `bind_interrupt_handler` when an interrupt occurs (for the interrupt being rebound)

Resolution: The interrupt being rebound must be explicitly disabled first.

C.8. Network Configuration Table

Situation: Executing `allocate_device_receiver` when a message arrives from that device.

Resolution: Simple value update (receiving process id). No locking needed.

Situation: Message arrives from a non-Kernel device while the device receiver is going thru *die* or *kill* processing

Resolution: The process is marked as dead in the first step of this processing, thus the message from the non-Kernel device is thrown away without ever being queued.

C.9. Timeslice Parameters

Situation: Executing `set_timeslice_quantum` when a slice expires.

Resolution: The timeslice quantum is a simple value. No locking needed.

Situation: Executing `set_timeslice_quantum` when an interrupt occurs whose interrupt handler executes `set_timeslice_quantum`.

Resolution: These calls serialize (simple write), with the last one to execute dictating the new value.

Situation: Executing `disable_time_slicing` when a slice expires.

Resolution: Simple value update. No locking needed.

Situation: Executing `disable_time_slicing` when an interrupt occurs whose interrupt handler executes `disable_time_slicing`.

Resolution: These calls serialize, with the last one to execute dictating the new value.

Appendix D: 68020 Assembler Interface

This appendix gives the assembler interface used in the MC68020 with the current compiler and version. Full details of the compiler and version conventions are found in [TeleSoft 88], Chapter 6.

D.1. Linkage

Linkage is effected by the pragmas *Interface* and *Linkname* [TSUG 6.12.1]:

```
procedure Low_Level_Action;  
  
pragma Interface (Assembly, Low_Level_Action);  
pragma Linkname (Low_Level_Action, "PSN_low_level_action");
```

The linkname always begins with the package short name (PSN); these prefixes are unique and do not clash with any compiler-generated names. The linkname continues with the name of the subprogram; the total length never exceeds the maximum significant length of a linkname, so this convention also ensures there are no name clashes for non-overloaded subprograms.

When two or more Ada specifications* with the same expanded name – *overloaded* subprograms – are implemented in assembler, the exported Ada names are generated by **renames** declarations, and the true subprograms have names made unique by appending a suffix.

Linknames are not case sensitive.

Special rules had to be followed for operators, since the compiler does not permit the pragma INTERFACE to be applied to operator designators. The function was given a conventional name and then renamed as the operator:

```
function plus (left, right : T) return T;  
pragma Interface(Assembly, plus);  
...  
function "+" (left, right : T) return T renames plus;
```

Within the assembler body, the linkname is generated by a standard XDEF directive:

```
XDEF PSN_low_level_action
```

The assembler routines must be presented to the Ada library as implementations of package or subprogram bodies. This is done by the *//Import* function of the Ada system [TSUG 5.1], which must be invoked for each assembler unit after it has been assembled and before any program requiring it can be linked. A file may contain either Ada code or assembler code, but not both. DARK project naming conventions require that a file containing assembler code be named exactly as it would be if it were in Ada, but with the additional suffix *_machine_code*. An Ada specification is implemented, therefore, by at most one Ada body and one assembler code body.

D.2. Program and Data Sections

No use was made of program or data sections; the target configuration makes no distinction between code and data.

The code as written is position independent.

D.3. Data Representation

The data representations common to both Ada and assembler levels are as specified in Appendix E.

There were two difficulties with these definitions:

- The compiler does not allocate single bytes for single byte-sized objects. It allocates at least a (2-byte) word. However, it can allocate bytes for byte-sized record components, and this is all that the Kernel requires.
- The compiler numbers the bits in an object from left to right [TSUG 6.8]. This is contrary to the target machine conventions, which are observed by all the hardware documentation. The solution was to name the individual bits in a byte in accordance with the hardware convention, so *bit0* is the least significant bit, and enforce compiler compliance by a representation clause.

D.4. Access to Ada Objects from Assembly Code

Not required.

D.5. Access to Assembler Objects from Ada Code

Not required.

D.6. Procedural Interface

The procedural interface uses the following protocol [TSUG 6.10, 6.12]:

D.6.1. Entry and Exit Protocol

- Call is by a JSR instruction, and on entry to the subprogram, the hardware stack pointer (A7) points to the return address. Above this are the parameters, in left-to-right order, so the last parameter is closest to A7. The caller therefore must push the parameters onto the stack before the call.
- Exit is by an RTS, similarly; the caller resets the stack to reclaim the parameter space.

D.6.2. Register Usage

- The called routine must save and restore any registers it uses except D0 and D1.
- It must return a simple (see D.6.4.1) function result in D0.
- If it is returning a function result by reference, it must return the address of the result in D0.

D.6.3. Stack Manipulation

- The called routine *must* at all times maintain in A7 a valid hardware stack pointer.
- The called routine may claim local storage by lowering the stack pointer; it must restore the old value before exit.
- The called routine might¹² have to build an Ada stack frame. This is done by issuing a LINK instruction at the beginning and a corresponding UNLK instruction at the end.

D.6.4. Parameter Passing

The parameter passing conventions are as follows [TSUG 6.10.2]:

D.6.4.1. Mode of Transmission

- All parameters smaller than a (2-byte) word are widened to 2 bytes and passed by value. With the current compiler, all such parameters must be of a scalar type.
- Parameters of simple (scalar and access) types and 4 bytes wide or smaller are passed by value.
- Parameters larger than 4 bytes, and *all* parameters of structured types, are passed by simple reference. This reference is the machine address of the lowest-numbered storage unit, and is a 4-byte value.

D.6.4.2. Manner and Order of Transmission

- Parameters are pushed on the hardware (A7) stack in *direct* order. That is, the leftmost parameter is pushed on the stack first, and the rightmost last.
- A parameter passed by reference passes the address of the actual as an In parameter.

D.6.4.3. Accessing Parameters and Returning Function Results

- Value In parameters pass the actual value. Value out parameters pass in binary zero and expect the out value to overwrite it. Value In out parameters pass in the actual value and expect the new value to overwrite it.
- Function results not larger than 4 bytes are returned in register D0.

¹²Only needed if an exception may have to propagate through the assembly language routine.

- Function results larger than 4 bytes are returned in a hidden **out** parameter whose address is passed as the *final* parameter to the function, after the last true parameter. In addition, this address must also be returned in D0.
- Although some Ada types require hidden parameters to be passed along with their actual values, no such types are used by any assembler subprogram.

Note that, since parameters are pushed on the stack left-to-right, the offset from the stack pointer of any given parameter (the first, say) depends on the number of parameters and their types. It is essential, therefore, that the Ada and assembler sides of this interface correspond exactly, otherwise serious execution-time errors result.

D.6.5. Example

The following example is taken from one of the Kernel modules. It is an implementation of the function plus:

```
function "+" (Left, Right : Kernel_Time) return Kernel_Time;
```

where the type *Kernel_time* is defined as a record with two components, each a 4-byte integer. The function is specified in the module *generic_kernel_time*, and so its assembler name begins with the package short name GKT.

```
HIGH EQU 4
LOW EQU 0

XDEF GKT_add [ 1]
GKT_add:
  LINK A6, #0 [ 2]
  MOVEM.L D2/A0, - (A7) [ 3]
  MOVEA.L 24 (A7), A0 [ 4]
  MOVE.L LOW(A0), D0
  MOVE.L HIGH(A0), D1
  MOVEA.L 20 (A7), A0 [ 5]
  ADD.L LOW(A0), D0
  MOVE.L HIGH(A0), D2
  ADDX.L D2, D1 [ 6]
  TRAPV [ 7]
  MOVEA.L 16 (A7), A0 [ 8]
  MOVE.L D0, LOW(A0)
  MOVE.L D1, HIGH(A0)
  MOVE.L A0, D0 [ 9]
  MOVEM.L (A7)+, D2/A0 [10]
  UNLK A6 [11]
  RTS [12]
```

The relevant instructions are annotated thus:

- [1]:The linkage directive makes the name of the function accessible.
- [2]:An Ada stack frame is built by a LINK instruction.
- [3]:Registers used are saved (except for D0 and D1).
- [4]:The first parameter is passed by address and is pushed first onto the stack. It is therefore farthest away from A7.

- [5]:The second parameter is likewise passed by reference and pushed next.
- [6]:The two *Kernel_time* values are added. This instruction may cause a numeric overflow.
- [7]:Accordingly, the TRAPV is necessary to detect any overflow and trap to the Ada runtime, which will raise the exception *numeric_error*.
- [8]:The address of the result is passed as a hidden final parameter.
- [9]:As well as storing the result there, the routine must return the same address in D0.
- [10]:The saved registers are restored.
- [11]:The Ada stack frame is unlinked.
- [12]:Finally, the function returns to the caller.

D.7. Exceptions

D.7.1. Raising Exceptions

The assembler code never raises a user-defined exception. Where appropriate, it raises an intrinsic exception by executing a hardware TRAP instruction, which traps to the Ada runtime.

D.7.2. Exception Propagation

The Ada runtime propagates exceptions upwards through stack frames, using the saved information in each stack frame to find each caller.

For this to work, every assembler body that can raise an exception, and every body *through which an exception might propagate*, must build a valid Ada stack frame, in the manner described above. The assembler body given in the example is a case in point: the exception *numeric_error* might have to be propagated through it, and so it must build a valid Ada stack frame.

In addition, the Ada runtime constructs a *backtrace* of the call stack for diagnostic purposes. The current Ada runtime tries to generate a complete backtrace before searching for an exception handler, but unfortunately the test for the root of the call graph — the Ada outermost level — does not function correctly when the root is a Kernel process. The backtrace has therefore been disabled by setting an Ada runtime tailoring parameter.

D.7.3. Guarded Regions

The model used by the compiler relies on static data structures — *code maps* — to identify guarded regions. The assumption made by the Ada runtime is that if the code map is missing then no guarded regions exist in that code. Accordingly, the assembler bodies contain no special code or data concerned with either guarded regions or their absence.

Appendix E: 68020 Tailoring

This appendix contains the hardware tailoring for the MC68020 with the current compiler and version.

E.1. Sizes of Data Types

```
bits_per_byte : constant := 8;

byte          : constant := 1;

word          : constant := 2;

longword      : constant := 4;
```

E.2. Untyped Storage

```
type hw_byte is range 0..255;

type hw_bits8 is record
    bit7 : Boolean;
    bit6 : Boolean;
    bit5 : Boolean;
    bit4 : Boolean;
    bit3 : Boolean;
    bit2 : Boolean;
    bit1 : Boolean;
    bit0 : Boolean;
end record;

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;

function to_hw_bits8 is
    new unchecked_conversion(hw_byte, hw_bits8);

type hw_byte_ptr is access hw_byte;
type hw_bits8_ptr is access hw_bits8;

function to_hw_byte_ptr
```

```

    is new unchecked_conversion(system.address, hw_byte_ptr);

function to_hw_bits8_ptr
    is new unchecked_conversion(system.address, hw_bits8_ptr);

function to_hw_bits8_ptr
    is new unchecked_conversion(hw_byte_ptr, hw_bits8_ptr);

```

E.3. Integer Types

```

type hw_integer is range -32_768 .. 32_767;
for hw_integer'size use 2 * bits_per_byte;

type hw_short_integer is range -128 .. 127;
for hw_short_integer'size use 1 * bits_per_byte;

type hw_long_integer is
    range -2_147_483_648 .. 2_147_483_647;
for hw_long_integer'size use 4 * bits_per_byte;

type hw_natural is range 0 .. 32_767;
for hw_natural'size use 2 * bits_per_byte;

type hw_positive is range 1 .. 32_767;
for hw_positive'size use 2 * bits_per_byte;

type hw_long_natural is range 0 .. 2_147_483_647;
for hw_long_natural'size use 4 * bits_per_byte;

type hw_long_positive is range 1 .. 2_147_483_647;
for hw_long_positive'size use 4 * bits_per_byte;

```

E.4. Duration

```

type hw_duration is
    new duration range -86_400.0 .. +86_400.0;
for hw_duration'small use 2.0 ** (-14);
for hw_duration'size use 4 * bits_per_byte;

```

E.5. Machine Addresses

```
type hw_address is new system.address;

function to_hw_address is
  new unchecked_conversion(hw_long_integer, hw_address);

null_hw_address : constant hw_address
  := to_hw_address(hw_long_integer'(0));
```

E.6. Strings

```
type hw_string is new string;
```

Appendix F: Procedure to Requirement Mapping

A few notes about the requirements mapping tables:

- Only procedures and functions that appear at the package specification level occur in the table. No instantiations are included (only the generic source).
- Top-level package specifications occur only when there are internal objects that fulfill requirements.
- Overloaded names appear multiple times.
- The ordering of procedures and entries follows the order of the entries in the specification.

