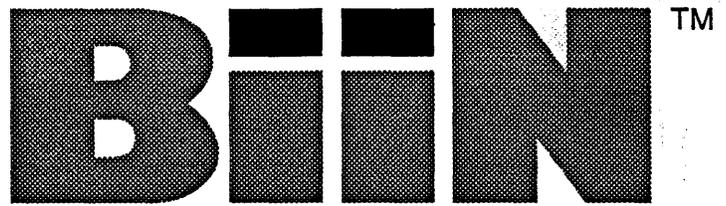


**SYSTEM SERVICES GUIDE  
VOLUME 2 OF 2**

**BiIN**™



# SYSTEM SERVICES GUIDE VOLUME 2 OF 2

Order Code: 6AN9010-1XA00-0BA2

## LIMITED DISTRIBUTION MANUAL

This manual is for customers who receive preliminary versions of this product. It may contain material subject to change.

BIIN™  
2111 NE 25th Ave.  
Hillsboro, OR 97124

© 1988, BIIN™

PRELIMINARY

REV.	REVISION HISTORY	DATE
-001	Preliminary Edition	7/88

BiiN™ MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

BiiN™ assumes no responsibility for any errors that may appear in this document. BiiN™ makes no commitment to update nor to keep current the information contained in this document.

No part of this document may be copied or reproduced in any form or by any means without written consent of BiiN™.

BiiN™ retains the right to make changes to these specifications at any time, without notice.

The following are trademarks of BiiN™: BiiN, BiiN/OS, BiiN/UX, BiiN Series 20, BiiN Series 40, BiiN Series 60, BiiN Series 80.

Apple and MacTerminal are trademarks of Apple Computer, Inc. UNIX is a trademark of AT&T Bell Laboratories. Torx is a trademark of Camcar Screw and Mfg. Ada is a certification mark of the Department of Defense, Ada Joint Program Office. DEC, VT102, and VAX are trademarks of Digital Equipment Corporation. Smartmodem is a trademark of Hayes Corporation. IBM is a trademark of International Business Machines, Inc. MULTIBUS is a registered trademark of Intel Corporation. Macintosh is a trademark of McIntosh Laboratory, Inc. Microsoft is a registered trademark of Microsoft Corporation. Mirror is a registered trademark of SoftKlone Distributing Corporation. WYSE is a registered trademark of Wyse Technology. WY-60 and WY-50 are trademarks of Wyse Technology.

Additional copies of this or any other BiiN™ manuals are available from:

BiiN™ Corporate Literature Dept.  
2111 NE 25th Ave.  
Hillsboro, OR 97124

# Part VI

## Program Services

---

This part of the *BiiN™/OS Guide* discusses program execution, concurrent programming, and scheduling.

The chapters in this part are:

### **Understanding Program Execution**

Explains the static and dynamic structure of programs, including jobs, processes, interprocess communication, and semaphores.

### **Building Concurrent Programs**

Shows you how to build concurrent programs, programs with multiple processes executing concurrently.

### **Scheduling**

Explains how the system schedules processors, physical memory, and I/O devices.

Program Services contains the following services and packages:

#### *concurrent programming service:*

Event\_Admin  
 Event\_Mgt  
 Job\_Admin  
 Job\_Mgt  
 Job\_Types  
 Pipe\_Mgt  
 Process\_Admin  
 Process\_Mgt  
 Process\_Mgt\_Types  
 Semaphore\_Mgt  
 Session\_Admin  
 Session\_Mgt  
 Session\_Types

#### *scheduling service:*

SSO\_Admin  
 SSO\_Types

#### *timing service:*

Clock\_Mgt  
 Protection\_Key\_Mgt  
 Time\_Zone\_Map  
 Timed\_Requests\_Mgt  
 Timing\_Admin  
 Timing\_Conversions  
 Timing\_String\_Conversions  
 Timing\_Uilities

