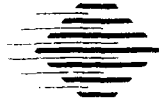


DTIC FILE COPY

Technical Report  
CMU/SEI-89-TR-40  
ESD-89-TR-51

2



Carnegie-Mellon University  
Software Engineering Institute

AD-A219 291

**DARK Porting and Extension Guide  
Kernel Version 3.0**

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

December 1989

DTIC  
ELECTE  
MAR 15 1990  
S D D CS

DISTRIBUTION STATEMENT A  
Approved for public release  
Distribution Unlimited

00-03814-047

**Technical Report**

**CMU/SEI-89-TR-40**

**ESD-89-TR-51**

**December 1989**

**DARK Porting and Extension Guide**

**Kernel Version 3.0**



**Judy Bamberger  
Timothy Coddington  
Daniel Klein  
David Stinchcomb  
Roger Van Scoy**

Distributed Ada Real-Time Kernel Project

Accession For	
NTIS - CRA&I	<input checked="" type="checkbox"/>
DTIC - TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability codes	
Dist	Availability or Special
A-1	



Approved for public release.  
Distribution unlimited.

**Software Engineering Institute**  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

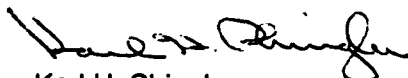
SEI Joint Program Office  
ESD/AVS  
Hanscom AFB, MA 01731

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 H. Shingler  
SEI Joint Program Office

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

Copyright © 1989 by 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

<b>I. 68020 to VMS Port</b>	<b>1</b>
1. Compiler Evaluation	2
1.1. Features Essential for Porting DARK	2
1.2. Compiler Dependencies	2
1.3. Machine Code Dependencies	2
2. General Issues	3
2.1. Hardware_Interface	3
2.2. Low-Level Hardware Interface	4
2.2.1. My_network_address	4
2.2.2. Is_Kproc	4
2.2.3. Is_Nproc	5
2.2.4. P	5
2.2.5. V	5
2.2.6. Set_interrupt_priority	5
2.2.7. Reset_interrupt_priority	5
2.2.8. Low_Level_Hardware Package Body	5
3. Processor Management	6
4. Process Management	7
4.1. Process Encapsulation	7
4.1.1. Dummy_call_frame	7
4.1.2. Indirect_call	8
4.2. Context Switcher	8
4.2.1. Save_context	9
4.2.2. Resume_process	9
4.2.3. Switch_processes	9
4.2.4. VMS Version	9
5. Semaphore Management	10
6. Schedule Management	11
7. Communication Management	12
7.1. Nproc	12
7.2. Datagram_management	12
7.3. Datagram_globals	13
7.4. Bus_io	13
7.4.1. Initialize	13
7.4.2. Receive_dg_interrupt_handler	13
8. Interrupt Management	14
8.1. Fast Interrupts	14
8.2. Slow Interrupts	15
9. Time Management	17

9.1. Generic_kernel_time	17
9.2. VMS_services	18
9.3. Timers	18
9.3.1. Set_timer	19
9.3.2. Cancel_timer	19
9.4. Clock	19
9.4.1. Start	19
9.4.2. Stop	19
9.4.3. Get_time	19
9.4.4. Adjust_elapsed_time	19
9.4.5. Adjust_epoch_time	19
10. Alarm Management	20
11. Tool Interface	21
12. Other Issues	22
12.1. AST Encapsulation	23
13. Portability Summary	25
<b>II. Kernel Extensions</b>	<b>26</b>
1. Dynamic Reconfiguration	27
1.1. Description	27
1.2. Rationale	27
1.3. Response	28
2. Fault Tolerance	29
2.1. Description	29
2.2. Rationale	29
2.3. Response	30
3. Tightly-Coupled Systems	31
3.1. Description	31
3.2. Rationale	32
3.3. Response	32
4. Code Overlays	33
4.1. Description	33
4.2. Rationale	33
4.3. Response	33
5. Communication Enhancements	34
5.1. Description	34
5.2. Rationale	34
5.3. Response	35
6. Duplicate Load Images	36
6.1. Description	36
6.2. Rationale	36

6.3. Response	36
7. Single Processor Mode	37
7.1. Description	37
7.2. Rationale	37
7.3. Response	37
8. Initialization Control	38
8.1. Description	38
8.2. Rationale	38
8.3. Response	38
9. Processor Global Priority Adjustment	39
9.1. Description	39
9.2. Rationale	39
9.3. Response	39
10. Fast Mode Change	40
10.1. Description	40
10.2. Rationale	40
10.3. Response	40
11. Memory Management	41
11.1. Description	41
11.2. Rationale	41
11.3. Response	41
12. Extensibility Summary	42
<b>Appendix A. Machine and Compiler Dependencies</b>	<b>43</b>
A.1. General 68020 Dependencies	44
A.2. Detailed 68020 Assembly Language Dependencies	46
A.3. General VAX/VMS Dependencies	47
A.4. Detailed VAX Assembly Language Dependencies	48

## List of Figures

<b>Figure 1:</b> SETIMR Interface Before Modifications	24
<b>Figure 2:</b> SETIMR Interface After Modifications	24

# DARK Porting and Extension Guide

**Abstract:** This document describes the modifications made to the Distributed Ada Real-Time Kernel (DARK) software when porting it from its original execution environment, the 68020-based testbed built at the Software Engineering Institute, to a VAX/VMS system. This document also contains information about logical extensions to the Kernel, and the impacts thereof, should the Kernel be used in operational systems.

## I. 68020 to VMS Port

Part I of this document is intended to give potential Kernel users information on the amount and type of work required to port the Kernel to another architecture. This part tries to present general issues and difficulties involved in rehosting the Kernel. All of the material contained in Part I is based on porting the Kernel from a bareboard 68020 target to a VAX/VMS target.<sup>1</sup>

There were several reasons for porting the Kernel to VMS:

1. To validate that the Kernel was indeed portable with a minimal amount of effort.
2. To provide the Kernel on a readily available platform.
3. To provide the Kernel on a platform from which other ports can more easily be launched.
4. To provide a host development version of the Kernel.

These goals dictated the direction of the port. They led to a Kernel that is syntactically equivalent to the 68020 version (i.e., any code that compiles with the 68020 version of the Kernel will compile with the VMS version) but not necessarily semantically equivalent (i.e., not all the operations are relevant on VMS nor does the VMS version obey the timing constraints defined in [KFD 89]).

After an overview of the compiler-related issues, Part I of this document will be structured along the lines of the *Kernel Facilities Definition* [KFD 89], following the breakdown of functional areas therein.

---

<sup>1</sup>An overview of the VMS version can be found in Appendix I of [KAM 89].



# 1. Compiler Evaluation

Before beginning the port of the Kernel to a new Ada compiler, it is necessary to examine some features of the compiler and the generated code. These features fall into three areas:

- Features essential for porting the Kernel.
- Adaptations required due to compiler dependencies.
- Adaptations required to interface to machine code.

## 1.1. Features Essential for Porting DARK

These are also described in full in Appendix J of [KAM 89]. In brief, the compiler must support:

- Chapter 13 of [ALRM 83].
- The use of generics to achieve "conditional compilation."

## 1.2. Compiler Dependencies

These are described in full in Appendix J of [KAM 89]. The major dependencies that must be taken into account are:

- The definition of basic data types in package *system*.
- Compiler dependencies in package elaboration order.
- Any special pragmata required to interface to machine code.

## 1.3. Machine Code Dependencies

These are described in full in Appendix K of the [KAM 89]. The porter must determine:

- How Ada interfaces with machine code.
- What parameter passing conventions are used.
- How the stack is organized.
- How exception handling is performed.

These conventions must be studied in detail, because the lower levels of the Kernel create and modify stacks, perform process context switches, encapsulate interrupts, and in general require exact knowledge of the target machine and the Ada compiler's conventions for code generation.

## 2. General Issues

### 2.1. Hardware\_Interface

The Kernel package *hardware\_interface* provides an interface to compiler-specific primitive types. Within the Kernel itself, there are no references to the Ada predefined types; all references to primitive types use names declared in package *hardware\_interface*. By doing this, certain implementation-dependent details are abstracted from the Kernel in a uniform manner.

In addition, the package *hardware\_interface* provides a similar interface to the Ada package *system*; all references to standard and implementation-specific names are encapsulated within package *hardware\_interface*.

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

The VMS version is functionally equivalent to the 68020 version, with the following changes:

1. A new type was introduced: *hw\_long\_unsigned*. This type is based on the VMS-specific type *system.unsigned\_longword*. *Hw\_long\_unsigned* provides a full 32-bit representation of an unsigned integer value.
2. The ordering of the names of bits in type *hw\_bits8* has been reversed, as VMS numbers bits from right-to-left. In this case, both the VAX Ada compiler and VMS have a consistent view of how bits are numbered, whereas the TeleSoft compiler had a different view from the M68020 documentation.
3. The type *hw\_short\_integer* is made available. VMS permits the declaration of 8-bit integer values; TeleSoft did not. The declaration of *hw\_short\_integer* did exist in the original package *hardware\_interface*, but was commented out. However, in order to maintain as much compatibility as possible between the 68020 and VMS versions of the Kernel, type *hw\_short\_integer* is currently *not* used anywhere in the VMS version.