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
5.1.1	kernel_exceptions exception_raiser.raise_exception exception_raiser
5.1.2	all Kernel primitives

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
5.1.3	time_keeper.remove_event time_keeper.insert_event time_keeper.initialize timer_controller.cancel_timer timer_controller.set_timer timer_controller.initialize scc_porta.set scc_porta.send scc_porta.put_byte scc_porta.initialize scc_porta.get_byte scc_porta.get scc_porta.enable_tx_interrupts scc_porta.enable_rx_interrupts scc_porta.disable_tx_interrupts scc_porta.disable_rx_interrupts scc_porta.deallocate scc_porta.allocate parallel_io_controller.receive_packet parallel_io_controller.send_packet parallel_io_controller.acknowledge_xmit_interrupt parallel_io_controller.acknowledge_rcv_interrupt parallel_io_controller.xmit_buffer_empty parallel_io_controller.rcv_buffer_full parallel_io_controller.disable_xmit_interrupt; parallel_io_controller.disable_rcv_interrupt; parallel_io_controller.enable_xmit_interrupt; parallel_io_controller.enable_rcv_interrupt; parallel_io_controller.initialize_xmit; parallel_io_controller.initialize_rcv; nproc.main_unit mz8305_definitions mvme133A_definitions memory_addresses low_level_interrupt_management.initialize low_level_interrupt_management.bind_slow_interrupt low_level_interrupt_management.bind_fast_interrupt low_level_hardware.reset_interrupt_priority low_level_hardware.set_interrupt_priority low_level_hardware.v low_level_hardware.p low_level_hardware.is_nproc low_level_hardware.is_kproc low_level_hardware.my_network_address low_level_hardware generic_network_globals context_switcher_globals clock.adjust_epoch_time clock.adjust_elapsed_time clock.get_time clock.stop_clock clock.start_clock bus_io.initialize

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
5.1.4	time_keeper_globals time_keeper.remove_event time_keeper.insert_event time_keeper.initialize timer_controller.cancel_timer timer_controller.set_timer timer_controller.initialize tc_body_machine_code scheduler.initialize scc_porta.set scc_porta.send scc_porta.put_byte scc_porta.initialize scc_porta.get_byte scc_porta.get scc_porta.enable_tx_interrupts scc_porta.enable_rx_interrupts scc_porta.disable_tx_interrupts scc_porta.disable_rx_interrupts scc_porta.deallocate scc_porta.allocate parallel_io_controller.receive_packet parallel_io_controller.send_packet parallel_io_controller.acknowledge_xmit_interrupt parallel_io_controller.acknowledge_rcv_interrupt parallel_io_controller.xmit_buffer_empty parallel_io_controller.rcv_buffer_full parallel_io_controller.disable_xmit_interrupt; parallel_io_controller.disable_rcv_interrupt; parallel_io_controller.enable_xmit_interrupt; parallel_io_controller.enable_rcv_interrupt; parallel_io_controller.initialize_xmit; parallel_io_controller.initialize_rcv; nproc.main_unit mz8305_definitions mvme133A_definitions memory_addresses low_level_storage_manager.allocate low_level_interrupt_management.initialize low_level_interrupt_management.bind_slow_interrupt low_level_interrupt_management.bind_fast_interrupt low_level_hardware.reset_interrupt_priority low_level_hardware.set_interrupt_priority low_level_hardware.v low_level_hardware.p low_level_hardware.is_nproc low_level_hardware.is_kproc low_level_hardware.my_network_address low_level_hardware low_level_clock

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
5.1.4 (continued)	lpe_body.Initialize_Process_State lim_body_machine_code lh_body_machine_code lcs_body_machine_code kim_body_machine_code kernel_exceptions.to_string ipi_body_machine_code interprocessor_interrupts.generate_kn_interrupt interprocessor_interrupts.enable_kn_interrupt interprocessor_interrupts internal_process_management.create_kernel_processes internal_process_management.create_internal_process internal_process_management.get_null_process_number internal_process_management.get_next_process_number hardware_interface.to_hw_bits8_ptr hardware_interface.to_hw_bits8_ptr hardware_interface.to_hw_byte_ptr hardware_interface.to_hw_bits8 hardware_interface gkt_body_machine_code generic_storage_manager.deallocate generic_storage_manager.allocate generic_storage_manager.initialize generic_queue_manager.remove_next generic_queue_manager.get_next generic_queue_manager.end_of_queue generic_queue_manager.initialize_backward_iterat generic_queue_manager.initialize_iterator generic_queue_manager.empty generic_queue_manager.get_element generic_queue_manager.get_head generic_queue_manager.remove generic_queue_manager.dequeue generic_queue_manager.enqueue generic_queue_manager.enqueue generic_queue_manager.delete generic_queue_manager.delete generic_queue_manager.create generic_process_table.size_of_process_table generic_process_table.locate_process generic_process_table.initialize_process_table generic_process_table.destroy_process_informatio generic_process_table.create_process_information generic_process_table."<=" " generic_process_table."<=" " generic_network_globals generic_communication_globals datagram_management.free_queue_status datagram_management.nproc_initialize datagram_management.kproc_initialize

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
5.1.4 (continued)	datagram_management.new_queue datagram_management.get_first datagram_management.free_dg datagram_management.enqueue datagram_management.dequeue datagram_management.delete datagram_management.alloc_dg datagram_globals dark_text_io.to_hex dark_text_io context_switcher_globals context_save_area clock.adjust_epoch_time clock.adjust_elapsed_time clock.get_time clock.stop_clock clock.start_clock bus_io.initialize
5.1.5	
5.1.6	network_configuration generic_schedule_types generic_process_table generic_process_managers generic_network_configuration
5.1.7	tool_interface_output time_keeper.remove_event time_keeper.insert_event time_keeper.initialize timer_controller.cancel_timer timer_controller.set_timer timer_controller.initialize generic_communication_globals clock.adjust_epoch_time clock.adjust_elapsed_time clock.get_time clock.stop_clock clock.start_clock
5.1.8	generic_time_management generic_timeslice_management generic_semaphore_management generic_schedule_types generic_process_managers generic_process_attribute_readers generic_process_attribute_modifiers generic_processor_management generic_network_globals generic_alarm_management

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
5.2.4	scheduler.schedule_ih scheduler.schedule scheduler.remove_process scheduler.insert_process scc_porta.set scc_porta.send scc_porta.put_byte scc_porta.initialize scc_porta.get_byte scc_porta.get scc_porta.enable_tx_interrupts scc_porta.enable_rx_interrupts scc_porta.disable_tx_interrupts scc_porta.disable_rx_interrupts scc_porta.deallocate scc_porta.allocate process_index_table.get_process_index process_index_table.get_process_identifier internal_process_management.get_null_process_number internal_process_management.get_next_process_number generic_storage_manager.deallocate generic_storage_manager.allocate generic_queue_manager.remove_next generic_queue_manager.get_next generic_queue_manager.end_of_queue generic_queue_manager.initialize_backward_iterat generic_queue_manager.initialize_iterator generic_queue_manager.empty generic_queue_manager.get_element generic_queue_manager.get_head generic_queue_manager.remove generic_queue_manager.dequeue generic_queue_manager.enqueue generic_queue_manager.enqueue generic_queue_manager.delete generic_queue_manager.delete

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
6.1.1	generic_processor_management.initialize_master_processor
6.1.2	generic_processor_management.initialize_subordinate_processor
6.1.3	generic_processor_management.initialize_subordinate_processor generic_processor_management.initialize_master_processor
6.1.4	generic_processor_management.initialize_subordinate_processor generic_processor_management.initialize_master_processor bus_io.multi_send
6.1.5	generic_processor_management.initialize_master_processor
6.1.6	generic_time_management generic_processor_management generic_network_configuration
6.1.7	generic_processor_management
6.1.8	generic_processor_management
6.1.9	generic_process_managers generic_process_attribute_modifiers generic_communication_management bus_io.multi_send
6.1.10	network_configuration generic_network_configuration

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
6.1.11	process_index_table.clear_process_index process_index_table.set_process_index process_index_table.get_process_index process_index_table.get_process_identifier parallel_io_controller.receive_packet parallel_io_controller.send_packet parallel_io_controller.acknowledge_xmit_interrupt parallel_io_controller.acknowledge_rcv_interrupt parallel_io_controller.xmit_buffer_empty parallel_io_controller.rcv_buffer_full parallel_io_controller.disable_xmit_interrupt; parallel_io_controller.disable_rcv_interrupt; parallel_io_controller.enable_xmit_interrupt; parallel_io_controller.enable_rcv_interrupt; parallel_io_controller.initialize_xmit; parallel_io_controller.initialize_rcv; nproc.main_unit network_configuration.get_processor_id network_configuration mz8305_definitions mvme133A_definitions memory_addresses low_level_hardware.reset_interrupt_priority low_level_hardware.set_interrupt_priority low_level_hardware.v low_level_hardware.p low_level_hardware.is_nproc low_level_hardware.is_kproc low_level_hardware.my_network_address low_level_hardware lih_body_machine_code ipi_body_machine_code interprocessor_interrupts.generate_kn_interrupt interprocessor_interrupts.enable_kn_interrupt interprocessor_interrupts internal_process_management.create_kernel_processes internal_process_management.create_internal_process generic_processor_management generic_network_globals generic_network_configuration.get_processor_id generic_network_configuration generic_communication_management datagram_management.free_queue_status datagram_management.nproc_initialize datagram_management.kproc_initialize datagram_management.new_queue datagram_management.get_first datagram_management.free_dg datagram_management.enqueue datagram_management.dequeue datagram_management.delete datagram_management.alloc_dg datagram_globals

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
6.2.1	time_keeper.initialize timer_controller.initialize scc_porta.initialize process_index_table.clear_process_index process_index_table.set_process_index network_configuration.get_processor_id network_configuration internal_process_management.create_kernel_processes internal_process_management.create_internal_process generic_storage_manager.initialize generic_queue_manager.create generic_process_managers.create_process generic_process_managers.declare_process generic_process_managers.declare_process generic_processor_management.initialization_complete generic_processor_management.initialize_subordinate_processor generic_processor_management.initialize_master_processor generic_network_configuration.get_processor_id generic_network_configuration clock.start_clock bus_io.initialize
6.2.2	generic_processor_management.initialize_master_processor

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
7.1.1	generic_process_managers generic_process_attribute_modifiers
7.1.2	generic_process_managers generic_process_attribute_modifiers
7.1.3	generic_process_managers.declare_process generic_process_managers.declare_process
7.1.4	process_encapsulation.dummy_call_frame low_level_process_encapsulation.indirect_call low_level_process_encapsulation.initialize_process_state low_level_context_switcher.hw_switch_processes low_level_context_switcher.hw_restore_process low_level_context_switcher.hw_save_context generic_process_table generic_process_managers.create_process context_switcher.switch_processes context_switcher.restore_process context_switcher.save_context
7.1.5	low_level_process_encapsulation.indirect_call generic_process_managers.create_process
7.1.6	generic_process_table generic_process_managers.create_process
7.1.7	generic_process_table generic_process_managers.create_process
7.1.8	generic_process_managers.create_process generic_process_managers
7.1.9	generic_process_table generic_process_managers.create_process
7.1.10	generic_process_managers.create_process
7.1.11	requirement deleted
7.1.12	process_encapsulation generic_process_managers.create_process
7.1.13	generic_process_managers generic_process_attribute_modifiers
7.1.14	network_configuration generic_network_configuration generic_communication_management.allocate_device_receiver
7.1.15	generic_processor_management.initialization_complete
7.1.16	network_configuration generic_processor_management.initialization_complete generic_network_configuration
7.1.17	generic_processor_management.initialization_complete