#### *resource service:*

Resource\_Mgt  
 Resource\_Mgt\_AM  
 Resource\_Types  
 Resource\_Uilities

#### *program building service:*

Control\_Types  
 Debug\_Support  
 Domain\_Mgt

## PRELIMINARY

Execution\_Support  
Link\_By\_Call  
Program\_Mgt  
RTS\_Support

*monitor service:*

Monitor\_Defs  
Monitor\_Mgt

# UNDERSTANDING PROGRAM EXECUTION

# 1

## Contents

Definition of a Program .....	VI-1-2
Program Structure .....	VI-1-2
The Program Object .....	VI-1-3
The Domain Object .....	VI-1-4
The Static Data Object .....	VI-1-5
The Instruction Object .....	VI-1-6
The Stack Object .....	VI-1-6
The Public Data Object .....	VI-1-6
The Debug Object .....	VI-1-7
The Handler Object .....	VI-1-8
Invoking a Program .....	VI-1-8
Program Execution .....	VI-1-9
Sessions, Jobs, and Processes .....	VI-1-9
Process Globals .....	VI-1-10
Interprocess Communication .....	VI-1-12
Events .....	VI-1-12
Pipes .....	VI-1-14
Pipes vs. Events .....	VI-1-15
Process Control .....	VI-1-15
Process States .....	VI-1-15
Local Event Cluster .....	VI-1-16
Semaphores .....	VI-1-17
Use of Multiple Processes .....	VI-1-19
Summary .....	VI-1-21

This chapter discusses what a program is and how it executes. It discusses the definition of a program, program structure, how a program is invoked, and how a program executes, including discussions of jobs, processes, the execution environment of processes, interprocess communication, process control, and the use of semaphores for mutual exclusion.

## VI-1.1 Definition of a Program

As explained in the `Program_Mgt` package, there are four program types: executable programs, executable image modules, non-executable image modules, and views. As used in this chapter, the term *program* refers to an *executable program* or *executable image module*.

An executable program is the end product of the compiler/linker translation process. The compiler translates source code into object modules, and the linker then links the object modules into an executable program. In other words, an executable program is a program in the conventional sense of the word.

Like an executable program, an executable image module is the end product of the compiler/linker process. But unlike an executable program, it is an independently linked, protected, and potentially shareable module that provides the runtime environment of a program (for example, the language runtime system or the operating system). An executable image module contains data structures and subroutines that initialize the data structures.

Before execution, a program has a static structure; that is, it is a collection of static, passivated objects that define the elements in a program: a *program object*, a *global debug table*, an *outside environment object*, and one or more *domain objects* (which reference other objects). Sections VI-1.2 through VI-1.2.8 (Pages VI-1-2 through VI-1-8) discuss the static structure of programs.

During execution, a program has a dynamic structure; that is, it is a collection of dynamic, active objects that define the course of execution: a *job*, one or more *processes*, and one or more *stacks*. Sections VI-1.4 through VI-1.7 (Pages VI-1-9 through VI-1-17) discuss the dynamic structure of programs.

## VI-1.2 Program Structure

This section discusses the static structure of programs.

A program is a network of objects rooted in a *program object*. A program object is created by the linker and referenced by a *program AD*. After creating a program, the linker passivates the objects and stores the program AD in a directory. A program consists of:

- A *program object* (Required)
- A *global debug table* (Required)
- An *outside environment object* (Required)
- One or more *domain objects* (required), each referencing:
  - A *static data object* (Required)
  - An *instruction object* (Required)
  - A *stack object* (Created at run time, referenced by a subsystem ID)

- A *public data object* (Optional)
- A *debug object* (Optional)
- A *handler object* (Required only for BiiN™ Ada programs)

Figure VI-1-1 shows the static structure of a program. (The stack object is referenced via a subsystem ID, indicated by dashed lines).