The following should be considered when porting the hardware interface capability to any new target:

1. There are currently no test cases that prove that the compiler does, in fact, provide *exactly* the number of bits defined by the representation specifications for each of the types in package *hardware\_interface*. When writing test cases, note that the *Ada Language Reference Manual* states that when 'size' is "applied to an object, it [*'size*] yields the number of bits allocated to hold the

object." However, when *'size* is "applied to a type or subtype, it yields the minimum number of bits ... " (Section 13.7.2(5)).

2. The *Ada Language Reference Manual* clearly states, in Section 13.2(5), that when *'size* is used as the *attribute* in a *length\_clause*, "the value of the expression specifies an upper bound for the number of bits to be allocated to objects of the type or first named subtype T." Note that, if the compiler adds on any "hidden descriptors" to the primitive value, storage for them must also be included in the *simple\_expression* used in the *length\_clause* for the types declared in package *hardware\_interface*.
3. Finally, the concept of system addresses (implemented in Ada via type *system.address*) must be considered. Package *hardware\_interface* exports a type *hw\_address*, which assumes that its value actually points to the first storage unit of the true value and not to any descriptors or supplemental address values. The *Ada Language Reference Manual* states that "*X'address* yields the address of the first of the storage units allocated to X" (Section 13.7.2(3)). Note that this first storage unit may be for a descriptor, a supplemental address, or the value itself.

## 2.2. Low-Level Hardware Interface

Package *low\_level\_hardware* was originally created to hide details of the 68020 target hardware from the Kernel; it still performs that function on the VMS target, but with reduced functionality.

### 2.2.1. My\_network\_address

This function returns the "node"<sup>2</sup> number on which the code is executing. Under VMS, this "node" identification is supplied by the user to the Kernel at initialization time (on the 68020, it is read directly from the hardware).

### 2.2.2. Is\_Kproc

This function returns a Boolean value:

- True, if the "node" is a Kernel processor (or Kproc).
- False, if the "node" is a Network processor (or Nproc).

Under VMS, this function always returns true, since all "nodes" are Kprocs.

---

<sup>2</sup>Node is always quoted when referring to a logical node versus a real, physically distinct processor, as in the 68020 version.

### **2.2.3. Is\_Nproc**

This function returns a Boolean value:

- True, if the "node" is a Network processor.
- False, if the "node" is a Kernel processor.

Under VMS, this function always returns false, since there are no Nprocs.

### **2.2.4. P**

This procedure is used to synchronize access to datagrams in shared memory on the 68020 target. With only one processor per "node" on VMS, there is no need to protect the datagram queue from concurrent access by another physical processor (although it is still protected against concurrent access by processes and interrupt handlers). Since it is not needed under VMS, the body is null.

### **2.2.5. V**

This procedure is used to synchronize access to datagrams in shared memory on the 68020 target. Because it is not needed under VMS, for the same reasons noted in Section 2.2.4, the body is null.

### **2.2.6. Set\_interrupt\_priority**

This function is used to mark the start of an atomic (or uninterruptable) region. Under VMS, hardware interrupts are replaced by VMS ASTs. This function is implemented using the VMS system service \$SETAST to disable the delivery of ASTs during atomic regions. This function still returns the previous interrupt level (i.e., ASTs enabled or disabled).

Since there are no priority levels associated with controlling AST delivery, the type *interrupt\_priority* is now a subtype of Boolean, where:

- True, enable AST delivery.
- False, disable AST delivery.

### **2.2.7. Reset\_interrupt\_priority**

This function is used to mark the end of an atomic (or uninterruptable) region. It operates as the inverse of *set\_interrupt\_priority* and is also implemented using \$SETAST.

### **2.2.8. Low\_Level\_Hardware Package Body**

The Kernel code that queries the user for the "node" number is located here. This ensures that the "node" number is set during elaboration, before any calls can be made to *low\_level\_hardware* operations.

### 3. Processor Management

*Generic\_processor\_management*, *processor\_management*, *network\_configuration\_table*, and *make\_nct* are unchanged from the 68020 version.<sup>3</sup>

---

<sup>3</sup>In the VMS version, the "nodes" can be started in any order and the network initialization protocol will function properly. This is a side effect of the queue discipline built into the VMS system services.

## 4. Process Management

### 4.1. Process Encapsulation

A Kernel process is an Ada procedure with neither parameters nor result. It executes in parallel with other processes under the control of the DARK Scheduler.

To create this effect, the process must be encapsulated in a manner that:

- Permits the Ada procedure to execute in its own environment.
- Executes the procedure in that environment.
- Properly handles exits from the procedure, either normal or abnormal (via exception), and cleanly terminates the Kernel process.

The machine-dependent parts of this code are the procedures *dummy\_call\_frame* and *indirect\_call*.

#### 4.1.1. Dummy\_call\_frame

This procedure is used to initialize a new stack that will become the stack of a Kernel process. To do this, *dummy\_call\_frame* must create, on the new process stack, a data structure that looks exactly like an Ada call frame, and must save in the process *context\_save\_area* a machine state that, when restored, will cause the process to begin execution in what it believes to be a valid manner.

The illusion *dummy\_call\_frame* is required to fabricate is this: procedure *dummy\_call\_frame* is executing as a parallel process and has just returned from a call of *low\_level\_process\_encapsulation*. Within a guarded region, *dummy\_call\_frame* invokes the Ada procedure that is the Kernel process.

However the Kernel process terminates – by a return or by exception propagation – the process encapsulation will itself never execute a return, but will destroy itself by a call of *die*. Hence, there is no need to fabricate on the new stack anything above the call frame of *dummy\_call\_frame*, since it will never be needed.

The new stack data structure that was created must allow correct access to:

- The local variables of *dummy\_call\_frame*.
- The parameters of *dummy\_call\_frame*.
- All Ada global variables.
- All visible Ada subprograms.

This requires appropriate values to be computed and saved for the: stack pointer, frame pointer, and argument pointer. It may also require other base registers to be set up. Note that no access is permitted from a Kernel process to the local variables of any lexically enclosing block, so there is no need to fabricate a display or static chain.

Finally, it may be necessary to take action to prevent Ada runtime diagnostic code, or an Ada debugger, from "walking" up the new stack beyond the top. This action is highly dependent on the internal details of the Ada run time and tools; however, it is usually safe (and expensive) to link the call frame at the top of the new stack back into the normal call chain (the chain that led to the call of *dummy\_call\_frame*) at a point that can never go out of scope (e.g., to the top-level call by which the Ada run time first invoked the Ada procedure, the DARK Main Unit).

The VMS port required rewriting the body of *low\_level\_process\_encapsulation* in machine code. In addition, the pragma *import\_procedure* had to be added to the specification of *low\_level\_process\_encapsulation*, which VAX Ada requires if a body is to be in machine code.

#### 4.1.2. Indirect\_call

This procedure takes as its parameter the address of an Ada procedure and calls it. It implements, therefore, the parametric procedure of languages such as Algol-60 or Modula-2, but without safe type checking.

*Indirect\_call* is required to do the following:

- Invoke its parameter as an Ada subprogram with neither parameters nor result.
- After a normal return from that subprogram, take a normal return to the caller.
- After an abnormal exit from that subprogram, propagate the exception to the caller.

The called subprogram must be able to access all global variables and any visible Ada subprograms. It is not allowed to access any local variables of a lexically enclosing block. The body of *indirect\_call* was rewritten in machine code. In addition, the pragma *import\_procedure* had to be added for the same reason as given above.

#### 4.2. Context Switcher

This package contains three procedures: *save\_context*, *resume\_process*, and *switch\_processes*.

The Ada part of *context\_switcher* makes all necessary changes in the Kernel data structures, and it is portable. The machine-dependent part is in *low\_level\_context\_switcher*, which performs the actual changes to machine state.

In all cases, the context to be saved and restored is the machine state that the conventions require to be preserved across a procedure call. Each process has the illusion that it made a simple call (e.g., a call to *switch\_processes*), which took a synchronous return back to the caller. In fact, there is a lapse of time between the call and the return, which the Kernel process does not perceive since the Kernel saved its state and subsequently restored it exactly.

### 4.2.1. Save\_context

This subprogram takes the address of a context save area as its parameter and saves in that context save area the current true machine state. *Save\_context* then returns. The effect is that, if the saved state were subsequently restored, a return would transfer control to exactly the point where *save\_context* was called.

### 4.2.2. Resume\_process

This subprogram does the opposite of *save\_context*. It takes the address of a context save area as its parameter and copies the contents of that context save area to the hardware, thus changing, for example, the general registers, the program counter, the stack pointers, and anything else appropriate. It then returns, not to the caller since the program counter and stack have been changed, but rather to the process whose state has been restored.

In addition, *resume\_process* takes as a second parameter the address of the end of the new stack – the one that will be restored. In most Ada compiler implementations, this value must be written into a variable in the Ada run time – the variable against which the Ada run time performs the stack overflow (*Storage\_Error*) check. It may not be easy to find this variable.

As an implementation note, observe that the process of restoring context will almost certainly render invisible the argument list with which the procedure was called (e.g., by overwriting the frame pointer). Care must be taken, therefore, not to perform this part of the restore until the argument list is no longer needed.

### 4.2.3. Switch\_processes

This performs the two actions *save\_context* and *resume\_process* in succession. It saves the current true machine state in the area designated by its first parameter, restores the state from the area designated by the second parameter, and if necessary resets the Ada runtime stack limit variable from its third parameter.

Exactly the same implementation considerations apply as in Sections 4.2.1 and 4.2.2.