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
7.1.18	generic_process_attribute_modifiers.kill generic_process_attribute_modifiers.die
7.1.19	requirement deleted
7.1.20	generic_process_attribute_modifiers.kill
7.1.21	generic_process_table generic_process_attribute_modifiers.die generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message bus_io.send_kernel_datagram bus_io.send_process_datagram
7.1.22	requirement deleted
7.1.23	generic_process_attribute_readers.who_am_i
7.1.24	generic_process_attribute_readers.name_of
7.1.25	generic_process_managers.create_process generic_process_attribute_readers.name_of generic_process_attribute_modifiers.kill generic_process_attribute_modifiers.die generic_processor_management.initialization_complete generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_kernel_datagram bus_io.send_process_datagram
7.1.26	generic_process_managers.create_process generic_process_attribute_readers.name_of generic_process_attribute_modifiers.kill generic_process_attribute_modifiers.die generic_processor_management.initialization_complete generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_kernel_datagram bus_io.send_process_datagram
7.1.27	generic_process_table generic_process_managers.create_process
7.2.1	generic_process_managers.create_process
7.2.2	generic_process_attribute_modifiers.kill generic_process_attribute_modifiers.die
7.2.3	requirement deleted
7.2.4	generic_communication_management.allocate_device_receiver
7.2.5	generic_process_managers.create_process

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
7.2.6	generic_process_managers.create_process

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
8.1.1	generic_semaphore_management generic_process_table
8.1.2	generic_semaphore_management generic_process_table
8.1.3	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim generic_process_table
8.1.4	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim generic_process_table
8.1.5	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_semaphore_management.claim generic_process_table
8.1.6	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_semaphore_management.claim generic_process_table
8.1.7	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim
8.1.8	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim
8.1.9	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim generic_process_table
8.1.10	generic_semaphore_management.release generic_process_table
8.1.11	generic_semaphore_management.release generic_process_table
8.1.12	time_keeper generic_process_attribute_modifiers.kill generic_process_attribute_modifiers.die
8.1.13	generic_semaphore_management.release
8.1.14	generic_semaphore_management

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
8.1.15	generic_semaphore_management.release
8.1.16	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim generic_process_table
8.2.1	generic_semaphore_management generic_process_table
8.2.2	generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim
8.2.3	generic_semaphore_management.release

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
9.1.1	generic_process_managers.create_process
9.1.2	generic_schedule_types generic_process_managers.create_process
9.1.3	generic_timeslice_management generic_schedule_types
9.1.4	generic_timeslice_management generic_schedule_types
9.1.5	scheduler.insert_process generic_timeslice_management generic_schedule_types
9.1.6	scheduler.schedule generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.set_process_priority generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_alarm_management.set_alarm
9.1.7	generic_process_table generic_process_managers.create_process
9.1.8	scheduler.schedule generic_semaphore_management.claim generic_semaphore_management.claim generic_semaphore_management.claim generic_process_table generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.set_process_priority generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_alarm_management.set_alarm
9.1.9	generic_process_table generic_process_attribute_readers.get_process_priority
9.1.10	generic_timeslice_management generic_schedule_types

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
9.1.11	generic_process_table generic_process_managers.create_process
9.1.12	scheduler.schedule generic_process_table generic_process_attribute_modifiers.set_process_preemption
9.1.13	generic_process_table generic_process_attribute_readers.get_process_preemption
9.1.14	generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait
9.1.15	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_process_attribute_modifiers.wait
9.1.16	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_process_attribute_modifiers.wait
9.1.17	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait
9.1.18	timeslice_parameters generic_timeslice_management.set_timeslice
9.1.19	time_keeper.remove_event time_keeper.insert_event timeslice_parameters timer_controller.cancel_timer timer_controller.set_timer generic_timeslice_management.enable_time_slicing generic_process_table
9.1.20	time_keeper.remove_event time_keeper.insert_event timeslice_parameters timer_controller.cancel_timer timer_controller.set_timer generic_timeslice_management.disable_time_slicing generic_process_table
9.1.21	timeslice_parameters generic_timeslice_management

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
9.1.22	generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.set_process_priority generic_process_attribute_modifiers.set_process_preemption generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_alarm_management.cancel_alarm generic_alarm_management.set_alarm
9.1.23	scheduler.schedule_ih scheduler.schedule scheduler.remove_process scheduler.insert_process generic_schedule_types generic_process_table
9.1.24	scheduler.schedule_ih scheduler.schedule generic_timeslice_management
9.1.25	scheduler.schedule_ih scheduler.schedule
9.1.26	scheduler generic_timeslice_management
9.1.27	scheduler.schedule
9.1.28	scheduler
9.1.29	generic_process_table generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait
9.2.1	generic_process_attribute_modifiers.set_process_priority
9.2.2	generic_process_attribute_modifiers.set_process_preemption
9.2.3	generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait context_switcher.save_context
9.2.4	scheduler.schedule_ih scheduler.schedule generic_timeslice_management context_switcher.restore_process
9.2.5	generic_timeslice_management.disable_time_slicing generic_timeslice_management.enable_time_slicing

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
9.2.6	scheduler.schedule_in scheduler.schedule generic_timeslice_management context_switcher.switch_processes
9.2.7	scheduler.schedule_in scheduler.schedule generic_timeslice_management

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
10.1.1	internal_process_management.get_null_process_number internal_process_management.get_next_process_number generic_process_managers.declare_process generic_process_managers.declare_process generic_process_attribute_readers.name_of
10.1.2	generic_process_table generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message
10.1.3	generic_process_table generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
10.1.4	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
10.1.5	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message
10.1.6	process_index_table.clear_process_index process_index_table.set_process_index process_index_table.get_process_index process_index_table.get_process_identifier network_configuration.get_processor_id network_configuration generic_process_managers.declare_process generic_process_managers.declare_process generic_network_globals generic_network_configuration.get_processor_id generic_network_configuration generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message bus_io.send_kernel_datagram bus_io.send_process_datagram
10.1.7	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_communication_management.send_message_and_wait

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
10.1.8	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_communication_management.send_message_and_wait
10.1.9	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
10.1.10	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
10.1.11	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
10.1.12	generic_process_table generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.13	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.14	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.15	process_index_table.clear_process_index process_index_table.set_process_index process_index_table.get_process_index process_index_table.get_process_identifier network_configuration.get_processor_id network_configuration generic_process_managers.declare_process generic_process_managers.declare_process generic_network_globals generic_network_configuration.get_processor_id generic_network_configuration generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.16	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_communication_management.receive_message

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
10.1.17	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_communication_management.receive_message
10.1.18	generic_communication_management.receive_message generic_communication_management.receive_message
10.1.19	generic_communication_management
10.1.20	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.21	generic_communication_management.receive_message generic_communication_management.receive_message datagram_management.new_queue datagram_management.get_first datagram_management.dequeue datagram_management.delete
10.1.22	generic_process_table generic_process_managers_globals generic_process_managers.create_process generic_communication_management
10.1.23	generic_process_managers.create_process generic_communication_management
10.1.24	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.25	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message
10.1.26	network_configuration generic_process_managers.declare_process generic_network_configuration generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message
10.1.27	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
10.1.28	generic_communication_management
10.1.29	generic_communication_management bus_io.send_kernel_datagram bus_io.send_process_datagram

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
10.1.30	generic_communication_management
10.1.31	generic_communication_management
10.1.32	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_process_datagram
10.1.33	generic_communication_management bus_io
10.1.34	generic_communication_management bus_io
10.1.35	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message
10.1.36	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message bus_io
10.1.37	generic_process_table
10.1.38	generic_process_table
10.1.39	generic_process_table
10.2.1	generic_communication_management.send_message bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.2	generic_communication_management.send_message bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.3	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.4	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.5	generic_communication_management.send_message bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.6	generic_communication_management.send_message bus_io.send_kernel_datagram bus_io.send_process_datagram

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
10.2.7	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.8	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.9	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.10	generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message bus_io.send_kernel_datagram bus_io.send_process_datagram
10.2.11	generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
11.1.1	interrupt_names generic_interrupt_globals
11.1.2	kernel_interrupt_management.bind_interrupt_handler kernel_encapsulation.in_interrupt generic_interrupt_management.bind_interrupt_handler generic_interrupt_management.simulate_interrupt
11.1.3	generic_interrupt_management.simulate_interrupt
11.1.4	kernel_interrupt_management.enable generic_interrupt_management.enable generic_interrupt_globals
11.1.5	kernel_interrupt_management.disable generic_interrupt_management.disable generic_interrupt_globals
11.1.6	interprocessor_interrupts generic_interrupt_globals
11.1.7	kernel_interrupt_management.enabled generic_interrupt_management.enabled
11.1.8	kernel_interrupt_management.simulate_interrupt generic_interrupt_management.simulate_interrupt
11.1.9	kernel_interrupt_management.bind_interrupt_handler generic_interrupt_management.bind_interrupt_handler
11.1.10	kernel_interrupt_management.bind_interrupt_handler generic_interrupt_management.bind_interrupt_handler
11.1.11	low_level_interrupt_management.bind_slow_interrupt low_level_interrupt_management.bind_fast_interrupt kernel_interrupt_management.bind_interrupt_handler generic_interrupt_management.bind_interrupt_handler
11.1.12	generic_process_attribute_modifiers.wait generic_process_attribute_modifiers.wait generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait
11.1.13	kernel_interrupt_management.bind_interrupt_handler generic_interrupt_management.bind_interrupt_handler
11.1.14	generic_interrupt_globals
11.1.15	kernel_interrupt_management generic_interrupt_management

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
11.1.16	time_keeper.initialize low_level_interrupt_management.initialize interrupt_names bus_io.initialize
11.1.17	generic_interrupt_globals
11.1.18	kernel_exceptions
11.1.19	generic_interrupt_globals
11.1.20	kernel_interrupt_management generic_interrupt_management
11.1.21	interrupt_names generic_interrupt_globals
11.2.1	llim_body_machine_code kim_body_machine_code
11.2.2	llim_body_machine_code kim_body_machine_code
11.2.3	kernel_interrupt_management.bind_interrupt_handler generic_interrupt_management.bind_interrupt_handler
11.2.4	llim_body_machine_code kim_body_machine_code

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
12.1.1	generic_time_globals generic_kernel_time
12.1.2	generic_time_management clock.adjust_epoch_time clock.adjust_elapsed_time clock.get_time clock.stop_clock clock.start_clock
12.1.3	timer_controller.get_current_count timer_controller.acknowledge_timer_interrup timer_controller.disable_timer timer_controller.enable_timer timer_controller.initialize_timer low_level_clock generic_time_management clock.get_time
12.1.4	clock.adjust_epoch_time clock.adjust_elapsed_time clock.get_time clock.stop_clock clock.start_clock
12.1.5	generic_kernel_time
12.1.6	generic_kernel_time
12.1.7	generic_time_management clock.adjust_epoch_time clock.adjust_elapsed_time clock.get_time clock.stop_clock clock.start_clock
12.1.8	generic_time_management
12.1.9	generic_time_management generic_time_globals.create_epoch_time generic_time_globals.create_elapsed_time
12.1.10	time_keeper.adjust_elapsed_time generic_time_management.adjust_elapsed_time clock.adjust_elapsed_time
12.1.11	time_keeper.adjust_elapsed_time generic_time_management generic_process_table
12.1.12	time_keeper.reset_epoch_time generic_time_management.adjust_epoch_time clock.adjust_epoch_time

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
12.1.13	time_keeper.reset_epoch_time generic_time_management generic_process_table
12.1.14	generic_time_management
12.1.15	timer_controller.get_current_count low_level_clock generic_time_management.read_clock
12.1.16	generic_time_management.synchronize generic_time_management.synchronize generic_time_management.synchronize
12.1.17	time_keeper.remove_event time_keeper.remove_event time_keeper.insert_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.cancel_timer timer_controller.set_timer timer_controller.set_timer generic_time_management.synchronize generic_time_management.synchronize generic_time_management.synchronize
12.1.18	generic_time_management.synchronize generic_time_management.synchronize generic_process_table
12.1.19	generic_process_table
12.1.20	generic_time_management.synchronize generic_time_management.synchronize generic_time_management.synchronize
12.1.21	generic_time_management.synchronize generic_time_management.synchronize generic_time_management.synchronize
12.1.22	generic_time_management.synchronize generic_time_management.synchronize