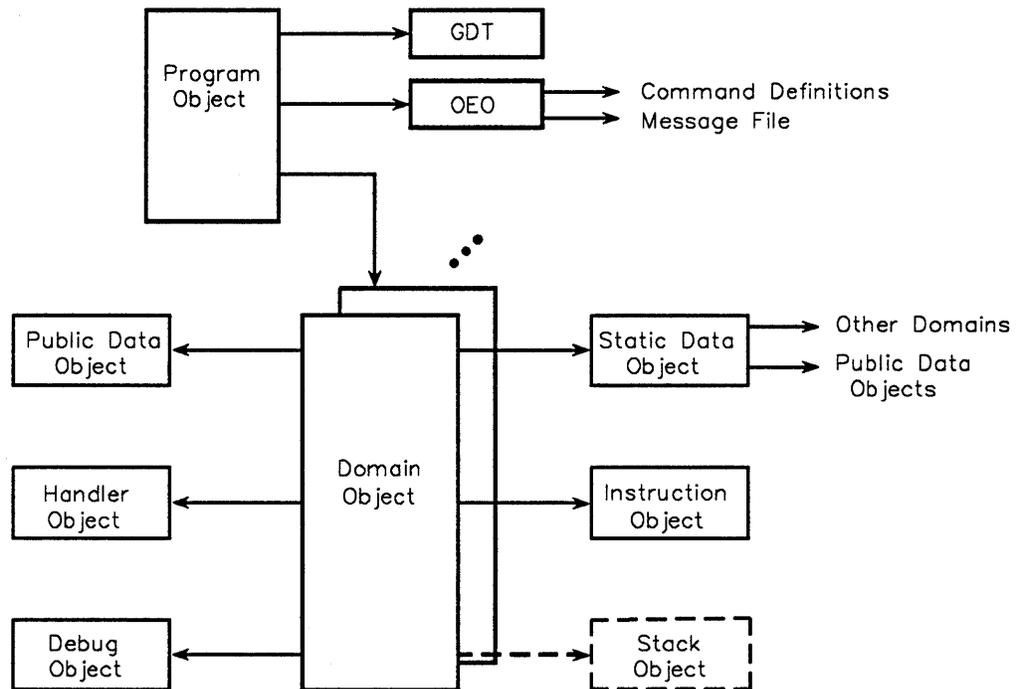


Figure VI-1-1. Static Structure of a Program

The following sections provide a brief introduction to these objects. For more detailed information, see:

- The packages `Program_Mgt`, `Domain_Mgt`, `Debug_Support`, `RTS_Support`, and `Execution_Support`.
- The *BiiN™ Systems Compiler Interface Guide*.
- The *BiiN™ Application Debugger Guide*.
- The *BiiN™ Systems Linker Guide*.

### VI-1.2.1 The Program Object

The program object is created by the linker each time object modules are linked together. It serves as the root object of the program and contains:

- *The program name and version number.*
- *The main entry point of the program.* This consists of the domain AD and procedure number where execution is to begin; generally this procedure is a startup routine in the language's runtime system.

- *An AD to the Global Debug Table (GDT).* The GDT lists the compilation units that were linked to form the program. For each compilation unit, there is a reference to the debug object containing the debug information for that unit.
- *An AD to the Outside Environment Object (OEO).* The OEO references the command definitions and messages associated with the program. These are used by the command language executive (CLEX).
- *A domain AD list.* This is a list of the domains that make up the program.

Figure VI-1-2 shows the structure of a program object.

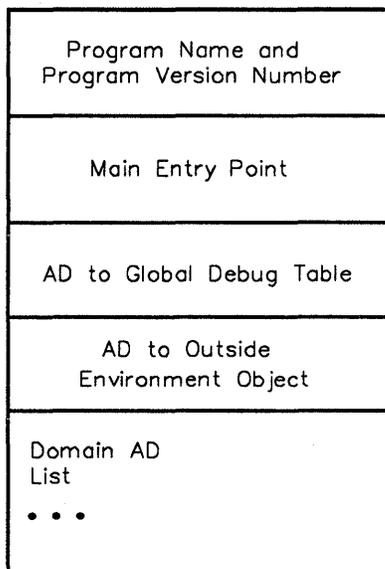


Figure VI-1-2. Program Object

---

### VI-1.2.2 The Domain Object

Domain objects are created by the linker from object modules. Every program has one or more domains. Each domain contains:

- *An AD to a static data object.* The static data object contains ADs to external domains and public data objects so that code in this domain can call procedures and reference data in other domains. The static data object usually contains an AD to the public data object of its own domain.
- *An AD to an instruction object.* The instruction object contains the code for this domain.
- *A subsystem ID.* The ID is used to allocate and reference a stack object at runtime.
- *An AD to a public data object.* The public data object defines the data in this domain that is visible to other domains.
- *An AD to a handler object.* The handler object contains the locations of handlers that should be invoked if a fault or exception occurs.
- *An AD to a debug object.* The debug object contains information needed to debug the code in this domain.