### 4.2.4. VMS Version

The VMS versions were written in machine code. They save and restore the following hardware registers:

- R2 through R11
- AP
- FP

They do not save and restore R0 and R1 because the VMS calling conventions do not require these registers to be saved across a procedure call. They do not save SP and PC because the RET at the end of each routine restores the correct values from the call frame addressed by FP.



## 5. Semaphore Management

*Generic\_semaphore\_management* and *semaphore\_management* are unchanged from the 68020 version.

## 6. Schedule Management

The only change to package *scheduler* code is addition of the line:

```
pragma export_procedure (schedule_ih);
```

to the specification. This change is required by VAX Ada to make the entry visible to assembly language code.

## 7. Communication Management

*Generic\_communication\_management* and *communication\_management* are unchanged from the 68020 version.

The impact of porting the Kernel to VMS occurred within the Kernel packages that implement the node-to-node communication.

### 7.1. Nproc

The Nproc and its associated I/O drivers were deleted. The Nproc is replaced by VMS services that use shared mailboxes and ASTs (see Appendix I of [KAM 89]).

### 7.2. Datagram\_management

The abstraction of datagrams and the manipulation of queues of datagrams is maintained in the VMS version of the Kernel (this is described in detail in the [KAM 89]). Several changes were made to *datagram\_management*:

- All datagram queue initialization was combined into one initialization procedure, *kproc\_initialize*. This allows all the Kproc code dependent on the *datagram\_management* package to remain unchanged.
- The number of datagrams placed in the initial free pool by *kproc\_initialize* was reduced. On the 68020, 1 megabyte of shared memory is reserved exclusively for the datagram pool. This memory is allocated until a storage error is generated (i.e., until all available shared memory is exhausted). Clearly this is not useful in the VMS environment. *Kproc\_initialize* allocates 100 datagrams for each size (small, large, and kernel).
- The use of unchecked programming in the datagram queue initialization was removed. This caused problems at run time, since the VAX Ada run time performs garbage collection at the end of each local scope and the use of unchecked conversion caused the garbage collector to lose track of the objects and allowed them to be reclaimed when they were actually being used elsewhere in the code.
- *Alloc\_dg* has been modified to allocate additional datagrams via the Ada allocator, if the initial free pool is exhausted. These datagrams are no different from the initial allocation and are returned to the free pool when no longer needed.

### 7.3. Datagram\_globals

The removal of unchecked programming from *datagram\_management* required that a discriminant be added to *datagram* type declaration. The modified declaration is:

```
type datagram (size: buffer_range) is
  record
    local      :      local_optimization_record ;
    header     :      datagram_header;
    buffer     :      data_buffer(1 .. size);
  end record;
```

### 7.4. Bus\_io

The low-level communication in *bus\_io* was minimally affected by the replacement of the Nproc by VMS. Since the receipt of messages is interrupt driven in the 68020 version, the conversion from interrupts to ASTs was simple.

#### 7.4.1. Initialize

*Initialize* was changed as follows:

1. The references to 68020 interrupts were excised from the code.
2. At execution time, a VMS mailbox is created to receive all incoming message traffic (for this "node").
3. At execution time, a connection is made to the mailboxes on all the other "nodes" (both Kernel and non-Kernel).<sup>4</sup>
4. At execution time, a \$QIO read with AST notice request is issued on the local mailbox telling VMS that as soon as any entity writes a message to the mailbox: read the data from the message and generate an AST to the *receive\_dg\_interrupt\_handler*.

Once *initialize* executes, the node has a connection to all other nodes and is prepared to field any incoming messages.

#### 7.4.2. Receive\_dg\_interrupt\_handler

The conversion of the 68020 interrupt handler to a VMS AST handler was also straightforward, the only differences being:

1. The handler is entered with a message already copied into a datagram.
2. A \$QIO read with an AST notice request is issued, in order to receive an AST when the next message is written to the mailbox, and the handler exits.

Thus, VMS performs exactly those services that the 68020 Nproc performed. It receives messages from a remote source, queues incoming messages until they are read, and generates "interrupts" when complete messages are received.

---

<sup>4</sup>Each mailbox is qualified by the VMS user\_id of the person executing the image. Thus, although multiple users can simultaneously run tests, any given user can only be executing one test at a time (per VMS machine).

## 8. Interrupt Management

The Kernel convention is that the application code may bind handlers to legal interrupts, using the procedure *bind\_interrupt\_handler*. The Kernel itself uses the same mechanism for the interrupts it uses.

An interrupt handler is an Ada procedure with neither parameters nor result. It must be able to access all global variables and any visible Ada subprograms. It is not allowed to access any local variables of a lexically enclosing block.

The machine-dependent part of the interrupt handling capability is found in package *low\_level\_interrupt\_management* and consists of two procedures: *bind\_fast\_handler* and *bind\_slow\_handler*.

The basic action of both procedures is the same: set up a structure that, when the interrupt occurs, will cause the Ada handler to be invoked in a valid environment. This is usually done by connecting the hardware interrupt to a machine code encapsulation. This encapsulation invokes the Ada handler, in the same way as *indirect\_call*, but wraps around this invocation any essential machine-dependent actions required to save and restore the state of the interrupted process and to set up the proper environment of the handler.

The encapsulation must also maintain the value of the variable *interrupt\_nesting*, which is what tells the rest of the Kernel that the currently executing code is part of an interrupt handler. This information is needed, for example, to allow a blocking primitive to reject a call from a handler.

In addition, any necessary critical regions must be opened and closed within the encapsulation to prevent an immediately succeeding interrupt from damaging machine state.

### 8.1. Fast Interrupts

A "fast" interrupt is one that always returns to the interrupted process. This model is usually supported well by the target machine or operating system. The basic actions of the encapsulation are:

- Save any volatile hardware state.
- Set the hardware state to be the correct environment for the handler.
- Invoke the Ada procedure that is the handler.
- After a normal return from the handler, restore the volatile hardware state, and execute the appropriate instructions to return from interrupt and resume the interrupted process.
- After an abnormal exit from the handler, recover control, discard the exception, restore the volatile hardware state, and execute the appropriate instructions to return from the interrupt handler and resume the interrupted process.

On VMS, the fast interrupts follow exactly the conventional response to an AST: the encapsulation merely calls the true handler and then performs a return.

## 8.2. Slow Interrupts

A "slow" interrupt is one that, after being handled, exits to the Kernel Scheduler, which may then choose either to resume the interrupted process or to suspend it and resume another. Many machines and operating systems are built on the assumption that the application programmer will not want to do this, so providing this capability in the Kernel can be difficult.

The Ada handler itself should be invoked exactly as for a fast interrupt. However, after the return (or exit), the slow interrupt encapsulation must do the following:

- Ensure that the saved volatile state of the interrupted process is in a very safe place.
- Change the machine state so that a return-from-interrupt can be executed. The encapsulation must:
  - Return control to the encapsulation code.
  - Place the processor back in normal state.
  - Be able to execute further machine code normally.

The return-from-interrupt will therefore fool the underlying system into returning not to the interrupted process, but rather back to the interrupt encapsulation code.

It is now necessary to invoke the Scheduler, which is done by a call of the parameterless procedure *Scheduler.schedule\_ih*.

The illusion that has been fabricated is that the interrupted process made a synchronous call of the encapsulation code, which then made a synchronous call of the Scheduler. The Scheduler will perform normal context saves and restores, and will eventually resume the interrupted process. Typically, a normal context switch does not save the entire the machine state, but rather only the state that must remain valid across a procedure call. The interrupt encapsulation must therefore save and restore, on behalf of the interrupted process, all other necessary state information.

When the Kernel Scheduler resumes the interrupted process, the resumption point is of course in the interrupt encapsulation, immediately after the call of *Scheduler.schedule\_ih*. The encapsulation must therefore restore all volatile state information and issue the appropriate return instruction to get back to the interrupted process. Note two things:

- The machine state must be restored exactly as it was at the moment of interruption. This includes things like coprocessor state, processor status words, condition codes, and the like.
- The volatile state must be saved in a process-specific data area, since it must be preserved across possibly many context switches and interrupts. The process stack is usually a reasonable place for this data.

Slow interrupts were very hard to implement in VMS, and required detailed knowledge of the AST mechanism and great care in perverting it from its intended purpose. Further information can be found in the documentation of the relevant module.

## 9. Time Management

The time functions were implemented using a signed 64-bit representation, as in the 68020 version. In the 68020 version, this 64-bit quantity was represented as two signed 32-bit integers, since the TeleSoft compiler did not support true unsigned variables. On VMS, it was possible to actually specify a 32-bit unsigned quantity, and time was stored as a 64-bit quantity, manifested as a 32-bit unsigned low longword and a 32-bit signed high longword. To maintain machine independence throughout the rest of the Kernel, the *kernel\_time* package required a new type declaration called *hw\_unsigned\_longword*, which was defined in *hardware\_interface* (see Section 2.1).

Porting the time-handling features to VMS required changing *generic\_kernel\_time* and creating *vms\_services*.