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
12.1.23	gkt_body_machine_code generic_time_globals.to_epoch_time generic_time_globals.to_elapsed_time generic_time_globals.to_kernel_time generic_time_globals.to_kernel_time generic_time_globals.to_Ada_duration generic_time_globals.to_elapsed_time generic_time_globals.">=" generic_time_globals.">" generic_time_globals."<=" generic_time_globals."<" generic_time_globals.">=" generic_time_globals.">" generic_time_globals."<=" generic_time_globals."<" generic_time_globals."" generic_time_globals."" generic_time_globals."" generic_time_globals."-" generic_time_globals."+" generic_time_globals.base_time generic_time_globals."-" generic_time_globals."-" generic_time_globals."+" generic_time_globals.microseconds generic_time_globals.milliseconds generic_time_globals.seconds generic_time_globals.seconds generic_kernel_time.">=" generic_kernel_time.">" generic_kernel_time."<=" generic_kernel_time."<" generic_kernel_time."-" generic_kernel_time."+" generic_kernel_time."" generic_kernel_time."" generic_kernel_time."" generic_kernel_time."-" generic_kernel_time."+" generic_kernel_time.seconds generic_kernel_time.milliseconds generic_kernel_time.seconds generic_kernel_time.seconds

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
12.1.24	gkt_body_machine_code generic_time_globals.to_epoch_time generic_time_globals.to_elapsed_time generic_time_globals.to_kernel_time generic_time_globals.to_kernel_time generic_time_globals.to_Ada_duration generic_time_globals.to_elapsed_time generic_time_globals.">=" generic_time_globals.">" generic_time_globals."<=" generic_time_globals."<" generic_time_globals.">=" generic_time_globals.">" generic_time_globals."<=" generic_time_globals."<" generic_time_globals."" generic_time_globals."" generic_time_globals."" generic_time_globals."-" generic_time_globals."+" generic_time_globals.base_time generic_time_globals."-" generic_time_globals."-" generic_time_globals."+" generic_time_globals.microseconds generic_time_globals.milliseconds generic_time_globals.seconds generic_time_globals.seconds generic_kernel_time.">=" generic_kernel_time.">" generic_kernel_time."<=" generic_kernel_time."<" generic_kernel_time."-" generic_kernel_time."+" generic_kernel_time."" generic_kernel_time."" generic_kernel_time."" generic_kernel_time."-" generic_kernel_time."+" generic_kernel_time.seconds generic_kernel_time.milliseconds generic_kernel_time.seconds generic_kernel_time.seconds
12.1.25	generic_time_globals generic_kernel_time
12.1.26	time_keeper
12.1.27	generic_time_management.synchronize generic_time_management.synchronize generic_process_table

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
12.2.1	time_keeper.adjust_elapsed_time generic_time_management.adjust_elapsed_time clock.adjust_elapsed_time
12.2.2	time_keeper.reset_epoch_time generic_time_management.adjust_epoch_time clock.adjust_epoch_time
12.2.3	generic_time_management.read_clock
12.2.4	generic_time_management.synchronize generic_time_management.synchronize generic_time_management.synchronize
12.2.5	generic_time_management.synchronize generic_time_management.synchronize generic_time_management.synchronize
12.2.8	clock.get_time

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
13.1.1	generic_alarm_management.set_alarm generic_alarm_management
13.1.2	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table
13.1.3	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_alarm_management.set_alarm
13.1.4	generic_alarm_management generic_alarm_management
13.1.5	generic_alarm_management.set_alarm generic_alarm_management generic_alarm_management
13.1.6	generic_alarm_management.set_alarm
13.1.7	generic_process_table generic_alarm_management.set_alarm
13.1.8	generic_process_table
13.1.9	generic_process_table generic_alarm_management.set_alarm
13.1.10	time_keeper.remove_event time_keeper.insert_event timer_controller.cancel_timer timer_controller.set_timer generic_process_table generic_alarm_management.cancel_alarm
13.1.11	generic_process_table
13.2.1	generic_alarm_management.set_alarm generic_alarm_management
13.2.2	generic_alarm_management.cancel_alarm
13.2.3	generic_alarm_management

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
14.1.1	generic_tool_interface.begin_collection
14.1.2	tool_interface_globals generic_tool_interface.begin_collection
14.1.3	tool_interface_globals generic_tool_interface.begin_collection
14.1.4	tool_interface_globals generic_tool_interface.begin_collection
14.1.5	tool_logger.log_message_contents tool_logger.log_message_attributes tool_logger.log_process_attributes tool_interface_globals scheduler.schedule_ih scheduler.schedule generic_process_attribute_modifiers.kill generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message
14.1.6	tool_logger.log_process_attributes tool_interface_globals generic_tool_interface.begin_collection generic_tool_interface
14.1.7	generic_tool_interface.begin_collection
14.1.8	generic_tool_interface.cease_collection
14.1.9	tool_logger.log_process_attributes tool_interface_globals generic_tool_interface.begin_collection generic_tool_interface
14.1.10	tool_logger.log_message_attributes tool_interface_globals generic_tool_interface.begin_collection generic_tool_interface
14.1.11	generic_tool_interface.cease_collection
14.1.12	tool_logger.log_message_attributes tool_interface_globals generic_tool_interface.begin_collection generic_tool_interface
14.1.13	tool_logger.log_message_contents tool_interface_globals generic_tool_interface.begin_collection generic_tool_interface

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
14.1.14	generic_tool_interface.cease_collection
14.1.15	tool_interface_globals generic_tool_interface.read_process_table generic_tool_interface.size_of_process_table generic_tool_interface generic_process_table.size_of_process_table
14.1.16	generic_tool_interface.read_interrupt_table generic_tool_interface
14.1.17	tool_logger.log_message_contents tool_logger.log_message_attributes tool_logger.log_process_attributes
14.1.18	generic_tool_interface.cease_collection generic_tool_interface.begin_collection
14.2.1	generic_tool_interface.begin_collection
14.2.2	generic_tool_interface.cease_collection
14.2.3	tool_logger.log_message_contents tool_logger.log_message_attributes tool_logger.log_process_attributes scheduler.schedule_ih scheduler.schedule generic_process_attribute_modifiers.kill generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message
14.2.4	tool_logger.log_message_contents tool_logger.log_message_attributes tool_logger.log_process_attributes generic_tool_interface.cease_collection generic_tool_interface.begin_collection
14.2.5	tool_logger.log_message_contents tool_logger.log_message_attributes tool_logger.log_process_attributes scheduler.schedule_ih scheduler.schedule generic_process_attribute_modifiers.kill generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message

Package/Procedure to Requirements Mapping	
Requirement	Package/Procedure
14.2.6	tool_logger.log_message_contents tool_logger.log_message_attributes tool_logger.log_process_attributes scheduler.schedule_ih scheduler.schedule generic_process_attribute_modifiers.kill generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.receive_message generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message_and_wait generic_communication_management.send_message

Appendix G: Requirement to Procedure Mapping

A few notes about the requirements mapping tables:

- Only procedures and functions that appear at the package specification level occur in the table. No instantiations are included (only the generic source).
- Top-level package specifications occur only when there are internal objects that fulfill requirements.
- Overloaded names appear multiple times.
- The ordering of procedures and entries follows the order of the entries in the specification.

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
bus_io	10.1.33 10.1.34 10.1.36	
bus_io.initialize	5.1.3 5.1.4 11.1.16	6.2.1
bus_io.send_process_datagram	7.1.21 7.1.25 7.1.26 10.1.6 10.1.29 10.1.32	10.2.1 10.2.2 10.2.3 10.2.4 10.2.5 10.2.6 10.2.7 10.2.8 10.2.9 10.2.10
bus_io.send_kernel_datagram	7.1.21 7.1.25 7.1.26 10.1.6 10.1.29	10.2.1 10.2.2 10.2.3 10.2.4 10.2.5 10.2.6 10.2.7 10.2.8 10.2.9 10.2.10
bus_io.multi_send	6.1.4 6.1.9	
clock.start_clock	5.1.3 5.1.4 5.1.7 12.1.2 12.1.4 12.1.7	6.2.1
clock.stop_clock	5.1.3 5.1.4 5.1.7 12.1.2 12.1.4 12.1.7	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
clock.get_time	5.1.3 5.1.4 5.1.7 12.1.2 12.1.3 12.1.4 12.1.7	12.2.8
clock.adjust_elapsed_time	5.1.3 5.1.4 5.1.7 12.1.2 12.1.4 12.1.7 12.1.10	12.2.1
clock.adjust_epoch_time	5.1.3 5.1.4 5.1.7 12.1.2 12.1.4 12.1.7 12.1.12	12.2.2
context_save_area	5.1.4	
context_switcher.save_context	7.1.4	9.2.3
context_switcher.restore_process	7.1.4	9.2.4
context_switcher.switch_processes	7.1.4	9.2.6
context_switcher_globals	5.1.3 5.1.4	
dark_text_io	5.1.4	
dark_text_io.to_hex	5.1.4	
datagram_globals	5.1.4 6.1.11	
datagram_management.alloc_dg	5.1.4 6.1.11	
datagram_management.delete	5.1.4 6.1.11 10.1.21	
datagram_management.dequeue	5.1.4 6.1.11 10.1.21	
datagram_management.enqueue	5.1.4 6.1.11 10.1.21	
datagram_management.free_dg	5.1.4 6.1.11	
datagram_management.get_first	5.1.4 6.1.11 10.1.21	
datagram_management.new_queue	5.1.4 6.1.11 10.1.21	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
datagram_management.kproc_initialize	5.1.4 6.1.11	
datagram_management.nproc_initialize	5.1.4 6.1.11	
datagram_management.free_queue_status	5.1.4 6.1.11	
exception_raiser	5.1.1	
exception_raiser.raise_exception	5.1.1	
generic_alarm_management	5.1.8 13.1.4 13.1.5 13.2.3	
generic_alarm_management.set_alarm	9.1.6 9.1.8 9.1.22 13.1.1 13.1.4 13.1.5 13.1.6 13.1.7 13.1.9	13.2.1
generic_alarm_management.set_alarm	9.1.6 9.1.8 9.1.22 13.1.1 13.1.3 13.1.5 13.1.6 13.1.7 13.1.9	13.2.1
generic_alarm_management.cancel_alarm	9.1.22 13.1.10	13.2.2
generic_communication_globals	5.1.4 5.1.7	
generic_communication_management	6.1.9 6.1.11 10.1.19 10.1.22 10.1.23 10.1.28 10.1.29 10.1.30 10.1.31 10.1.33 10.1.34	
generic_communication_management.send_message	7.1.21 10.1.2 10.1.5 10.1.6 10.1.25 10.1.26 14.1.5	10.2.1 10.2.2 10.2.5 10.2.6 10.2.11 14.2.3 14.2.5 14.2.6