- *A procedure table.* The procedure table lists the addresses and types of the procedures in this domain that can be called from other domains.

Figure VI-1-3 shows the structure of a domain object.

---

Static Data AD	0
Instruction Object AD	4
Subsystem ID	8
Not Used	12
Handler Object AD	16
Debug Object AD	20
Public Data Object AD	24
Reserved	28
Reserved	32
Reserved	36
Reserved	40
Reserved	44
Procedure Table	48
⋮	

**Figure VI-1-3. Domain Object**

---

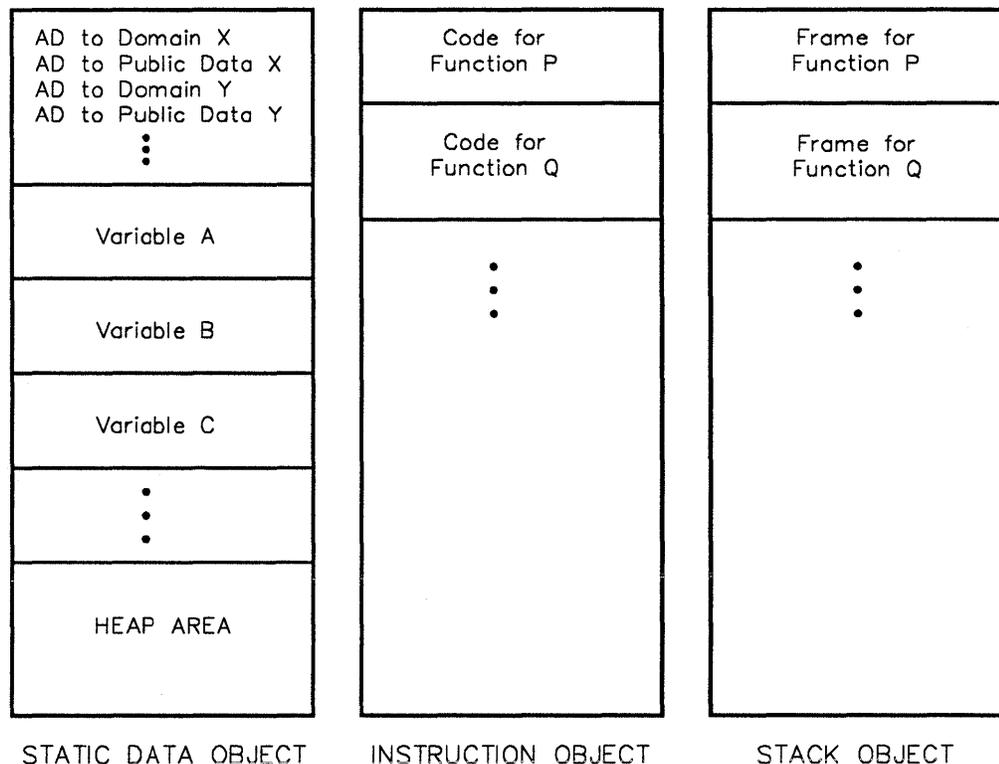
### VI-1.2.3 The Static Data Object

The static data object contains data that cannot be referenced outside the current domain. If a program has only one domain, the static data object contains all variables having a global lifetime. If a program has several domains, variables referenced from another domain (for example, C foreign variables and Ada variables defined in packages with `pragma external`) must be allocated in the public data object.

The static data object also contains ADs to domains whose external procedures can be called from this domain, as well as ADs to objects containing data accessible from this domain.

The static data object can also contain a heap area. Heap allocation routines in the language run-time system (RTS) can resize the static data object during execution.

Figure VI-1-4 shows the structure of a static data object.



**Figure VI-1-4. Static Data, Instruction, and Stack Objects**

### VI-1.2.4 The Instruction Object

The instruction object contains the code for all subprograms defined in this domain. It can also be used to store constant data (but not access descriptors).