### 9.1. Generic\_kernel\_time

In the 68020 version of package *generic\_kernel\_time*, time was represented as a 64-bit signed integer quantity with a resolution of 1 $\mu$ s and a range of roughly 150,000 years. The choice of resolution was based on the accuracy of the hardware timers available on the parallel I/O controllers. When the Kernel was ported to VMS, it was decided to utilize the full accuracy of the clocks available through VMS. Hence, although time is still represented as a 64-bit signed value, the resolution of the Kernel clock under VMS was changed to 100ns, with a corresponding change in range to roughly 15,000 years.<sup>5</sup> Since the bit patterns of *kernel\_time* and VMS time are almost equivalent, converting between *kernel\_time* and VMS time is often a null conversion with a simple overflow check.<sup>6</sup>

Finally, each of the routines for performing operations on type *kernel\_time* were re-written. Originally written in 68020 assembly language, the routines to add, subtract, negate, multiply, divide, and compare *kernel\_time*, as well as those to convert integral numbers of seconds, milliseconds, and microseconds into *kernel\_time*, were rewritten in VAX assembly language.<sup>7</sup>

---

<sup>5</sup>This reduction in range is not viewed as a problem for most embedded applications.

<sup>6</sup>An alternative implementation would have been to keep the original resolution and range of the Kernel clock, and use the conversion functions in *VMS\_Services* to perform the multiplication and division by 10 to convert between the *kernel\_time* and the VMS time.

<sup>7</sup>Although technically the 64-bit arithmetic routines could have been written in Ada, they would have been far more cumbersome than the resulting assembly language versions.



## 9.2. VMS\_services

The package *vms\_services* was needed to convert between the Kernel representation of time and the VMS representation of time. The Kernel exports two time types (*epoch\_time* and *elapsed\_time*), and maintains all time values internally as a single type (*kernel\_time*). The VMS version of the Kernel needed to preserve these types, but also had to interface with the underlying notion of VMS time, which was represented as either an *absolute* time or a *delta* time (see the *VMS System Services Manual, Volume 8D*, for more details). The new functions *to\_kernel\_time*, *to\_vms\_absolute\_time*, and *to\_vms\_delta\_time* perform these conversions.

The conversion functions were complicated by the underlying definition of base time. The Kernel defines base time to be Julian day 1 (that is, 12:00 on 1 January, 4713 BCE, as defined in the [KFD 89], Section 12.1.14), while VMS uses the Smithsonian base date (that is, 00:00 on 17 November, 1858 CE) as its base time. When converting from *epoch\_time* to VMS absolute time, it is necessary to check that the *epoch\_time* is later than the Smithsonian base date. Furthermore, while the Kernel allows an *elapsed\_time* to have a negative value (representing a time already passed), VMS allows delta times to only have positive values (representing only times in the future). When converting to a VMS delta time, therefore, a check must be made to ensure that the time has not already passed. If it has, then the time is mapped to zero delta time; achieving the desired result, i.e., immediate processing of the activity.

## 9.3. Timers

Use of the hardware timers is much simpler using VMS. Package *timer\_controller* written for the 68020 version of the Kernel was deleted in the VMS version and its functionality subsumed directly into the body of package *time\_keeper*.

VMS supplies two services: *\$SETIMR* and *\$CANTIM* (described in detail in *VMS System Services, Volume 8D*). These services, along with ASTs, provide all the tools needed to implement both the package *time\_keeper* and the Kernel's time event queue. The conversion to VMS affected the body of *time\_keeper* as follows:

- All code that initialized timers was removed from *initialize*.
- All calls to *timer\_controller.set\_timer* were replaced by calls to the internal procedure *set\_timer*.
- All calls to *timer\_controller.cancel\_timer* were replaced by calls to the internal procedure *cancel\_timer*.

### 9.3.1. Set\_timer

The VMS system service \$SETIMR allows the user to set up an AST to occur at some future time. When this future time arrives, an AST is generated, and the AST handler specified in the call to \$SETIMR is invoked. This handler then performs the necessary processing.

All that *set\_timer* does is:

1. Convert the *epoch\_time* of the event into a VMS delta time (since all Kernel event times are maintained in *epoch\_time*).
2. Set up the timer to invoke the *ast\_encapsulation* procedure (described in Section 12.1) when the AST occurs.

### 9.3.2. Cancel\_timer

Since the Kernel has only one timer active at any given time, a simple call to \$CANTIM terminates the pending timer without generating an AST.

## 9.4. Clock

The VMS clock is used exactly as the 68020 hardware clock is used. Here, the implementation is much simpler since it is built on the clock services provided by VMS.

### 9.4.1. Start

This operation is really a no-op. Since the VMS system clock is never stopped, it is never started via this primitive. The sole action of this primitive is to log the *base\_epoch\_time* for the user.

### 9.4.2. Stop

This operation is a no-op.

### 9.4.3. Get\_time

This operation is implemented using \$GETTIM and *vms\_services.to\_kernel\_time*. The VMS system clock is read, converted to *kernel\_time*, and returned to the user.

### 9.4.4. Adjust\_elapsed\_time

This operation is implemented as a no-op. It is certainly possible to implement this operation, but it was not deemed to be a useful operation for a host development version.

### 9.4.5. Adjust\_epoch\_time

This operation is implemented as a no-op, for the same reasons as *adjust\_elapsed\_time* above.

## 10. Alarm Management

*Generic\_alarm\_management* and *alarm\_management* are unchanged from the 68020 version.

## 11. Tool Interface

*Generic\_tool\_interface*, *tool\_interface*, and *generic\_tool\_interface\_output* are unchanged from the 68020 version.

## 12. Other Issues

This chapter summarizes other issues encountered during the port to VMS.

- Most of the 68020 target-dependent packages have no counterpart in the VMS version and were deleted (see [KAM 89] for more details on these packages):
  - *interprocessor\_interrupts*
  - *kernel\_interrupt\_management*
  - *memory\_addresses*
  - *mz8305\_definitions*
  - *mvme133a\_definitions*
  - *parallel\_io\_controller*
- *Scc\_porta* had all of its functionality removed in order to maintain consistency with the 68020 version in the higher levels of the Kernel.
- Hidden compiler dependencies:<sup>8</sup>
  - Use of null array slices: In a number of places, the Kernel code relies on the use of null array slices to short circuit array assignment statements. It turns out that the TeleSoft compiler generates code that terminates the assignments as soon as a null slice is detected. On the other hand, the VAX Ada compiler generated code to evaluate all the parts of the statement before testing for the null slice. In some (expected) situations, this generates an error under VMS while executing under Telesoft.
  - Garbage collection in VAX Ada: Another dependency was inadvertent. The datagram queues are created and the linked lists built by a procedure in the Datagram\_Management package. Unfortunately, while the queue control structures are local to the package body, the queues themselves are local to the procedure body. Since TeleSoft does not do garbage collection on locally allocated objects and VAX Ada does, this led to useless datagram queues in the VMS version.
  - Elaboration problems: Instantiations that elaborate correctly under TeleSoft, all need *pragma elaborate* on the corresponding generic to elaborate correctly under VAX Ada. The reason for this problem is unknown. This is discussed in more detail in [KUM 89].
  - Hand instantiations: *Generic\_kernel\_time* had to be hand-instantiated due to a VAX Ada restriction that does not allow a generic package to have a machine-code body.
- The VMS version of the Kernel can be executed by any user with normal VMS privileges.

---

<sup>8</sup>These problems occurred only in the VMS version of the Kernel and each one has been corrected.

## 12.1. AST Encapsulation

Under VMS, ASTs are normally bound to Ada task entries. Since tasks are not allowed in a Kernel process, the default Ada binding provided by VMS had to be changed. As with 68020 interrupts, the Kernel binds procedures to ASTs. Also, under VMS, ASTs always return to the interrupted context; however, with the Kernel, ASTs may result in the need to switch context to another process. To avoid these difficulties, VMS must be fooled. This is done by two steps:

1. Modifying the Ada signature of the system call that generates the AST.
2. Using the VMS-specific package *ast\_manager*.

The first step decouples ASTs from Ada task entries. The default signature provided by package *starlet* to the Ada code is modified to create a new overloading for the entry. For example, the system service \$SETIMR (shown in Figure 1) takes as a parameter the name of an Ada task entry (ASTADR) to invoke when the timer expires. The type *ast\_handler* can only be generated as an attribute of an Ada task entry. An examination of the system service \$SETIMR in *VAX VMS System Services, Volume 8D*, shows that this parameter is simply the address of a procedure to execute – exactly what is required for the Kernel. Therefore, to remove the tie to Ada tasking, the specification for \$SETIMR is modified as shown in Figure 2.<sup>9</sup> The differences are highlighted by uppercase.

Once a proper signature has been developed, the second step must be tackled. The new signature allows a procedure to be called in response to an AST, but it does not allow the Kernel to perform a context switch as a result of an AST. To do this requires placing a Kernel layer between the AST handler and VMS. This layer is *ast\_manager*. Instead of binding the AST directly to the desired AST handler, the AST is bound to *ast\_handler\_encapsulation* (ASTADR in Figure 2), with the parameter being the address of the actual AST handler (REQIDT in Figure 2). Thus, when the AST occurs, *ast\_handler\_encapsulation* is invoked. It immediately calls the true handler. When the handler finishes and exits normally, then *ast\_handler\_encapsulation* fools VMS into thinking the AST has completed, at which point it can safely call *scheduler.schedule\_ih* to determine if a context switch is needed.