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_communication_management.send_message_and_wait	7.1.21	10.2.3
	7.1.25	10.2.4
	7.1.26	10.2.7
	9.1.6	10.2.8
	9.1.8	10.2.11
	9.1.22	14.2.3
	10.1.2	14.2.5
	10.1.3	14.2.6
	10.1.4	
	10.1.5	
	10.1.6	
	10.1.10	
	10.1.11	
	10.1.25	
	10.1.27	
	10.1.32	
	11.1.12	
	14.1.5	
generic_communication_management.send_message_and_wait	7.1.21	10.2.3
	7.1.25	10.2.4
	7.1.26	10.2.7
	9.1.6	10.2.8
	9.1.8	10.2.11
	9.1.22	14.2.3
	10.1.2	14.2.5
	10.1.3	14.2.6
	10.1.4	
	10.1.5	
	10.1.6	
	10.1.7	
	10.1.9	
	10.1.10	
	10.1.11	
	10.1.25	
	10.1.27	
	10.1.32	
	11.1.12	
	14.1.5	
generic_communication_management.send_message_and_wait	7.1.21	10.2.3
	7.1.25	10.2.4
	7.1.26	10.2.7
	9.1.6	10.2.8
	9.1.8	10.2.11
	9.1.22	14.2.3
	10.1.2	14.2.5
	10.1.3	14.2.6
	10.1.4	
	10.1.5	
	10.1.6	
	10.1.8	
	10.1.9	
	10.1.10	
	10.1.11	
	10.1.25	
	10.1.27	
	10.1.32	
	11.1.12	
	14.1.5	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_communication_management.receive_message	9.1.6 9.1.8 9.1.22 10.1.12 10.1.13 10.1.14 10.1.15 10.1.20 10.1.21 10.1.24 10.1.26 10.1.35 10.1.36 11.1.12 14.1.5	10.2.9 10.2.10 14.2.3 14.2.5 14.2.6
generic_communication_management.receive_message	9.1.6 9.1.8 9.1.22 10.1.12 10.1.13 10.1.14 10.1.15 10.1.16 10.1.18 10.1.20 10.1.21 10.1.24 10.1.26 10.1.35 10.1.36 11.1.12 14.1.5	10.2.9 10.2.10 14.2.3 14.2.5 14.2.6
generic_communication_management.receive_message	9.1.6 9.1.8 9.1.22 10.1.12 10.1.13 10.1.14 10.1.15 10.1.17 10.1.18 10.1.20 10.1.24 10.1.26 10.1.35 10.1.36 11.1.12 14.1.5	10.2.9 10.2.10 14.2.3 14.2.5 14.2.6
generic_communication_management.allocate_device_receiver	7.1.14	7.2.4
generic_interrupt_globals	11.1.1 11.1.4 11.1.5 11.1.6 11.1.14 11.1.17 11.1.19 11.1.21	
generic_interrupt_management	11.1.15 11.1.20	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_interrupt_management.enable	11.1.4	
generic_interrupt_management.disable	11.1.5	
generic_interrupt_management.enabled	11.1.7	
generic_interrupt_management.simulate_interrupt	11.1.2 11.1.3 11.1.8	
generic_interrupt_management.bind_interrupt_handler	11.1.2 11.1.9 11.1.10 11.1.11 11.1.13	11.2.3
generic_kernel_time	12.1.1 12.1.5 12.1.6 12.1.25	
generic_kernel_time.seconds	12.1.23 12.1.24	
generic_kernel_time.seconds	12.1.23 12.1.24	
generic_kernel_time.milliseconds	12.1.23 12.1.24	
generic_kernel_time.seconds	12.1.23 12.1.24	
generic_kernel_time."+"	12.1.23 12.1.24	
generic_kernel_time."-"	12.1.23 12.1.24	
generic_kernel_time.""	12.1.23 12.1.24	
generic_kernel_time.""	12.1.23 12.1.24	
generic_kernel_time."/"	12.1.23 12.1.24	
generic_kernel_time."+"	12.1.23 12.1.24	
generic_kernel_time."-"	12.1.23 12.1.24	
generic_kernel_time."<"	12.1.23 12.1.24	
generic_kernel_time."<="	12.1.23 12.1.24	
generic_kernel_time.">"	12.1.23 12.1.24	
generic_kernel_time.">="	12.1.23 12.1.24	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_network_configuration	5.1.6 6.1.6 6.1.10 6.1.11 7.1.14 7.1.16 10.1.6 10.1.15 10.1.26	6.2.1
generic_network_configuration.get_processor_id	6.1.11 10.1.6 10.1.15	6.2.1
generic_network_globals	5.1.3 5.1.4 5.1.8 6.1.11 10.1.6 10.1.15	
generic_processor_management	5.1.8 6.1.6 6.1.7 6.1.8 6.1.11	
generic_processor_management.initialize_master_processor	6.1.1 6.1.3 6.1.4 6.1.5	6.2.1 6.2.2
generic_processor_management.initialize_subordinate_processor	6.1.2 6.1.3 6.1.4	6.2.1
generic_processor_management.initialization_complete	7.1.15 7.1.16 7.1.17 7.1.25 7.1.26	6.2.1
generic_process_attribute_modifiers	5.1.8 6.1.9 7.1.1 7.1.2 7.1.13	
generic_process_attribute_modifiers.die	7.1.18 7.1.21 7.1.25 7.1.26 8.1.12	7.2.2
generic_process_attribute_modifiers.kill	7.1.18 7.1.20 7.1.25 7.1.26 8.1.12 14.1.5	7.2.2 14.2.3 14.2.5 14.2.6
generic_process_attribute_modifiers.set_process_preemption	9.1.12 9.1.22	9.2.2

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_process_attribute_modifiers.set_process_priority	9.1.6 9.1.8 9.1.22	9.2.1
generic_process_attribute_modifiers.wait	9.1.6 9.1.8 9.1.14 9.1.16 9.1.17 9.1.22 9.1.29 11.1.12	9.2.3
generic_process_attribute_modifiers.wait	9.1.6 9.1.8 9.1.14 9.1.15 9.1.17 9.1.22 9.1.29 11.1.12	9.2.3
generic_process_attribute_readers	5.1.8	
generic_process_attribute_readers.name_of	7.1.24 7.1.25 7.1.26 10.1.1	
generic_process_attribute_readers.who_am_i	7.1.23	
generic_process_attribute_readers.get_process_preemption	9.1.13	
generic_process_attribute_readers.get_process_priority	9.1.9	
generic_process_managers	5.1.6 5.1.8 6.1.9 7.1.1 7.1.2 7.1.8 7.1.13	
generic_process_managers.declare_process	7.1.3 10.1.1 10.1.6 10.1.15	6.2.1
generic_process_managers.declare_process	7.1.3 10.1.1 10.1.6 10.1.15 10.1.26	6.2.1

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_process_managers.create_process	7.1.4	6.2.1
	7.1.5	7.2.1
	7.1.6	7.2.5
	7.1.7	7.2.6
	7.1.8	
	7.1.9	
	7.1.10	
	7.1.12	
	7.1.25	
	7.1.26	
	7.1.27	
	9.1.1	
	9.1.2	
	9.1.7	
	9.1.11	
	10.1.22	
	10.1.23	
generic_process_managers_globals	10.1.22	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_process_table	5.1.6 7.1.4 7.1.6 7.1.7 7.1.9 7.1.21 7.1.27 8.1.1 8.1.2 8.1.3 8.1.4 8.1.5 8.1.6 8.1.9 8.1.10 8.1.11 8.1.16 9.1.7 9.1.8 9.1.9 9.1.11 9.1.12 9.1.13 9.1.15 9.1.16 9.1.19 9.1.20 9.1.23 9.1.29 10.1.2 10.1.3 10.1.7 10.1.8 10.1.12 10.1.16 10.1.17 10.1.22 10.1.37 10.1.38 10.1.39 12.1.11 12.1.13 12.1.18 12.1.19 12.1.27 13.1.2 13.1.3 13.1.7 13.1.8 13.1.9 13.1.10 13.1.11	8.2.1
generic_process_table."<="	5.1.4	
generic_process_table."<="	5.1.4	
generic_process_table.create_process_information_record	5.1.4	
generic_process_table.destroy_process_information_record	5.1.4	
generic_process_table.initialize_process_table	5.1.4	
generic_process_table.locate_process	5.1.4	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_process_table.size_of_process_table	5.1.4 14.1.15	
generic_queue_manager.create	5.1.4	6.2.1
generic_queue_manager.delete	5.1.4	5.2.4
generic_queue_manager.delete	5.1.4	5.2.4
generic_queue_manager.enqueue	5.1.4	5.2.4
generic_queue_manager.enqueue	5.1.4	5.2.4
generic_queue_manager.dequeue	5.1.4	5.2.4
generic_queue_manager.remove	5.1.4	5.2.4
generic_queue_manager.get_head	5.1.4	5.2.4
generic_queue_manager.get_element	5.1.4	5.2.4
generic_queue_manager.empty	5.1.4	5.2.4
generic_queue_manager.initialize_iterator	5.1.4	5.2.4
generic_queue_manager.initialize_backward_iterator	5.1.4	5.2.4
generic_queue_manager.end_of_queue	5.1.4	5.2.4
generic_queue_manager.get_next	5.1.4	5.2.4
generic_queue_manager.remove_next	5.1.4	5.2.4

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_schedule_types	5.1.6 5.1.8 9.1.2 9.1.3 9.1.4 9.1.5 9.1.10 9.1.23	
generic_semaphore_management	5.1.8 8.1.1 8.1.2 8.1.14	8.2.1
generic_semaphore_management.claim	8.1.3 8.1.4 8.1.7 8.1.8 8.1.9 8.1.16 9.1.6 9.1.8	8.2.2
generic_semaphore_management.claim	8.1.3 8.1.4 8.1.5 8.1.7 8.1.8 8.1.9 8.1.16 9.1.6 9.1.8	8.2.2
generic_semaphore_management.claim	8.1.3 8.1.4 8.1.6 8.1.7 8.1.8 8.1.9 8.1.16 9.1.6 9.1.8	8.2.2
generic_semaphore_management.release	8.1.10 8.1.11 8.1.13 8.1.15	8.2.3
generic_storage_manager.initialize	5.1.4	6.2.1
generic_storage_manager.allocate	5.1.4	5.2.4
generic_storage_manager.deallocate	5.1.4	5.2.4
generic_timeslice_management	5.1.8 9.1.3 9.1.4 9.1.5 9.1.10 9.1.21 9.1.24 9.1.26	9.2.4 9.2.6 9.2.7
generic_timeslice_management.set_timeslice	9.1.18	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_timeslice_management.enable_time_slicing	9.1.19	9.2.5
generic_timeslice_management.disable_time_slicing	9.1.20	9.2.5
generic_time_globals	12.1.1 12.1.25	
generic_time_globals.create_elapsed_time	12.1.9	
generic_time_globals.create_epoch_time	12.1.9	
generic_time_globals.seconds	12.1.23 12.1.24	
generic_time_globals.seconds	12.1.23 12.1.24	
generic_time_globals.milliseconds	12.1.23 12.1.24	
generic_time_globals.microseconds	12.1.23 12.1.24	
generic_time_globals."+"	12.1.23 12.1.24	
generic_time_globals."-"	12.1.23 12.1.24	
generic_time_globals."-"	12.1.23 12.1.24	
generic_time_globals.base_time	12.1.23 12.1.24	
generic_time_globals."+"	12.1.23 12.1.24	
generic_time_globals."-"	12.1.23 12.1.24	
generic_time_globals.""	12.1.23 12.1.24	
generic_time_globals.""	12.1.23 12.1.24	
generic_time_globals."/"	12.1.23 12.1.24	
generic_time_globals."<"	12.1.23 12.1.24	
generic_time_globals."<="	12.1.23 12.1.24	
generic_time_globals.">"	12.1.23 12.1.24	
generic_time_globals.">="	12.1.23 12.1.24	
generic_time_globals."<"	12.1.23 12.1.24	
generic_time_globals."<="	12.1.23 12.1.24	
generic_time_globals.">"	12.1.23 12.1.24	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_time_globals.">="	12.1.23 12.1.24	
generic_time_globals.to_elapsed_time	12.1.23 12.1.24	
generic_time_globals.to_Ada_duration	12.1.23 12.1.24	
generic_time_globals.to_kernel_time	12.1.23 12.1.24	
generic_time_globals.to_kernel_time	12.1.23 12.1.24	
generic_time_globals.to_elapsed_time	12.1.23 12.1.24	
generic_time_globals.to_epoch_time	12.1.23 12.1.24	
generic_time_management	5.1.8 6.1.6 12.1.2 12.1.3 12.1.7 12.1.8 12.1.9 12.1.11 12.1.13 12.1.14	
generic_time_management.adjust_elapsed_time	12.1.10	12.2.1
generic_time_management.adjust_epoch_time	12.1.12	12.2.2
generic_time_management.read_clock	12.1.15	12.2.3
generic_time_management.synchronize	12.1.16 12.1.17 12.1.20 12.1.21	12.2.4 12.2.5
generic_time_management.synchronize	12.1.16 12.1.17 12.1.18 12.1.20 12.1.21 12.1.22 12.1.27	12.2.4 12.2.5
generic_time_management.synchronize	12.1.16 12.1.17 12.1.18 12.1.20 12.1.21 12.1.22 12.1.27	12.2.4 12.2.5
generic_tool_interface	14.1.6 14.1.9 14.1.10 14.1.12 14.1.13 14.1.15 14.1.16	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
generic_tool_interface.begin_collection	14.1.1 14.1.2 14.1.3 14.1.4 14.1.6 14.1.7 14.1.9 14.1.10 14.1.12 14.1.13 14.1.18	14.2.1 14.2.4
generic_tool_interface.cease_collection	14.1.8 14.1.11 14.1.14 14.1.18	14.2.2 14.2.4
generic_tool_interface.size_of_process_table	14.1.15	
generic_tool_interface.read_process_table	14.1.15	
generic_tool_interface.read_interrupt_table	14.1.16	
gkt_body_machine_code	5.1.4 12.1.23 12.1.24	
hardware_interface	5.1.4	
hardware_interface.to_hw_bits8	5.1.4	
hardware_interface.to_hw_byte_ptr	5.1.4	
hardware_interface.to_hw_bits8_ptr	5.1.4	
hardware_interface.to_hw_bits8_ptr	5.1.4	
internal_process_management.get_next_process_number	5.1.4 10.1.1	5.2.4
internal_process_management.get_null_process_number	5.1.4 10.1.1	5.2.4
internal_process_management.create_internal_process	5.1.4 6.1.11	6.2.1
internal_process_management.create_kernel_processes	5.1.4 6.1.11	6.2.1
interprocessor_interrupts	5.1.4 6.1.11 11.1.6	
interprocessor_interrupts.enable_kn_interrupt	5.1.4 6.1.11	
interprocessor_interrupts.generate_kn_interrupt	5.1.4 6.1.11	
interrupt_names	11.1.1 11.1.16 11.1.21	
ipi_body_machine_code	5.1.4 6.1.11	
kernel_encapsulation.in_interrupt	11.1.2	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
kernel_exceptions	5.1.1 11.1.18	
kernel_exceptions.to_string	5.1.4	
kernel_interrupt_management	11.1.15 11.1.20	
kernel_interrupt_management.enable	11.1.4	
kernel_interrupt_management.disable	11.1.5	
kernel_interrupt_management.enabled	11.1.7	
kernel_interrupt_management.simulate_interrupt	11.1.8	
kernel_interrupt_management.bind_interrupt_handler	11.1.2 11.1.9 11.1.10 11.1.11 11.1.13	11.2.3
kim_body_machine_code	5.1.4	11.2.1 11.2.2 11.2.4
llcs_body_machine_code	5.1.4	
llh_body_machine_code	5.1.4 6.1.11	
llim_body_machine_code	5.1.4	11.2.1 11.2.2 11.2.4
llpe_body.Initialize_Process_State	5.1.4	
low_level_clock	5.1.4 12.1.3 12.1.15	
low_level_context_switcher.hw_save_context	7.1.4	
low_level_context_switcher.hw_restore_process	7.1.4	
low_level_context_switcher.hw_switch_processes	7.1.4	
low_level_hardware	5.1.3 5.1.4 6.1.11	
low_level_hardware.my_network_address	5.1.3 5.1.4 6.1.11	
low_level_hardware.is_Kproc	5.1.3 5.1.4 6.1.11	
low_level_hardware.is_Nproc	5.1.3 5.1.4 6.1.11	
low_level_hardware.P	5.1.3 5.1.4 6.1.11	
low_level_hardware.V	5.1.3 5.1.4 6.1.11	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
low_level_hardware.set_interrupt_priority	5.1.3 5.1.4 6.1.11	
low_level_hardware.reset_interrupt_priority	5.1.3 5.1.4 6.1.11	
low_level_interrupt_management		
low_level_interrupt_management.bind_fast_interrupt	5.1.3 5.1.4 11.1.11	
low_level_interrupt_management.bind_slow_interrupt	5.1.3 5.1.4 11.1.11	
low_level_interrupt_management.initialize	5.1.3 5.1.4 11.1.16	
low_level_process_encapsulation.initialize_process_state	7.1.4	
low_level_process_encapsulation.indirect_call	7.1.4 7.1.5	
low_level_storage_manager.allocate	5.1.4	
memory_addresses	5.1.3 5.1.4 6.1.11	
mvme133A_definitions	5.1.3 5.1.4 6.1.11	
mz8305_definitions	5.1.3 5.1.4 6.1.11	
network_configuration	5.1.6 6.1.10 6.1.11 7.1.14 7.1.16 10.1.6 10.1.15 10.1.26	6.2.1
network_configuration.get_processor_id	6.1.11 10.1.6 10.1.15	6.2.1
nproc.main_unit	5.1.3 5.1.4 6.1.11	