Figure VI-1-4 shows the structure of an instruction object.

### VI-1.2.5 The Stack Object

The stack object contains the frames used during subprogram call and return. Each frame contains the parameters, local variables, and housekeeping information related to a call.

All domains in the same subsystem and executing in the same process share a single stack object. Domains in different non-null subsystems use different stack objects.

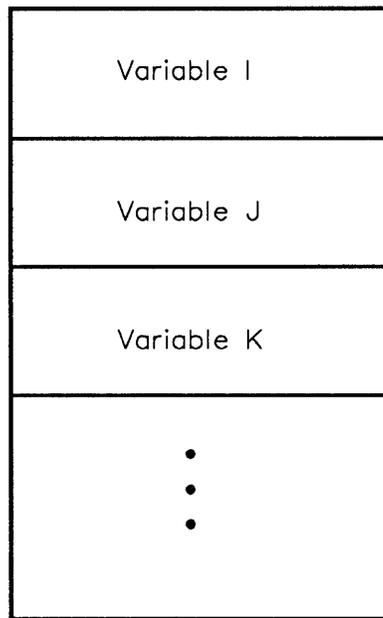
The OS allocates the stack object when program execution begins and resizes it dynamically during execution. See Page VI-1-9 for further information.

Figure VI-1-4 shows the structure of a stack object.

### VI-1.2.6 The Public Data Object

The public data object contains data that can be referenced from other domains (which have an AD to the public data object in their static data objects.)

Figure VI-1-5 shows the structure of a public data object.



PUBLIC DATA OBJECT

**Figure VI-1-5. Public Data Object**

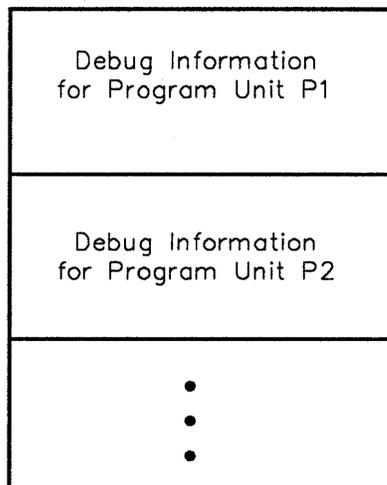
---

### VI-1.2.7 The Debug Object

The debug object contains compiler-generated debug information about the subprograms in the domain's instruction object.

For each subprogram, the debug object has a debug unit that contains information about the blocks, variables, constants, types, and statements in the subprogram.

Figure VI-1-6 shows the structure of a debug object.



DEBUG OBJECT

**Figure VI-1-6. Debug Object**

---

### VI-1.2.8 The Handler Object

Communication between procedures typically occurs by executing explicit call/return instruction sequences. However, another mechanism is required during fault handling and exception propagation. A domain's handler object identifies the language-defined runtime system (RTS) associated with each procedure in the domain. Each RTS has a trace fault handler, a nontrace fault handler, and a number of exception handlers.

The OS handles all faults initially and handles some of them by itself. Upon encountering a fault it cannot handle, the OS needs to transfer control to the RTS fault handler corresponding to the procedure in which the fault occurred. However, the OS cannot identify the procedure's language and therefore cannot directly call the fault handler. Instead, it calls an RTS invoker routine which searches the handler object to locate the RTS's fault handler. The RTS invoker routine is defined by the linker.

When an RTS needs to propagate an exception to another subsystem, the RTS calls the OS. As with a fault, the OS then calls the RTS invoker, which searches the handler object to locate the RTS's exception handler. (If the exception needs to be propagated to another procedure in the same subsystem, the RTS, not the OS, searches the handler object to locate the exception handler.)

See the *BiiN™ Systems Compiler Interface Guide* for more detailed information about the handler object.