---

<sup>9</sup>These signatures are readily available in the file ADA\$PREDEFINED:STARLET\_ADC.

```

procedure set_timer (
  status : out cond_value_type;           -- return value
  efn    : in starlet.ef_number_type    :=
  starlet.ef_number_zero;
  daytim : in starlet.date_time_type;
  ASTADR : IN  AST_HANDLER              := NO_AST_HANDLER;
  REQIDT : IN  USER_ARG_TYPE           := USER_ARG_ZERO);

pragma interface (external, set_timer);

pragma import_valued_procedure (
  set_timer, "sys$setimr",
  (cond_value_type,
   starlet.ef_number_type,
   starlet.date_time_type,
   AST_HANDLER,
   USER_ARG_TYPE),
  (value,
   value,
   reference,
   value,
   value));

```

Figure 1: SETIMR Interface Before Modifications

```

procedure set_timer (
  status : out cond_value_type;           -- return value
  efn    : in starlet.ef_number_type    :=
  starlet.ef_number_zero;
  daytim : in starlet.date_time_type;
  ASTADR : IN  SYSTEM.ADDRESS;
  REQIDT : IN  HW_ADDRESS);

pragma interface (external, set_timer);

pragma import_valued_procedure (
  set_timer, "sys$setimr",
  (cond_value_type,
   starlet.ef_number_type,
   starlet.date_time_type,
   SYSTEM.ADDRESS,
   HW_ADDRESS),
  (value,
   value,
   reference,
   value,
   value));

```

Figure 2: SETIMR Interface After Modifications

## 13. Portability Summary

Aside from providing an additional artifact, the VMS port was intended to verify that the Kernel design and implementation is indeed portable. This was confirmed in a number of ways:

- The machine- and compiler-dependent packages, identified in [KAM 89], were the only dependencies and were thoroughly isolated from the rest of the Kernel.
- All the user-visible packages and package bodies were ported without modification.
- The internal Kernel abstractions were the correct ones, only two new VAX-specific packages were required for the VMS version.
- The abstractions in packages *datagram\_globals*, *datagram\_management*, and *bus\_io* present to the Kernel allowed the entire network subsystem to be replaced by VMS without impacting any Kernel code.
- The primary difficulties in the VMS port resided in exactly the same areas as in the initial 68020 development:
  - Implementation of the context switch.
  - Interfacing to interrupts (or ASTs in the VMS case).
  - Context switching as a result of an interrupt.

Clearly, in any port of the Kernel, these are the critical areas that must be addressed. What is significant, however, is that even these difficult parts were successfully ported.

The bottom line is the clearest evaluation of the Kernel's portability. Comparing the initial development effort to the porting effort:

- Initial: 12 labor-years of effort to design, document, and implement.
- Port: 8 labor-months of effort to implement and update the documentation (the requirements and design remained the same).

Although the VMS environment was much easier to work in than the 68020 environment, porting the Kernel required only a small fraction of the time the initial development required.



## II. Kernel Extensions

The intent of Part II of this document is to present an industry perspective on extensibility issues related to the Distributed Ada Real-time Kernel (DARK) developed at the Software Engineering Institute (SEI). The document is structured in the form of a dialogue, with the industry request and rationale first, followed by the SEI response.

Most of the industry material was provided by Westinghouse Corporation, whose invaluable assistance is gratefully acknowledged.

### Background

The DARK artifact developed at the SEI represents a proof of concept that, for various reasons, makes many simplifications to the problem of building distributed real-time systems in Ada. In order for DARK to be of more general use in the MCCR community, some of the restrictions it assumes (described in [KFD 89]) must be relaxed. Accordingly, this document presents, from an industry perspective, several recommendations on Kernel extensibility. Extensions to the Kernel include any enhancement, modification, or addition required to make the Kernel better suited to support distributed real-time processing and to fulfill mission-critical computer resource requirements.<sup>10</sup> The recommendations contained in Part II of this document are a consolidation of thoughts presented by many individuals in industry and the DARK project team and do not necessarily form an exhaustive list of required extensions. No attempt has been made to explain how the extensions can be accomplished and some extensions may prove to be solved through application code.

### Content

Each chapter in Part II contains information for a specific extension of the Kernel. The information includes a description of the extension, a rationale for why the extension is required or desired, and a brief response (for each of the major recommendations) in terms of the desirability, feasibility, and importance of the requested extension.<sup>11</sup> The recommendations are presented in relative order of importance, with the first recommendation being of highest priority. The final chapter contains some concluding remarks.

---

<sup>10</sup>See [Flynn 79], [Natarajan 87], and [Grover 85] for additional background information.

<sup>11</sup>The responses represent the technical opinion of the DARK project, and are not an official SEI position.

# 1. Dynamic Reconfiguration

## 1.1. Description

Dynamic reconfiguration is a rather broad area and is a complex issue to deal with in the Kernel, but certain capabilities are required in order for the MCCR community to adopt the Kernel and meet project requirements. Simply stated, dynamic reconfiguration is the ability to modify and extend a system while it is running.<sup>12</sup> This ability includes modifications and extensions to both hardware and software.

Dynamic reconfiguration of hardware will require a dynamic Network Configuration Table (NCT) that allows an application to change or add entries during execution. Since hardware redundancy can currently be built into a system and communicated to the Kernel, this ability has a low priority. However, the ability to replace hardware components, such as processor boards, without system shutdown will be useful and may require modifications to the NCT.

Dynamic reconfiguration of software, on the other hand, is an important area for the Kernel to implement, one that will greatly support fault tolerance. One capability desired is the ability to start-up and allow system communication within a redundant processor or a replaced processor while the rest of the system continues processing. Another capability is to allow the dynamic declaration and creation of processes. These capabilities may require a periodic executive routine instead of a terminating main unit to manage reconfiguration and may require the logging of valid system states in the form of critical data snapshots so that recovery can be accomplished after reconfiguration.

## 1.2. Rationale

Many industrial projects cannot use the Kernel and fulfill their requirements because of the static nature of the Kernel imposed by many of its restrictions. Many of these requirements deal with a system's ability to recover from failures while still maintaining whatever processing is possible in the degraded condition. For example, air traffic control systems require the ability to replace failed boards and start their execution while the system is running and processing data. Capabilities like this are part of the essence of mission-critical systems; if the system is not mission-critical, then it can be powered down and repaired without a catastrophic result. Since the Kernel is targeted to mission-critical systems, then a certain degree of dynamic reconfiguration should be implemented in order for these systems to meet their requirements.

---

<sup>12</sup>See [Kramer 85], pg. 424.

### 1.3. Response

This extension seems to be requested mainly as a way of implementing part of the next request, fault tolerance. Reconfiguration is a necessary part of any scheme for fault tolerance.

Much of the technical work is straightforward. Allowing the dynamic declaration and creation of processes is mostly a matter of removing error and consistency checks. Also, it would not be difficult to allow the NCT to be changed dynamically; the main task would be the design and implementation of the necessary protocol messages.

The major technical issue, however, is not that of updating the NCT. It is rather the issue of how process-local state can migrate along with the code of a process. Such state includes local data, claimed resources, and (especially) pending messages. If process migration is not required as part of the extension, then it appears much easier.

## 2. Fault Tolerance

### 2.1. Description

Fault tolerance is another broad and complex issue that overlaps with dynamic reconfiguration and a dynamic implementation rather than the static nature of the Kernel.<sup>13</sup> Fault tolerant computer systems are defined as "systems capable of recovering from failures of their hardware or software components to provide uninterrupted real-time service."<sup>14</sup> [KFD 89] specifies that the "Kernel does not implement fault tolerance, but it does detect the presence of certain [...] classes of faults" and that the "Kernel provides the capability for an application to build some rudimentary degree of fault tolerance."

Specific extensions to this approach are difficult to identify, but, the overall goal is to offer the mechanisms that will allow a system to function in the presence of failures and to recover from those failures. This goal implies that the Kernel is capable of detecting, identifying, and containing faults and correctly reporting them to the application. Also implied is that the Kernel offers the facilities to allow an application to assess the faults and recover from them. Some examples of capabilities desired include the Kernel's ability to detect and identify failed processors, devices, or communication links and to report them to the application; the Kernel's allowing reconfiguration and dynamic process creation and start-up; the Kernel's allowing code and data download and migration; and the Kernel's providing the applications programmer finer control over the level of fault tolerance and exception handling. These capabilities imply that the Kernel offers the necessary interfaces to the application so that the application can communicate the system status to the Kernel and control the recovery in the presence of failures.

### 2.2. Rationale

Fault tolerance is another issue that affects the Kernel's ability to meet the near-term needs of the MCCR community. Fault tolerance requirements are some of the most stringent requirements the MCCR community must meet and they can only be met by the combined capabilities of hardware and both system and application software. Therefore, in order for the MCCR community to adopt the Kernel and deal with such requirements, the Kernel should supply a greater degree of fault tolerance.

---

<sup>13</sup>See [Knight 87] and [Northcutt 87] for additional background information.

<sup>14</sup>See [Vick 84], pg. 437

### **2.3. Response**

This is a most desirable requirement and, if feasible, should be given the highest priority. Unfortunately, it is also by far the most difficult. The DARK project has always viewed this issue as too large to be addressed within the project's time and resource constraints.