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
parallel_io_controller.initialize_rcv;	5.1.3 5.1.4 6.1.11	
parallel_io_controller.initialize_xmit;	5.1.3 5.1.4 6.1.11	
parallel_io_controller.enable_rcv_interrupt;	5.1.3 5.1.4 6.1.11	
parallel_io_controller.enable_xmit_interrupt;	5.1.3 5.1.4 6.1.11	
parallel_io_controller.disable_rcv_interrupt;	5.1.3 5.1.4 6.1.11	
parallel_io_controller.disable_xmit_interrupt;	5.1.3 5.1.4 6.1.11	
parallel_io_controller.rcv_buffer_full	5.1.3 5.1.4 6.1.11	
parallel_io_controller.xmit_buffer_empty	5.1.3 5.1.4 6.1.11	
parallel_io_controller.acknowledge_rcv_interrupt	5.1.3 5.1.4 6.1.11	
parallel_io_controller.acknowledge_xmit_interrupt	5.1.3 5.1.4 6.1.11	
parallel_io_controller.send_packet	5.1.3 5.1.4 6.1.11	
parallel_io_controller.receive_packet	5.1.3 5.1.4 6.1.11	
process_encapsulation	7.1.12	
process_encapsulation.dummy_call_frame	7.1.4	
process_index_table.get_process_identifier	6.1.11 10.1.6 10.1.15	5.2.4
process_index_table.get_process_index	6.1.11 10.1.6 10.1.15	5.2.4
process_index_table.set_process_index	6.1.11 10.1.6 10.1.15	6.2.1
process_index_table.clear_process_index	6.1.11 10.1.6 10.1.15	6.2.1

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
scc_porta.allocate	5.1.3 5.1.4	5.2.4
scc_porta.deallocate	5.1.3 5.1.4	5.2.4
scc_porta.disable_rx_interrupts	5.1.3 5.1.4	5.2.4
scc_porta.disable_tx_interrupts	5.1.3 5.1.4	5.2.4
scc_porta.enable_rx_interrupts	5.1.3 5.1.4	5.2.4
scc_porta.enable_tx_interrupts	5.1.3 5.1.4	5.2.4
scc_porta.get	5.1.3 5.1.4	5.2.4
scc_porta.get_byte	5.1.3 5.1.4	5.2.4
scc_porta.initialize	5.1.3 5.1.4	5.2.4 6.2.1
scc_porta.put_byte	5.1.3 5.1.4	5.2.4
scc_porta.send	5.1.3 5.1.4	5.2.4
scc_porta.set	5.1.3 5.1.4	5.2.4
scheduler	9.1.26 9.1.28	
scheduler.initialize	5.1.4	
scheduler.insert_process	9.1.5 9.1.23	5.2.4
scheduler.remove_process	9.1.23	5.2.4
scheduler.schedule	9.1.6 9.1.8 9.1.12 9.1.23 9.1.24 9.1.25 9.1.27 14.1.5	5.2.4 9.2.4 9.2.6 9.2.7 14.2.3 14.2.5 14.2.6
scheduler.schedule_ih	9.1.23 9.1.24 9.1.25 14.1.5	5.2.4 9.2.4 9.2.6 9.2.7 14.2.3 14.2.5 14.2.6
tc_body_machine_code	5.1.4	
timer_controller.initialize	5.1.3 5.1.4 5.1.7	6.2.1

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
timer_controller.initialize_timer	12.1.3	
timer_controller.enable_timer	12.1.3	
timer_controller.disable_timer	12.1.3	
timer_controller.acknowledge_timer_interrupt	12.1.3	
timer_controller.get_current_count	12.1.3 12.1.15	
timer_controller.set_timer	5.1.3 5.1.4 5.1.7 8.1.5 8.1.6 9.1.15 9.1.16 9.1.19 9.1.20 10.1.7 10.1.8 10.1.16 10.1.17	
timer_controller.cancel_timer	5.1.3 5.1.4 5.1.7 8.1.5 8.1.6 9.1.15 9.1.16 9.1.19 9.1.20 10.1.7 10.1.8 10.1.16 10.1.17	
time_burner.sponge	5.1.4	
timeslice_parameters	9.1.18 9.1.19 9.1.20 9.1.21	
time_keeper	8.1.12 12.1.26	
time_keeper.initialize	5.1.3 5.1.4 5.1.7 11.1.16	6.2.1

Requirements to Package/Procedure Mapping		
Package/Procedure	Behavior	Performance
time_keeper.insert_event	5.1.3 5.1.4 5.1.7 8.1.5 8.1.6 9.1.15 9.1.16 9.1.19 9.1.20 10.1.7 10.1.8 10.1.16 10.1.17	
time_keeper.remove_event	5.1.3 5.1.4 5.1.7 8.1.5 8.1.6 9.1.15 9.1.16 9.1.19 9.1.20 10.1.7 10.1.8 10.1.16 10.1.17	
time_keeper.adjust_elapsed_time	12.1.10 12.1.11	12.2.1
time_keeper.reset_epoch_time	12.1.12 12.1.13	12.2.2
time_keeper_globals	5.1.4	
tool_interface_globals	14.1.2 14.1.3 14.1.4 14.1.5 14.1.6 14.1.9 14.1.10 14.1.12 14.1.13 14.1.15	
tool_interface_output	5.1.7	
tool_logger.log_process_attributes	14.1.5 14.1.6 14.1.9 14.1.17	14.2.3 14.2.4 14.2.5 14.2.6
tool_logger.log_message_attributes	14.1.5 14.1.10 14.1.12 14.1.17	14.2.3 14.2.4 14.2.5 14.2.6
tool_logger.log_message_contents	14.1.5 14.1.13 14.1.17	14.2.3 14.2.4 14.2.5 14.2.6

Appendix H: Short Names

Package Short Names	
Package	Short Name
ALARM_MANAGEMENT	AM
BUS_IO	BIO
CLOCK	C
COMMUNICATION_GLOBALS	CG
COMMUNICATION_MANAGEMENT	CM
CONTEXT_SAVE_AREA	CSA
CONTEXT_SWITCHER	CS
CONTEXT_SWITCHER_GLOBALS	CSG
CS_BODY	CS
DATAGRAM_GLOBALS	DGG
DATAGRAM_MANAGEMENT	DGM
EXCEPTION_RAISER	ER
GENERIC_ALARM_MANAGEMENT	GAM
GENERIC_COMMUNICATION_GLOBALS	GCG
GENERIC_COMMUNICATION_MANAGEMENT	GCM
GENERIC_INTERRUPT_GLOBALS	GIG
GENERIC_INTERRUPT_MANAGEMENT	GIM
GENERIC_KERNEL_TIME	GKT
GENERIC_NETWORK_CONFIGURATION	GNC
GENERIC_NETWORK_GLOBALS	GNG
GENERIC_PROCESSOR_MANAGEMENT	GRM
GENERIC_PROCESS_ATTRIBUTE_MODIFIERS	GPAM
GENERIC_PROCESS_ATTRIBUTE_READERS	GPAP
GENERIC_PROCESS_MANAGERS	GPM
GENERIC_PROCESS_MANAGERS_GLOBALS	GPMG
GENERIC_PROCESS_TABLE	GPTB
GENERIC_QUEUE_MANAGER	GQM
GENERIC_SCHEDULE_TYPES	GST
GENERIC_SEMAPHORE_MANAGEMENT	GMM
GENERIC_STORAGE_MANAGER	GSTM

Package	Short Name
GENERIC_TIMESLICE_MANAGEMENT	GTSM
GENERIC_TIME_GLOBALS	GTG
GENERIC_TIME_MANAGEMENT	GTM
GENERIC_TOOL_INTERFACE	GTI
HARDWARE_INTERFACE	HI
INTERPROCESSOR_INTERRUPTS	IPI
INTERRUPT_GLOBALS	IG
INTERRUPT_MANAGEMENT	IM
INTERRUPT_NAMES	INames
KERNEL_ENCAPSULATION	KEN
KERNEL_EXCEPTIONS	KE
KERNEL_INTERRUPT_MANAGEMENT	KIM
KERNEL_TIME	KT
KIM_BODY	KIM
LLIM_BODY	LLIM
LOW_LEVEL_CONTEXT_SWITCHER	LLCS
LOW_LEVEL_INTERRUPT_MANAGEMENT	LLIM
LOW_LEVEL_PROCESS_ENCAPSULATION	LLPE
LOW_LEVEL_STORAGE_MANAGER	LLSM
MEMORY_ADDRESSES	MEM
MVME133A_DEFINITIONS	MVME
MZ8305_DEFINITIONS	MZ
NCT_DEBUG	NCTD
NETWORK_CONFIGURATION	NC
NETWORK_GLOBALS	NG
NPROC	NPROC
PARALLEL_IO_CONTROLLER	PIO
PE_BODY	PE
PIO_BODY	PIO
PROCESSOR_MANAGEMENT	RM
PROCESS_ATTRIBUTE_MODIFIERS	PAM
PROCESS_ATTRIBUTE_READERS	PAR
PROCESS_ENCAPSULATION	PE

Package	Short Name
PROCESS_INDEX_TABLE	PIT
PROCESS_MANAGERS	PM
PROCESS_MANAGERS_GLOBALS	PMG
PROCESS_TABLE	PTB
PTB_DEBUG	PTB_DEBUG
SCC_PORTA	PORTA
SCHEDULER	SCH
SCHEDULE_TYPES	ST
SEMAPHORE_MANAGEMENT	MM
TC_BODY	TC
TIMER_CONTROLLER	TC
TIMESLICE_MANAGEMENT	TSM
TIMESLICE_PARAMETERS	TSP
TIME_GLOBALS	TG
TIME_KEEPER	TK
TIME_KEEPER_GLOBALS	TKG
TIME_MANAGEMENT	TM
TOOL_INTERFACE	TI
TOOL_INTERFACE_GLOBALS	TIG
TOOL_INTERFACE_OUTPUT	TIO
TOOL_LOGGER	TL

Appendix I: Overview of VMS Version

The VMS version of the Kernel has the same functionality and structure as the 68020 version. The algorithms shown in the PDL of this document are applicable to both the 68020 and VMS versions. It is only in the code that the impact of each specific target manifests itself.