### VI-1.3 Invoking a Program

After creating a program, the linker passivates it. Some time later, at a user's request, the BiiN™ Command Language Executive (CLEX) invokes the program in the following way:

- A user requests execution of a program by typing the program's name on a terminal.
- CLEX calls `Directory_Mgt.Retrieve` to obtain the program AD.
- CLEX uses the program's *outside environment object* (OEO) to validate the command line parameters.
- If the parameters are valid, CLEX sets up the job's environment variables and calls `Job_Mgt.Invoke_job` to create the job and its initial process.
- A CLEX-supplied initial procedure—running in the new job's initial process—calls `Program_Mgt.Run` (or `Program_Mgt.Debug`) with the program AD. `Run` (or `Debug`) then calls the program's main entry point. This activates the program, and causes the job's initial process to start executing the program's initial procedure. (This is usually a start-up routine in the language runtime system, from which control transfers to a procedure defined in one of the program's domains.)
- The program executes. After execution, control returns to CLEX (regardless of whether the program terminates normally or abnormally), and CLEX informs the user of the outcome (for example, printing any error messages).

## VI-1.4 Program Execution

This section discusses the dynamic structure of programs.

A program is executed by a job. The job's initial process begins execution in one domain, obtaining instructions from the instruction object and referencing local data and procedures through the static data object.

At any time, the process may switch domains by making an interdomain call (a machine instruction) to a procedure in another domain. When this occurs, the new domain's subsystem ID is used to identify the new domain's stack object. (If the new domain is in the same subsystem as the current domain, the same stack is used). A frame is pushed on the target stack and execution continues in the new domain. A return to the original domain is accomplished by executing a return instruction using the caller's frame.

During execution, the debug object and Global Debug Table are used by the debugger to debug the program (if the debugger was invoked). Also, the handler object is used by the RTS invoker routine to identify RTS fault and exception handlers, as described earlier. (See the *BiiN™ Application Debugger Guide* and the *BiiN™ Systems Compiler Interface Guide* for more detailed information.)

During execution, a process may spawn other processes which execute concurrently. The following sections describe process behavior in greater detail.

### VI-1.4.1 Sessions, Jobs, and Processes

A session is the collection of jobs executed during a user's interaction with the system. A session is usually an interactive logon/logoff period, and it typically contains several jobs.

A job represents an executing program. Each job has its own address space, memory resource, and processing resource. Scheduling, resource control, and resource reclamation are done on a per-job basis. A job can contain multiple processes executing concurrently and sharing data and resources.

A process is one thread of execution within a job. Processes share the job's resources and cooperate to perform the job's computational task. A job begins with an initial process, which can spawn other processes. See Figure VI-1-7.

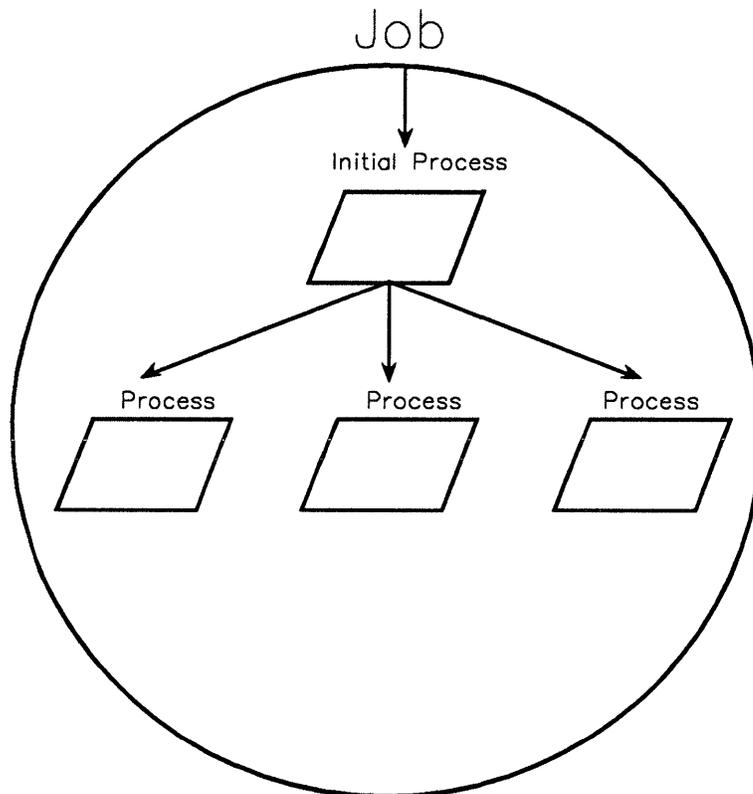


Figure VI-1-7. Job and Processes

#### VI-1.4.2 Process Globals

A process executes in an environment defined by its *process globals*, a list of ADs associated with the process. The entries in a process's globals are named by the `Process_Mgt_Types.process_globals_entry` enumeration type.