## 3. Tightly-Coupled Systems

### 3.1. Description

The Kernel should be extended to support tightly-coupled systems which are systems that share a common backplane and utilize shared memory. There are certainly various degrees to which the Kernel can support such systems. The current extreme is that the Kernel, which is based on loosely-coupled systems, does not assume the inclusion of shared memory in a system and therefore does not offer any facilities to use it.<sup>15</sup> At the other extreme, the Kernel could absolutely rely on shared memory by making the Kernel's data structures shared entities so that processes can be allocated across processors, thereby, distributing the processing load in a processor-independent manner. The latter extreme prevents the Kernel from being used within a loosely-coupled system and greatly restricts the Kernel's portability. Since efficiency is lost in implementations that are too generic, the best solution may be to create two separate versions of the Kernel, one to support loosely-coupled systems and one to support tightly-coupled systems. For the short term, however, the current recommendation is a compromise of the two extremes.

The compromise suggested is in the areas of communication, interrupts, and semaphores. The Kernel design indicates that each process receives messages through a single message port that is contained in local memory and managed by the Kernel and that the Kernel has control of the underlying communication medium. Communication across processors is through this medium from Kernel to Kernel. In a tightly-coupled system, communication between processors is primarily through shared memory. This method of communication requires a greater degree of management than sending a datagram out over a communication medium. The recommendation in the area of communication is that the Kernel supports process communication through shared memory.

The Kernel assumes, according to the [KFD 89], that interrupts "are events local to a processor and cannot be directly handled or bound by processes running on a different processor." In a tightly-coupled system, this assumption is not necessarily true; interrupts can be global events and in many cases it is advantageous to make them so. By making interrupts global events, processors that are least busy or least critical can service the interrupts, thereby, creating efficiency through flexibility. The recommendation in the area of interrupts is that the Kernel allows and supports interrupts as global events.

The Kernel assumes, according to the [KFD 89], that a "semaphore is visible only on the processor on which it is declared, and therefore can be used only by processes local to the processor." The recommendation in the area of semaphores is that the Kernel extend the use of semaphores to include global resources.

---

<sup>15</sup>Nor does the Kernel interface with the use of shared memory by the application via typical Ada means.

## 3.2. Rationale

The Kernel is intended to meet the near-term needs of the mission-critical computer resource (MCCR) community who are involved with real-time, distributed Ada. This community implements their applications within both tightly-coupled and loosely-coupled environments depending upon specific needs and requirements. The Electronic Systems Group of Westinghouse utilizes both types of systems. Tightly-coupled systems are frequently used within ground-based radar systems where large amounts of data must remain accessible to all processes throughout the processing flow, thereby requiring shared memory and fast access to the shared memory. For the Kernel to meet its objective for the whole community, it should address the issues related to tightly-coupled systems and offer the implementors of such systems an alternative that may be more cost-effective and offer more user control than some of the existing products on the market.

## 3.3. Response

Technically, there are three aspects to implementation: communication via shared memory, shared semaphores, and global interrupts. The first is possible using the existing communication primitives, since one process can send to another a message that conveys access rights to a piece of shared memory. The Kernel does not prevent any process from acquiring shared memory, using it, and communicating its access rights to another process.

Shared semaphores can be implemented without major impact to the other parts of the Kernel. The data structure and access primitives would need to guard against true concurrent access from more than one processor, and the *Release* primitive would need to signal to the processor on which the next claimant for the semaphore was sited.

Without further detail, the feasibility of having interrupts made global across several processors cannot be assessed. At first sight, this seems a major change, since to perform the requested action – select a processor to respond to the interrupt dynamically on the basis of load – the several incarnations of the DARK scheduler on the several processors would have to communicate in ways never envisaged in their design.

A related issue in tightly-coupled, shared memory systems is that of dynamic load balancing among the processors. It may be possible to perform this automatically by moving the process run queue to shared memory and protecting it against concurrent access. In this situation, whenever the current running process blocks, the Scheduler on the processor would go to the common run queue and select the next process to execute.

## **4. Code Overlays**

### **4.1. Description**

Code overlays involve the swapping of executable code and/or data in and out of memory. In embedded systems, the overlay code is usually contained in read-only memory (ROM) or shared memory and loaded and executed when required. [KFD 89] indicates that the "Kernel operates under the restriction that all processes and all data are memory resident at all times" and that this "does not prohibit the application from building processes that can be rolled in and out of memory." This specification appears to allow the application code to perform code overlays on its own; however, since processes are created statically, the Kernel must maintain the data structures and other overhead for all processes at all times. This situation may not be acceptable in systems that require overlaying and the solution may be accomplished by altering the Kernel to allow the dynamic creation of processes. In addition, the system model for the Kernel limits a single load image per processor and in a system that requires code overlays and that has available the ability for dynamic reconfiguration, multiple load images may be needed on a single processor.

### **4.2. Rationale**

Code overlaying is of great use in embedded systems that have tight specifications for system volume such as in tactical fighter radar systems. Systems like this have limited memory capacity while requiring a great degree of functionality. Code overlaying allows memory to be utilized efficiently by allowing processes that execute once, such as start-up routines, and processes that are executed under special circumstances, such as error recovery routines to be swapped in and out of memory.

### **4.3. Response**

One of the assumptions in the DARK design is that the target processors will not have tight limits on memory. Given that assumption, there seems no need to provide support for code overlays within the Kernel itself.



## 5. Communication Enhancements

### 5.1. Description

The following recommendations are for extensions to the communication scheme adopted by the Kernel:

- Allow the user to select a communication mode that is as fast as possible.
- Allow message priorities that automatically adjust the order of messages in the process queues.
- Enhance the Kernel's ability to ensure message integrity by implementing error detection and recovery within the data link layer.
- Add a parameter to the synchronous send that indicates the number of automatic retries required if the message is NAK-ed.
- Incorporate a network layer that allows Kernel communications between homogeneous and heterogeneous machines.

### 5.2. Rationale

Requirements for communications vary widely among MCCR projects. Some areas that may be common among these requirements are communication speed, message priorities, and message integrity. Fast communications are required in many applications and imply that the Kernel offers the mechanism for such communications and trusts the user to ensure that the communications are performed correctly. One method suggested to achieve greater efficiency in Kernel communications is to eliminate the datagram conversion and implement a straight direct memory access (DMA) transfer as an option. This option can be selected for systems that require fast, no-frills communications and will transfer much of the responsibility for communication verification to the user.

Within many systems developed by the MCCR community there exist emergency, test, and timing messages that must be processed at a higher priority than other messages to ensure that they reach their destination promptly. The Kernel currently does not offer provision for these high priority messages due to the single message port for each process and the lack of message priorities. It is necessary that certain provisions be included.

Another area in which the MCCR community must meet requirements for communication is message integrity. Message integrity is primarily left up to the application code by the Kernel at present and depends a great deal on the communication medium selected for the system. Incorporating certain techniques within the Kernel that can be toggled such as checksums, Hamming correction codes, and automatic retries would help to meet integrity requirements in systems that do not have sufficient built-in integrity checks while keeping communication overhead to a minimum.

Many projects must deal with network communications and if the possibility exists to incorporate a generalized network layer within the Kernel, then it will be of benefit to the MCCR community. This layer would allow Kernel communications between machines over various communication media and accommodate a heterogeneous network, e.g., a Motorola 68020 processor, a Motorola 88000 processor, and a MIPS processor on the same network running the Kernel and communicating with each other.

### 5.3. Response

All the proposed extensions seem desirable. The technical difficulty varies considerably.

Message priorities would be relatively easy to implement, provided a reasonable action to take on message-queue overflow could be specified. Note, however, that such priorities would be *global*; that is, the set of message priorities across all processors would have to be assigned in a globally consistent manner.

Automatic error detection, recovery, and retry pose more difficulty, since they require the sending Kernel to keep copies of originated messages until they are acknowledged or abandoned. This complicates the storage management, and hence the user customization, but is not intractable. It would be easiest to implement this extension as part of a revised *Send and Wait* primitive, since the message text would automatically remain available for a retry.

Providing communications between heterogeneous machines is a complicated task, but can be isolated from the rest of the Kernel at least as far as the Kernel protocol and datagram formats are concerned. This assumes that the application is responsible for reformatting the message bodies themselves, taking due account of byte order, character sets, number representations, and so on.

## 6. Duplicate Load Images

### 6.1. Description

This recommendation addresses the restrictions imposed on the applications programmer by the [KFD 89] requirement for globally unique process names. Implementing a different technique to uniquely identify processes across the system, possibly by utilizing processor id's, will allow load images to be duplicated or cloned on many processors.

### 6.2. Rationale

The [KFD 89] requirement for globally unique process names forces limitations on the methods available to an applications programmer for program partitioning. There are many applications, especially within a tightly-coupled implementation, where partitioning based on data and then cloning the load image to process the partitioned data is advantageous. Cloning load images has advantages for data integrity, processing throughput, and fault tolerance and is therefore recommended for the Kernel implementation.

### 6.3. Response

The globally unique process names are used only during initialization to identify each process to all nodes in the network. These names, however, can be constructed dynamically by the application code – they are merely string values – and so do not preclude the same load image being used on several processors. Indeed, most of the DARK unit tests use this technique, to avoid having to be linked in multiple versions. The problem with incorporating a processor-id in the process name is that all other processes have to know where the process is sited, which conflicts with a DARK requirement. In addition, it would make manual reconfiguration of the application code more difficult, since a process's name would have to change whenever its site changed. However, if these limitations are acceptable, the Kernel does at present provide the ability to construct process names dynamically in a robust manner.

## **7. Single Processor Mode**

### **7.1. Description**

A provision for a single processor mode will allow the Kernel to manage and control communication for a multi-processing application on a single processor. This capability may already be possible within the Kernel, but there may exist additional optimizations that can be included under this mode.