The resulting logical structure of the VMS version is shown in Figure 47. This view highlights where VMS concepts were applied to the Kernel.

- Each "node" is a single VMS process, with the Main Unit as the driver and the Kernel processes executing under control the Kernel. All of the "nodes" run on a single VAX processor, using windows or terminals to simulate nodes in a network.¹³ *This simulates the 68020 version, where each Main Unit executes on its own, dedicated processor.*
- Each Kernel uses a unique VMS TIMER for use in maintaining timeout and alarm operations. *This replaces the 68020 parallel_io timers with VMS timers.*
- Each Kernel uses the shared VMS (system) clock. *This replaces the individual 68020 clocks with one VMS system clock.*
- Each Kernel has its own mailbox, designated by node number, for receiving input from any other node in the system. *This replaces the 68020 interprocessor interrupts with VMS Asynchronous System Traps (ASTs).*
- Each Kernel has access to the mailbox of all other Kernels in the system (as illustrated for node 0 in Figure 47). *This replaces the Nproc (and all associated I/O packages) with VMS system services and shared mailboxes for communicating with remote Kernels.*

The net result is that all the special-purpose hardware required for the 68020 version (timers, parallel I/O controllers, Nproc, etc.) has all been absorbed by services provided by VMS. Additional information about the VMS implementation is described in [port 89].

¹³It may be possible to run each node on a different VAX machine communicating over DECnet, but no work has been pursued along these lines.

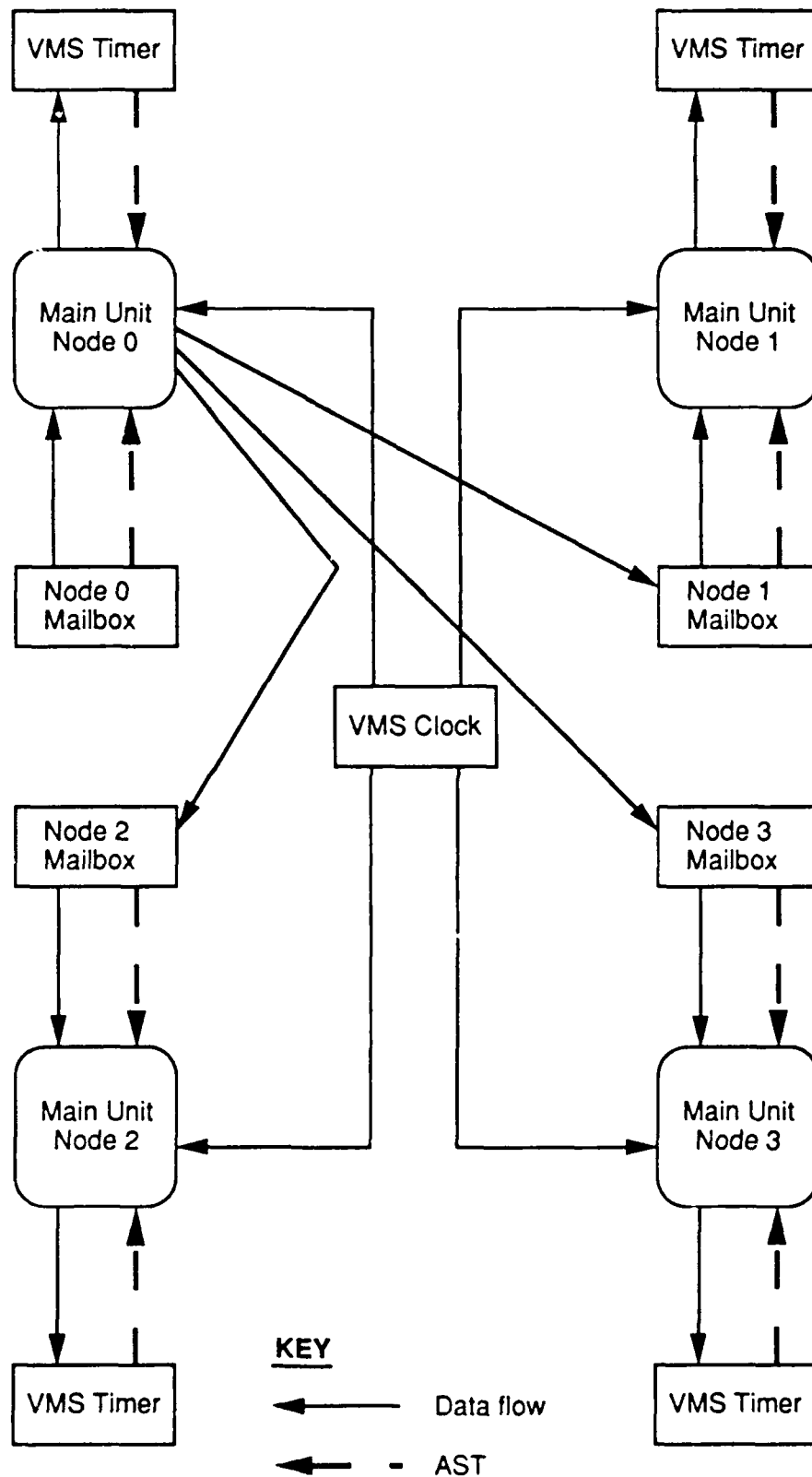


Figure 47: VMS Overview

Appendix J: VMS Ada Compiler Dependencies

This appendix explains the compiler and machine dependencies of the DARK port to VAX/VMS. The structure of this appendix parallels that of Part IX, TeleSoft Ada Compiler Dependencies.

J.1. Introduction

The DARK Project has ported DARK to VAX/VMS. This required three sets of changes

1. Adapting the compiler-dependent packages to VAX Ada.
2. Adapting the machine-dependent parts to the VAX-11.
3. Replacing some functional modules by others that invoke VAX Ada runtime services.

This appendix addresses the first two items; the third is explained in [port 89].

J.1.1. Relevant Documents

VAX Ada Language Reference Manual [VLRM] (DEC AA-EG29A-TE)

VAX Ada Programmers Runtime Reference Manual [VPRRM] (DEC AA-EF88A-TE)

[DARK Ada Style Guide]

[DARK VDIG]

J.2. Major Dependencies

The DARK software exhibits dependencies on the Ada development system in three major areas:

- Aspects of basic software architecture and design.
- Representation and use of basic data types.
- Encapsulation of hand-coded assembler.

These will be discussed at successively greater levels of detail.

J.2.1. Software Architecture and Design

The features considered efficient by the Ada style guide for the most part remain efficient, including the representation of guarded regions.

There are two potential new areas of inefficiency

1. Parameters are never passed by value; they are always passed by reference and the caller makes a local copy. If the corresponding actual is a packed

record component, or an object of a constrained subtype, the caller also makes a copy. This is very inefficient; its potential impact on DARK is minor, though, since the VMS version is not intended to support real-time applications [VPRRM 3].

2. When the target of an `Unchecked_Conversion` is a constrained record type, the compiler generates a constraint check. This potentially affects the datagram allocation code [VLRM 13.10].

J.2.2. Basic Data Types

The basic data types were in accordance with the expectations of the project; the differences from TeleSoft MC68020 Ada are within the scope of the `Hardware_Interface` abstraction [VLRM C].

The type `System.Address` is private, but there are many useful operations defined on it [VLRM F].

J.2.3. Encapsulation of Assembler

Full information is given in [VLRM 13.9]. It is straightforward to interface to separate machine-code bodies, though the specifications all have to be changed. It is also possible to interface data objects as well as subprograms; this feature makes it easier for machine-code bodies to see global variables declared in Ada [VLRM 1309a].

However, the compiler does not support the package `Machine_Code` [VLRM 13.8], so one module of DARK had to be rewritten (the subprogram `Process_Encapsulation.Indirect_Call`).

Calling conventions are given in full in [VPRRM 3.4]. To a limited extent, they are tailorable.

J.3. Software Architecture and Design Dependencies

This chapter gives the main compiler dependencies and the findings with respect to the VAX Ada compiler.

J.3.1. Code Customization

Coding conventions rely on these features of the Ada compiler:

- Generic instantiation is performed by code substitution at compile time.
- Constant actual values are substituted for the formals, and simple constant comparisons will be done at compile time.
- Code guarded by conditional statements that are known at compile time to be false ("dead code") will be removed by the compiler and will not generate any object code. (This is not necessary for correct functioning of the Kernel, but makes it smaller and faster.)

According to the compiler documentation, all these assumptions are valid. The compiler tests showed that the given optimizations are, in fact, performed.

J.3.2. Representation of Errors

The Kernel systematically represents error conditions by user-defined exceptions, and reports them by raising the exception, presumably to be handled by the invoking code.

For this to be feasible, certain compiler features are assumed:

- The execution cost of guarded regions and unraised exceptions is very small, preferably zero.
- The cost of raising and propagating an exception is reasonable; in particular, it is not so great as to prevent timely recovery by the exception handler.
- The exception mechanism can function safely and accurately in the context of a DARK process.

The first assumption is false. The VAX hardware automatically creates a null guarded region as part of the procedure call mechanism; this costs one longword of space and a certain amount of time. However, since this overhead is an inescapable part of the subprogram call, it might as well be put to good use. The marginal cost to DARK is, indeed, zero.

The cost of raising and propagating an exception is high, but unavoidable given the VAX signal-handling design. An explicit raise is encoded as a call to a library routine. Unfortunately, so is a compiler-generated raise, which implies that a successful constraint check, for example, is followed by a branch around a call, the call and associated parameters being big enough to cause an I-cache miss.

Since the exception mechanism is part of the standard call sequence, exceptions can freely propagate through assembler bodies. Means exist to raise exceptions from assembler code. A stack plug can be constructed either in Ada or in assembler.

The current DARK alarm management design will work without change.

J.3.3. Module Initialization

The pragma ELABORATE is implemented [VLRM B]. No difficulty was anticipated in the module initialization code. Unfortunately, there were difficulties with generic instantiations; as a consequence, it was necessary to include in every module that performs an instantiation a pragma ELABORATE in specifying the generic being instantiated. This change could safely be made in the other version of DARK, so is not VMS specific.

J.3.4. Chapter 13 Issues

Those parts of the Kernel that manipulate the target machine rely to some extent on the features provided in Chapter 13 of the *Ada Language Reference Manual*.

The specific features required are:

- Size specifications [13.2(a)] are implemented, and will indeed pack objects to the necessary bit or byte level.