Most process globals entries can be modified and assigned arbitrary ADs. Your application controls the correctness of modified entries: that they are not null, have needed access rights, and reference objects of the correct type. Often your application will not need to modify the process globals entries at all; values inherited from the command interpreter or the parent process will suffice.

Table VI-1-1 describes all the process globals entries. The "Inherited?" column indicates whether an entry is inherited when a process is spawned (designated by *PS*), a job is created (designated by *JC*), or both (designated by *PS/JC*).

The "Modifiable?" column indicates whether a process globals entry can be modified. An entry can be modified when a process or job is created or by calling `Process_Mgt.Set_process_globals_entry`. In the "Modifiable?" column:

**PRELIMINARY**

- "Admin-only" Indicates that an entry can only be modified using the Process\_Admin or Job\_Admin packages.
- "Process-only" Indicates that an entry can only be modified using Process\_Mgt or Process\_Admin and cannot be modified using Job\_Mgt or Job\_Admin.
- "Process\_Admin-only" Indicates that an entry can only be modified using the Process\_Admin package.
- "Yes" Indicates that an entry can be modified using any of the four packages (Process\_Mgt, Process\_Admin, Job\_Mgt or Job\_Admin).
- "No" Indicates that an entry can NOT be modified using any of the four packages (Process\_Mgt, Process\_Admin, Job\_Mgt or Job\_Admin).

**Table VI-1-1. Process Globals Entries**

Entry	Description	Inherited?	Modifiable?
home_dir	Process's home directory	PS/JC	Admin-only
current_dir	Process's current directory	PS/JC	Yes
authority_list	Default authority list for objects with master ADs stored by this process	PS/JC	Yes
id_list	IDs for which process is granted access. First ID in list is owner ID and is default owner for objects with master ADs stored by this process. Second ID in list is group ID for BiIN™/UX processes.	PS/JC	Admin-only
cmd_name_space	Command name space used for retrieving command programs specified with relative pathnames	PS/JC	Yes
standard_input	Standard input opened device	PS/JC	Yes
standard_output	Standard output opened device	PS/JC	Yes
standard_message	Standard opened device for writing information, warning, and error messages	PS/JC	Yes
user_dialog	Controlling terminal. Used for operations on /dev/tty	PS/JC	Yes
ux_environ	Used for BiIN™/UX processes; null in other processes	No	Process_Admin-only
lang_environ	Used by language run-time system	PS only	Process_Admin-only
site_environ	Can be used by system administrator for site-specific purposes	No	Process_Admin-only
transaction_stack	Stack of active transactions. If the stack is not empty, the top entry is the default transaction.	No	Process_Admin-only
creator	Process that created this process, with control rights. Null if this process is a job's initial process.	No	No
process	AD to this process, with control rights.	No	No
job	Job that contains this process, with list rights and control rights.	Inherited when a process is spawned but not when a new job is invoked	No

Table VI-1-1: Process Globals Entries (cont.)

Entry	Description	Inherited?	Modifiable?
session	Session that contains this process, with list rights and control rights.	Inherited when a process is spawned and normally when a job is invoked, but not if a job is invoked using Job Admin and specifying a different session.	No, but can be implicitly modified if a job is invoked using Job Admin and specifying a different session.
name	Optional AD to text record containing readable name for this process.	No	Process-only
CLI_extern	For use by Command Line Interpreter (CLEX, for example).	PS only	Process-only
program	For use by the OS.	PS only	Process-only
sms	For use by the Software Management System.	No	Process-only

## VI-1.5 Interprocess Communication

This section discusses events and pipes, two basic methods of interprocess communication.