### **7.2. Rationale**

At present, there exist many MCCR applications that can effectively execute on a single processor and that are required to do so, such as in missile applications that only offer space for one processor. As throughput capacities increase with improved processor technology, the number of single processor applications will continue to increase as well. Therefore, the Kernel should address the single processor needs of a growing area of MCCR applications.

### **7.3. Response**

The Kernel can already function on a single-processor target, but with the restriction that the application is a single Ada program. This allows DARK to be used directly for single-processor applications. It would also be possible to customize the code for single-processor use, though the Kernel does not at present do so.

The VMS version of DARK provides exactly the requested capability to run multiple Ada programs, representing multiple DARK nodes, on the same processor. However, this takes advantage of the Ada multiprogramming capabilities of the language implementation, and of the communication facilities of the operating system. While it was straightforward to implement on this target, there is no guarantee that other targets would be as amenable, since their language implementations might not provide reasonable support for Ada multiprogramming.

## 8. Initialization Control

### 8.1. Description

This recommendation involves allowing more application code control over system initialization as an alternative to rewriting the initialization protocol. Some controls recommended include bypassing the Network Configuration Table (NCT) verification and allowing the Master processor to start-up selected processors as they become ready, instead of waiting for all processors to become ready. In addition, the fact that the Master processor is a single point-of-failure in the system is cause for concern and forcing pre-elaboration in some cases to save start-up time is a desirable feature.

### 8.2. Rationale

[KFD 89] specifies that initialization is not a time-critical function. This assumption is not necessarily true within MCCR projects where some requirements for initialization are on the order of 100 ms and even lower. Allowing more application code control over initialization will help the applications programmer satisfy these strict requirements. Bypassing the NCT verification simply leaves the assurance of NCT integrity up to the applications programmer. Allowing control over processor start-up has some risks but it will enable high priority processing to begin as soon as possible. Any single point-of-failure in a mission-critical fault tolerant system is cause for concern and should be avoided if possible.

### 8.3. Response

One of the aims in designing the initialization protocol was to make it both robust and easy to use. To some extent, the features that would implement the extensions are already present, but hidden from the application level.

In particular, it would be easy to omit NCT broadcast and verification; this could even be a customization option. The ability to start the subordinate processors in a specific order already exists; it is embedded in the NCT. Finally, there is not an absolute single point of failure, since the Kernel does not prevent a processor from aborting its "subordinate" initialization (via a time out) and starting a new initialization sequence as "master."

## 9. Processor Global Priority Adjustment

### 9.1. Description

[KFD 89] states that a "Kernel process may set only its own priority." This recommendation is to extend the Kernel to allow priorities to be set globally within each processor; that is, a process may set the priority of any other process within the same processor. It should be noted that if dynamic reconfiguration is implemented to the extent that processes may float between processors, then adjusting priority across processors may also be needed.

### 9.2. Rationale

This extension will be useful when processes must be activated based on aperiodic situations. For example, within a threat detection and response system, threat response processing should remain dormant until a threat is detected; when a threat is detected, threat response should be activated quickly and at a very high priority. There certainly exist other methods to handle these situations, but adding global priority adjustment seems to be a simple and direct change to the Kernel that will offer more flexibility to the applications programmer.

### 9.3. Response

This would be technically straightforward, but its desirability is questionable. One objection is that it damages the predictability of the scheduler: at present, a process that blocks with high priority can be guaranteed to respond rapidly to the expected event; this would not be true if another process could asynchronously downgrade it. Of course, it is currently possible to effect such changes by sending a message to a process and the process adjusting its own priority based on that message.

## 10. Fast Mode Change

### 10.1. Description

This suggestion is for the Kernel, and more specifically the scheduler, to be able to respond quickly to a processing mode change without a restart or much overhead. A mode change, for example, is a jet fighter radar system switching from a navigation mode to a track-while-scan mode and then to a threat response mode without a restart or significant overhead.

### 10.2. Rationale

This extension will enable the user of the Kernel to address specific requirements for fast mode changes in certain MCCR systems.

### 10.3. Response

It is currently feasible to implement a simple, fast mode change by having the critical process(es) blocked on a receive with a high resumption priority. Then, when a mode change needs to be effected, a message can be sent to the appropriate process(es). A more sophisticated mode change scheme could be implemented using some of the other proposed extensions, such as:

- Message priorities: to force through high priority data.
- Dynamic process creation: to create the new mode dynamically on the fly (although this may actually be too time consuming to be reasonable).
- Processor priority adjustment: to raise the importance of a process as a result of a mode change.

## **11. Memory Management**

### **11.1. Description**

This recommendation suggests that the Kernel should offer facilities to manage processor memory resources. Within the present static implementation of the Kernel, it is unclear exactly what facilities are needed, except possibly for memory protection. However, if dynamic reconfiguration is implemented, then more extensive facilities will be needed to manage the memory for processes as they move across processors. Two questions that arise from this suggestion are how will the Kernel obtain the necessary information from the linker to manage and protect the memory space, and how much will the Kernel rely on the run time to perform memory allocation and deallocation.

### **11.2. Rationale**

Memory management additions to the Kernel will offer more capability and flexibility to an applications programmer and will be useful in situations where static allocation of memory is not the most efficient way to handle the memory resource or when memory capacity does not allow it. Also, memory management will support the implementation of dynamic reconfiguration.

### **11.3. Response**

This extension is appropriate if the Kernel is ported to a target that provides memory-management services. There are two technical issues involved in providing memory management through the Kernel. One is the setting of appropriate protections, especially dynamically as processes are suspended and resumed. This is not difficult, and would be an isolable change to the Kernel. However, it could have a severe impact on Kernel performance, since it adds to the cost of every context switch.

The other is the issue of dynamic allocation and deallocation of memory. This is more difficult especially if the Kernel performance is to remain predictable (the classic real-time versus dynamic storage allocation issue).



## 12. Extensibility Summary

Part II of this document presents an industry perspective, and more specifically a Westinghouse perspective, on the suitability of the Kernel to support distributed real-time processing and to fulfill mission-critical computer resource (MCCR) requirements. [KFD 89] states that the "main purpose of the Kernel is to demonstrate that it is possible to develop application code entirely in Ada that will have acceptable quality and real-time performance" and that the resulting prototype "is not intended to solve all the problems of embedded, real-time systems." The Kernel as presented is sufficient to fulfill this purpose. However, in order for the Kernel to be widely adopted for use within the MCCR community, extensions must be implemented. Implementing extensions that will alleviate many of the restrictions related to these areas will produce a Kernel better suited to meet the needs of the MCCR community.

## Appendix A: Machine and Compiler Dependencies

Because DARK is targeted to specific hardware and software configurations, there are a number of general dependencies built into the Kernel. The general hardware and compiler dependencies are discussed in the *Kernel Architecture Manual*. This appendix identifies specific dependencies that are peculiar to the compilation systems and target hardware, as well as subtleties in the code.

These fall into seven categories:

1. 68020: dependencies peculiar to the 68020 CPU architecture.
2. VAX: dependencies peculiar to the VAX CPU architecture.
3. TS-Ada: dependencies peculiar to the TeleSoft V3.22 VMS to 680X0 compilation system.
4. VAX-Ada: dependencies peculiar to the VAX Ada V1.5 compilation system.
5. 68K-Net: dependencies peculiar to the communications network used by 68020 version of the Kernel.
6. VMS-Net: dependencies peculiar to the communications network used by VMS version of the Kernel.
7. Note: these are not dependencies, they identify obscure subtleties in the code.

## A.1. General 68020 Dependencies

Package	Dependency	Comment
<i>bus_io</i>	Note	<i>Multi_send</i> is a pure software broadcast.
<i>clock</i>	68020	<i>Get_time</i> and <i>llc_initialize</i> are implemented in assembly language. Use Ada to access an assembler object. Maximum time between clock interrupts.
<i>context_save_area</i>	68020 FP CoProc	The process context is 68020-specific, including Floating Point CoProcessor data.
<i>datagram_globals</i>	TS-Ada	Uses rep spec to control layout of datagrams. Uses the Ada allocator during initialization.
<i>generic_kernel_time</i>	68020 Note	Defines a signed 64-bit value to hold time. Interfaces to assembly language to manipulate time.
<i>generic_network_configuration</i>	TS-Ada	Must be hand-instantiated, because of a TS-Ada bug.
<i>generic_network_globals</i>	68K-Net	8-bit node address is specific to the DARK ring architecture.
<i>generic_process_table</i>	68020	Stack addresses, word alignment, and <i>context_save_area</i> .
<i>generic_process_managers</i>	TS-Ada	Minimum required stack size.
<i>generic_processor_management</i>	TS-Ada	All Kernel initialization calls encapsulated here to avoid elaboration-order dependencies.
<i>generic_storage_manager</i>	Note	Uses the Ada allocator.
<i>generic_time_globals</i>	Note	Base time defined to be Julian Day 1.
<i>hardware_interface</i>	TS-Ada	<i>Hw_byte</i> defined as a 16-bit number with a rep spec to move the data to the low-order byte. Defines all the primitive data types.
<i>interprocessor_interrupts</i>	68020	Interfaces to assembly language.
<i>interrupt_names</i>	68020	Defines the interrupt addresses reserved by the Kernel.
<i>kernel_interrupt_management</i>	68020	Interfaces to assembly language.
<i>kernel_time</i>	Note	The number of clock ticks/second is specific to the DARK architecture.
<i>low_level_hardware</i>	68020	Interfaces to assembly language.
<i>low_level_process_encapsulation</i>	68020	Uses a machine code insert. Interfaces to assembly language.
<i>low_level_storage_manager</i>	Note	Uses the Ada allocator.
<i>low_level_context_switcher</i>	68020	Interfaces to assembly language.