- Record representation clauses [13.3] are implemented as needed.
- Address clauses [13.5] are *not* implemented. The trick of using an access value set by hand will work, but was not needed in the port since there was no longer a requirement to bind a data structure to a specific address.
- The package *System* [13.7] is present as needed.
- The system-dependent named numbers [13.7.1] are defined properly.
- The attribute *'Address* [13.7.2] is defined for both objects and subprograms. There are some words about subprograms in [VLRM 13.7.2], whose effect is:
 - any subprogram whose address is taken must be the subject of an *EXPORT* pragma
 - such subprograms must be declared at the outermost level of a package specification or body
- The appropriate pragmata were added to comply with the first restriction; the DARK code already obeyed the second.
- The attribute *'Size* [13.7.2] is implemented.
- The machine-code insertion facility [13.8] is *not* implemented; the relevant module had to be rewritten.
- The pragma *Interface* is implemented as required. In fact, it is rather more powerful than required. In particular, it allows data objects to be shared between Ada and assembler – which allowed some of the code to be simplified – and it allows, to a limited extent, the user to specify the parameter-passing strategies to be used when Ada calls assembler. Unfortunately, one strategy it does not permit is to pass the parameters in registers
- The pragma *Inline* is implemented, and works across compilation boundaries.
- The generic *Unchecked_Conversion* [13.10.2] is implemented for all relevant combinations of types. In addition, the VAX version of package *System* contains some useful instantiations, including for example conversions between integers and addresses. One problem with the VAX implementation, however, is that the compiler insists on checking the constraints of the target subtype after the conversion. Problems were anticipated with this, but were not in fact encountered.

J.3.5. Pragmas

The Kernel uses the following standard pragmas:

- pragma *INLINE*.
- pragma *INTERFACE*.

These are both implemented, as detailed above.

J.3.6. Ada Use Subset

As well as making certain assumptions about what the Ada system does provide, the Kernel was designed and written under certain assumptions about what it *need not* provide. In effect, it employs an *application subset* of the language, avoiding constructs that the implementation team believed either unnecessary or possibly inefficient.

- The Kernel makes no use of tasking.
- Records with discriminants are not used. This has no visible impact on the application, but has caused some slightly strange coding styles in parts of the Kernel.
- Objects of dynamic size are never declared within subprograms.
- Subprograms are not nested within other subprograms.
- Allocated storage is never deallocated, either explicitly or implicitly. All uses of the Ada allocator could be removed from the Kernel, if it seemed desirable for a port to use a custom storage-management system.
- The **separate** clause is not used. This is to avoid the name management problems that arise with library subunits.
-

All the above were still worth avoiding. Objects of dynamic size are particularly unpleasant in VAX Ada.

J.4. Basic Data Types and Operations

Appendix L contains the hardware tailoring for the VAX-11 with the current compiler and version.

J.5. Encapsulation of Assembly Code

Appendix A contains the assembler interface for the VAX-11 with the current compiler and version.

Appendix K: VAX-11 Assembler Interface

This appendix gives the assembler interface used in the VAX-11 with the current compiler and version. Full details of the compiler and version conventions can be found in [VPRRM 3].

K.1. Linkage

Linkage is effected by the pragma `INTERFACE`, which can also specify the parameter-passing strategy and the linkname. It is possible to implement overloaded subprograms in machine code, provided the linknames are unique.

The linker accepts 31 characters as significant, which was enough to allow the pervious DARK conventions for linkname generation to be used unchanged.

Within the assembler body, the linkname is generated by a standard `GLOBAL` directive:

```
.GLOBAL Linkname
```

The assembler routines must be presented to the Ada library as implementations of package or subprogram bodies, as described in [VPPRM 3]. A file may contain either Ada code or assembler code, but not both. DARK naming conventions require that a file containing assembler code be named exactly as it would be if it were in Ada, but with the additional suffix `_machine_code`. An Ada specification can be implemented, therefore, by at most one Ada body and one assembler code body. However, the VMS Ada library does not permit more than one body for any specification. In that event, only one DARK module had originally possessed two bodies – `process_encapsulation` – and that had to be revised for other reasons.

K.2. Program and Data Sections

Code and data must be generated in the appropriate sections (PSECTs) and with the correct attributes. These are given in [VPRRM 3.4] and were copied exactly in the machine-code bodies. In particular, the code must be read-only, reentrant, and position independent.

K.3. Data Representation

The data representations common to both Ada and assembler levels are as specified in Appendix A. There was no difficulty with these representations.

The supplied package *Standard* also defines a 64-bit integer data type and associated operations. Unfortunately, it is defined as `Unsigned_Quadword`, which made it inappropriate as an efficient representation of `Kernel_Time`, so the DARK module `kernel_time` had to be rewritten in VAX machine code.

The definition of time used by the Kernel was, however, changed to correspond to the VMS representation. This is as a 64-bit signed integer with 100 ns resolution. This change was encapsulated in the `kernel_time` module; the only external impact is that the legal range of epoch and elapsed time contracted to approximately 15,000 years.

K.4. Procedural Interface

The procedural interface uses the following protocol [VPRRM 3]

K.4.1. Entry and Exit Protocol

- Call is by a CALLS instruction, according to the VAX standard. This requires a valid stack front pointer to be maintained in SP, and a valid frame pointer in FP. Parameters are evaluated and pushed onto the stack in *reverse order* before the call.
- Return is likewise by an RET instruction, which pops the stack frame *and* the parameters. Accordingly, *out* parameters cannot be returned by value – the parameter space is deallocated by the RET before the caller can reclaim the returned value.

K.5. Register Jsage

- The called routine must save and restore any registers it uses except R0 and R1.
- It must return a simple function result in R0, if 32 bits or less, and in <R0,R1> if 64 bits or less and scalar. Other function results are returned in a hidden *out* parameter.

K.5.1. Stack Manipulation

- The called routine *must* at all times maintain in SP a valid hardware stack pointer. It must also maintain a valid frame pointer in FP.
- The called routine may claim local storage by lowering the stack pointer; it need not restore the old value since the RET instruction automatically restored SP from FP.
- The called routine does not have to build any special stack frame for Ada.

K.5.2. Parameter Passing

The parameter passing conventions are as follows.

K.5.3. Mode of Transmission

- Parameters are never passed by value. If the RM requires the effect of by-value transmission, the parameter is passed by reference and the caller makes a local copy. If the parameter is of mode *In out* or *out*, and the returned value might violate a constraint, the caller also makes a local copy and passes a pointer to it.
- Parameters of simple (scalar, access and address) types are passed by 'fake value' in this manner.
- Parameters larger than 4 bytes, and *all* parameters of structured types, are passed by simple reference. This reference is the machine address of the lowest-numbered storage unit, and is a 4-byte value.
- Some parameters of dynamic size are passed by VAX 'descriptor' of which the less said the better.

K.5.4. Manner and Order of Transmission

- Parameters are pushed on the hardware (SP) stack in *reverse* order. That is, the rightmost parameter is pushed on the stack first, and the leftmost last.
- All parameters passed by reference pass the address as an *In* parameter.

K.5.5. Accessing Parameters and Returning Function Results

- All simple parameters can be treated as if passed by reference.
- Results not larger than 4 bytes are returned in register R0. Scalar results not larger than 8 bytes are returned in <R0,R1>.
- Other results are returned in a hidden *out* parameter whose address is passed as the *first* parameter to the function, before the first true parameter.
- Although some Ada types require hidden parameters to be passed along with their actual values, no such types are used by any assembler subprogram.

K.6. Exceptions

K.6.1. Raising Exceptions

The assembler code never raises a user-defined exception. Where appropriate, it raises an intrinsic exception by an explicit call of LIB\$STOP, which is part of the Ada runtime.

K.6.2. Exception Propagation

The Ada runtime propagates exceptions upwards through stack frames, using the saved information in each stack frame to find each caller.

In order for this to function, every assembler body that can raise an exception, and every body *through which an exception might propagate*, must build a valid stack frame. This is done as part of the hardware CALLS instruction, which therefore shall be used throughout.

K.6.3. Guarded Regions

The representation of guarded regions was not researched.

Appendix L: VAX-11 Tailoring

What follows is barely different from the MC68020 version, and captures the identical functionality. Note that, had the types in package *System* been used directly, DARK would have had to change all uses of Integer, Long_Integer, and Address.

L.1. Sizes of Data Types

```
bits_per_byte : constant := 8;
```

```
byte          : constant := 1;
```

```
word          : constant := 2;
```

```
longword      : constant := 4;
```

L.2. Untyped Storage

```
type hw_byte is range 0..255;

type hw_bits8 is record
    bit0 : Boolean;
    bit1 : Boolean;
    bit2 : Boolean;
    bit3 : Boolean;
    bit4 : Boolean;
    bit5 : Boolean;
    bit6 : Boolean;
    bit7 : Boolean;
end record;

for hw_bits8 use record
    bit0 at 0 range 0..0;
    bit1 at 0 range 1..1;
    bit2 at 0 range 2..2;
    bit3 at 0 range 3..3;
    bit4 at 0 range 4..4;
    bit5 at 0 range 5..5;
    bit6 at 0 range 6..6;
    bit7 at 0 range 7..7;
end record;

function to_hw_bits8 is
    new unchecked_conversion(hw_byte, hw_bits8);

type hw_byte_ptr is access hw_byte;
type hw_bits8_ptr is access hw_bits8;

function to_hw_byte_ptr
    is new unchecked_conversion(system.address, hw_byte_ptr);

function to_hw_bits8_ptr
    is new unchecked_conversion(system.address, hw_bits8_ptr);

function to_hw_bits8_ptr
    is new unchecked_conversion(hw_byte_ptr, hw_bits8_ptr);
```

L.3. Integer Types

```
type hw_integer is range -32_768 .. 32_767;
for hw_integer' size use 2 * bits_per_byte;

type hw_short_integer is range -128 .. 127;
for hw_short_integer' size use 1 * bits_per_byte;

type hw_long_integer is
    range -2_147_483_648 .. 2_147_483_647;
for hw_long_integer' size use 4 * bits_per_byte;

type hw_natural is range 0 .. 32_767;
for hw_natural' size use 2 * bits_per_byte;

type hw_positive is range 1 .. 32_767;
for hw_positive' size use 2 * bits_per_byte;

type hw_long_natural is range 0 .. 2_147_483_647;
for hw_long_natural' size use 4 * bits_per_byte;

type hw_long_positive is range 1 .. 2_147_483_647;
for hw_long_positive' size use 4 * bits_per_byte;
```

L.4. Duration

```
type hw_duration is
    new duration range -86_400.0 .. +86_400.0;
for hw_duration' small use 2.0 ** (-14);
for hw_duration' size use 4 * bits_per_byte;
```

L.5. Machine Addresses

```
type hw_address is new system.address;

function to_hw_address is
    new unchecked_conversion(hw_long_integer, hw_address);

null_hw_address : constant hw_address
    := to_hw_address(hw_long_integer'(0));
```

L.6. Strings

`type hw_string is new string;`

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.
- [KFD 89] Bamberger, J., C. Colket, R. Firth, D. Klein, R. Van Scoy.
Kernel Facilities Definition.
Technical Report CMU/SEI-88-TR-16, ESD-TR-88-17, ADA198933, Software Engineering Institute, December, 1989.
- [KUM 89] Bamberger, J., T. Coddington, R. Firth, D. Klein, D. Stinchcomb, R. Van Scoy.
Kernel User's Manual.
Technical Report CMU/SEI-89-UG-1, ESD-TR-89-15, Software Engineering Institute, December, 1989.
- [port 89] Bamberger, J., T. Coddington, R. Firth, D. Klein, D. Stinchcomb, R. Van Scoy.
Kernel Porting and Extension Guide.
Technical Report CMU/SEI-89-TR-40, ESD-TR-89-51, Software Engineering Institute, Oct, 1989.
- [Raynal 86] Raynal, M.
Algorithms for Mutual Exclusion.
The MIT Press, Cambridge, MA, 1986.
- [TeleSoft 88] *TeleGen2 - The TeleSoft Second Generation Ada Development System for VAX/VMS to Embedded MC680X0 Targets*
TeleSoft, 1988.
- [Ward 85] Ward, P.T. and S.J. Mellor.
Structured Development for Real-Time Systems.
Yourdon Press, Englewood Cliffs, NJ, 1985.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-TR-19			5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-89-TR-27		
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.		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 HANSCOM, MA 01731		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE		8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003		
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO. 63752F	PROJECT NO. N/A	TASK NO. N/A
11. TITLE (Include Security Classification) KERNEL ARCHITECTURE MANUAL					
12. PERSONAL AUTHOR(S) Judy Bamberger, Timothy Coddington, Currie Colket, Robert Firth, Daniel Klein, Roger VanScoy, David Stinchcomb,					
13a. TYPE OF REPORT FINAL		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) December 1989	
				15. PAGE COUNT 373	
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.			
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION		
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER			22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630		22c. OFFICE SYMBOL SEI JPO