Package	Dependency	Comment
<i>low_level_interrupt_management</i>	68020	Interfaces to assembly language.
<i>memory_addresses</i>	68020	Defines addresses used in shared memory.
<i>mvme133a_definitions</i>	68020 TS-Ada	Defines board-specific registers and addresses. Uses the TeleSoft-specific notation for hex values with sign bit set.
<i>mz8305_definitions</i>	68020 TS-Ada	Defines board specific registers and addresses. Uses the TeleSoft-specific notation for hex values with sign bit set.
<i>nproc</i>	68020 TS-Ada	Uses shared memory for datagram buffers. Closely coupled with DARK ring architecture. Requires TeleSoft S Linker option to move the heap to shared memory.
<i>parallel_io_controller</i>	68020	DARK communication hardware.
<i>process_encapsulation</i>	Note	There are procedures here that do not return, but rather transfer control elsewhere.
<i>scc_porta</i>	68020 TS-Ada	Defines board-specific registers and addresses. Uses the TeleSoft specific notation for hex values with sign bit set.
<i>scheduler</i>	Note	There is a procedure here that does not return, but rather transfers control elsewhere.
<i>timer_controller</i>	68020	Specific to the DARK architecture. Interfaces to assembly language.

## A.2. Detailed 68020 Assembly Language Dependencies

Package	Dependency	Comment
<i>llcs_body_machine_code</i>	68020	Saves and restores the 68020-specific process context.
<i>llpe_body_machine_code</i>	TS-Ada	Provides the process interface that allows processes to run.
<i>kim_body_machine_code</i>	68020	Interface to the 68020 interrupt mechanism.
<i>llim_body_machine_code</i>	68020	Interface to the 68020 interrupt mechanism.
<i>tc_body_machine_code</i>	68020	Interface to the timer hardware.
<i>ipi_body_machine_code</i>	68020	Interface to communication hardware.
<i>gkt_body_machine_code</i>	68020	Implementation of DARK's time representation.
<i>llh_body_machine_code</i>	68020	Interface to communication hardware.
<i>low_level_clock</i>	68020	Interface to the timer hardware.

### A.3. General VAX/VMS Dependencies

Package	Dependency	Comment
<i>ast_manager</i>	VAX	Interfaces to assembly language. Interfaces to VMS ASTs.
<i>bus_io</i>	VMS-Net	Relies on VAX system services for communication.
<i>clock</i>	VAX	Relies on VAX system services.
<i>context_save_area</i>	Note FP CoProc	No VMS-specific structure has been imposed on the context save area.
<i>datagram_globals</i>	VMS-Ada	Uses rep spec to control layout of datagrams. Uses the Ada allocator during initialization.
<i>generic_kernel_time</i>	VAX Note	Defines a signed 64-bit value to hold time. Interfaces to assembly language to manipulate time. Must be hand-instantiated.
<i>generic_process_table</i>	VAX	Stack addresses, word alignment, and context_save_area.
<i>generic_process_managers</i>	VAX-Ada	Minimum required stack size.
<i>generic_processor_management</i>	VMS-Ada	All Kernel initialization calls encapsulated here to avoid elaboration-order dependencies.
<i>generic_time_globals</i>	Note	Base time defined to be Julian Day 1.
<i>hardware_interface</i>	VMS-Ada	Defines all the primitive data types. Defines VAX-specific unsigned type.
<i>interrupt_names</i>	Note	Defines the interrupt addresses reserved by the Kernel, but not used by VMS version.
<i>kernel_time</i>	Note	The number of clock ticks/second is specific to the DARK architecture.
<i>low_level_hardware</i>	VAX	Uses VMS system services.
<i>low_level_process_encapsulation</i>	VAX	Interfaces to assembly language.
<i>low_level_storage_manager</i>	Note	Uses the Ada allocator.
<i>low_level_context_switcher</i>	VAX	Interfaces to assembly language.
<i>process_encapsulation</i>	Note	There are procedures here that do not return, but rather transfer control elsewhere.
<i>scheduler</i>	Note	There is a procedure here that does not return, but rather transfers control elsewhere.

#### A.4. Detailed VAX Assembly Language Dependencies

Package	Dependency	Comment
<i>am_body_machine_code</i>	VAX	Encapsulates DARK interrupt handlers as VMS AST handlers.
<i>llcs_body_machine_code</i>	VAX	Saves and restores the VAX-specific process context.
<i>llpe_body_machine_code</i>	VAX-Ada	Provides the process interface that allows processes to run.
<i>gkt_body_machine_code</i>	VAX	Implementation of DARK's time representation.

## 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.
- [Flynn 79] Flynn, M.J., J.N. Gray, et. al.  
*Operating Systems, An Advanced Course.*  
Springer-Verlag, New York, NY, 1979.
- [Grover 85] Grover, V.  
*Guidelines for a Minimal Ada Runtime Environment.*  
Technical Report ESD-TP-85-139, Softech, Inc., 460 Totten Pond Rd., Waltham, MA 02154, Jan, 1985.
- [KAM 89] Bamberger, J., T. Coddington, C. Colket, R. Firth, D. Klein, D. Stinchcomb, R. Van Scoy.  
*Kernel Architecture Manual.*  
Technical Report CMU/SEI-89-TR-19, ESD-TR-89-27, Software Engineering Institute, December, 1989.
- [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.
- [Knight 87] Knight, J. and J. Urquhart.  
*On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems.*  
*IEEE Transactions on Software Engineering* SE-13, No. 5:553-563, 1987.
- [Kramer 85] Kramer, J. and J. Magee.  
*Dynamic Configuration for Distributed Systems.*  
*IEEE Transactions on Software Engineering* SE-11, No. 4:424-436, 1985.
- [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.
- [Natarajan 87] Natarajan, N. and T. Jian.  
*Kernel Mechanisms for Distributed Real-time Programs.*  
The Pennsylvania State University, University Park, PA, 1987.
- [Northcutt 87] Northcutt, J.D.  
*Mechanisms for Reliable Distributed Real-Time Operating Systems.*  
Academic Press, Inc., Boston, MA, 1987.
- [Vick 84] Vick, C.R., et. al.  
*Handbook of Software Engineering.*  
Van Nostrand Reinhold Company Inc., New York, NY, 1984.



REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>		1b. RESTRICTIVE MARKINGS <b>NONE</b>	
2a. SECURITY CLASSIFICATION AUTHORITY <b>N/A</b>		3. DISTRIBUTION/AVAILABILITY OF REPORT <b>APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED</b>	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE <b>N/A</b>			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) <b>CMU/SEI-89-TR-40</b>		5. MONITORING ORGANIZATION REPORT NUMBER(S) <b>ESD-89-TR-54</b>	
6a. NAME OF PERFORMING ORGANIZATION <b>SOFTWARE ENGINEERING INST.</b>	6b. OFFICE SYMBOL (If applicable) <b>SEI</b>	7a. NAME OF MONITORING ORGANIZATION <b>SEI JOINT PROGRAM OFFICE</b>	
6c. ADDRESS (City, State and ZIP Code) <b>CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213</b>		7b. ADDRESS (City, State and ZIP Code) <b>ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731</b>	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION <b>SEI JOINT PROGRAM OFFICE</b>	8b. OFFICE SYMBOL (If applicable) <b>ESD/XRS1</b>	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER <b>F1962885C0003</b>	
8c. ADDRESS (City, State and ZIP Code) <b>CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213</b>		10. SOURCE OF FUNDING NOS.	
		PROGRAM ELEMENT NO. <b>63752F</b>	PROJECT NO. <b>N/A</b>
		TASK NO. <b>N/A</b>	WORK UNIT NO. <b>N/A</b>
11. TITLE (Include Security Classification) <b>DARK PORTING AND EXTENSION GUIDE Kernel Version 3.0</b>			
12. PERSONAL AUTHOR(S) <b>Judy Bamberger, Timothy Coddington, Daniel Klein, David Stinchcomb, Roger Van Scoy</b>			
13a. TYPE OF REPORT <b>FINAL</b>	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) <b>December 1989</b>	15. PAGE COUNT <b>50</b>
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB. GR.	
			<b>Ada Kernel</b>
			<b>DARK operating system</b>
			<b>distributed real-time</b>
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This document describes the modifications made to the Distributed Ada Real-Time Kernel (DARK) software when porting it from its original execution environment, the 68020-based testbed built at the Software Engineering Institute, to a VAX/VMS system. This document also contains information about logical extensions to the Kernel, and the impacts thereof, should the Kernel be used in operational systems.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <b>UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/></b>		21. ABSTRACT SECURITY CLASSIFICATION <b>UNCLASSIFIED, UNLIMITED DISTRIBUTION</b>	
22a. NAME OF RESPONSIBLE INDIVIDUAL <b>KARL H. SHINGLER</b>	22b. TELEPHONE NUMBER (Include Area Code) <b>412 268-7630</b>	22c. OFFICE SYMBOL <b>SEI JPO</b>	