### VI-1.5.1 Events

*Events* are a mechanism for interprocess communication with these characteristics:

- Events can be used as software interrupts, invoking *event handler* procedures and then continuing the interrupted processes.
- Events can be used to send interprocess messages. Processes can wait for events to be received. If a process is not waiting, events can be queued until the process elects to receive the events.
- Events can carry information between processes, either two words of immediate information or a pointer to a larger data structure.
- Events signalled to a job are signalled to every process in the job.
- Event clusters can be created to define additional event values or to define different process groupings:
  - An event cluster is specified by a process AD, job AD, or explicit cluster AD.
  - Each process has a predefined *local event cluster*; signalling an event using a process AD signals the local event cluster of that process.
  - A job has no cluster; signalling an event using a job AD signals the event to the local event cluster of every process in the job.
  - An explicit cluster is a *global event cluster*. Processes can associate and disassociate with global event clusters. Signalling an event using a global event cluster (AD) signals every process currently associated with the cluster.
  - The local event cluster is used for process control. See Page VI-1-16.
- Events can be signalled to remote processes or jobs.

Events are grouped in *event clusters*, each with 32 *event values*. To signal an event, you call `Event_Mgt.Signal` with an `action_record` that specifies:

## PRELIMINARY

event	An event value (1 to 32).
message	A two-word virtual address. Can be used to send immediate data or a virtual address to the data.
destination	One of: <ol style="list-style-type: none"><li>1. Process with control rights. Event is signaled to the process's local event cluster.</li><li>2. Job with control rights. Event is signalled to the local event clusters of all processes in the job.</li><li>3. Global event cluster with signal rights. Event is signalled to all processes associated with the cluster.</li></ol>

The action record specified to `Event_Mgt.Signal` is passed to any event handler or returned from any `Event_Mgt.Wait` call that receives the event.

Each process controls how it will handle events with a particular event value by assigning the `event_status` record for that value:

handler	Handler to establish for event. If <code>System.null_subprogram</code> , default handler (if any) is reestablished. Otherwise, handler must be in a domain with a nonnull subsystem ID.						
state	New event state. One of: <table><tr><td>enabled</td><td>If the event has a handler, the handler is called for each event received. Otherwise, events are queued and can be dequeued using the <code>Event_Mgt.Wait</code> calls.</td></tr><tr><td>disabled</td><td>Received events are discarded. If an event value's state is changed to <code>disabled</code>, any previously queued events for that value are discarded, emptying the queue.</td></tr><tr><td>handler_disabled</td><td>If the event has a handler, the handler is disabled. Received events are queued and can be dequeued using the <code>Event_Mgt.Wait</code> calls. If the event value's state is then changed to <code>enabled</code> and the event has a handler, then the handler is called for each queued event, emptying the queue.</td></tr></table>	enabled	If the event has a handler, the handler is called for each event received. Otherwise, events are queued and can be dequeued using the <code>Event_Mgt.Wait</code> calls.	disabled	Received events are discarded. If an event value's state is changed to <code>disabled</code> , any previously queued events for that value are discarded, emptying the queue.	handler_disabled	If the event has a handler, the handler is disabled. Received events are queued and can be dequeued using the <code>Event_Mgt.Wait</code> calls. If the event value's state is then changed to <code>enabled</code> and the event has a handler, then the handler is called for each queued event, emptying the queue.
enabled	If the event has a handler, the handler is called for each event received. Otherwise, events are queued and can be dequeued using the <code>Event_Mgt.Wait</code> calls.						
disabled	Received events are discarded. If an event value's state is changed to <code>disabled</code> , any previously queued events for that value are discarded, emptying the queue.						
handler_disabled	If the event has a handler, the handler is disabled. Received events are queued and can be dequeued using the <code>Event_Mgt.Wait</code> calls. If the event value's state is then changed to <code>enabled</code> and the event has a handler, then the handler is called for each queued event, emptying the queue.						
interrupt_system_call	Flag indicating whether the handler can interrupt a blocked system call if the process is in the <code>allow_system_call_interrupt</code> mode. (See the <code>Typemgr_Support</code> package and <code>process_special_conditions.allow_system_call_interrupt</code> in the <code>Process_Mgt_Types</code> package for further information.)						

Figure VI-1-8 shows how received events are processed.

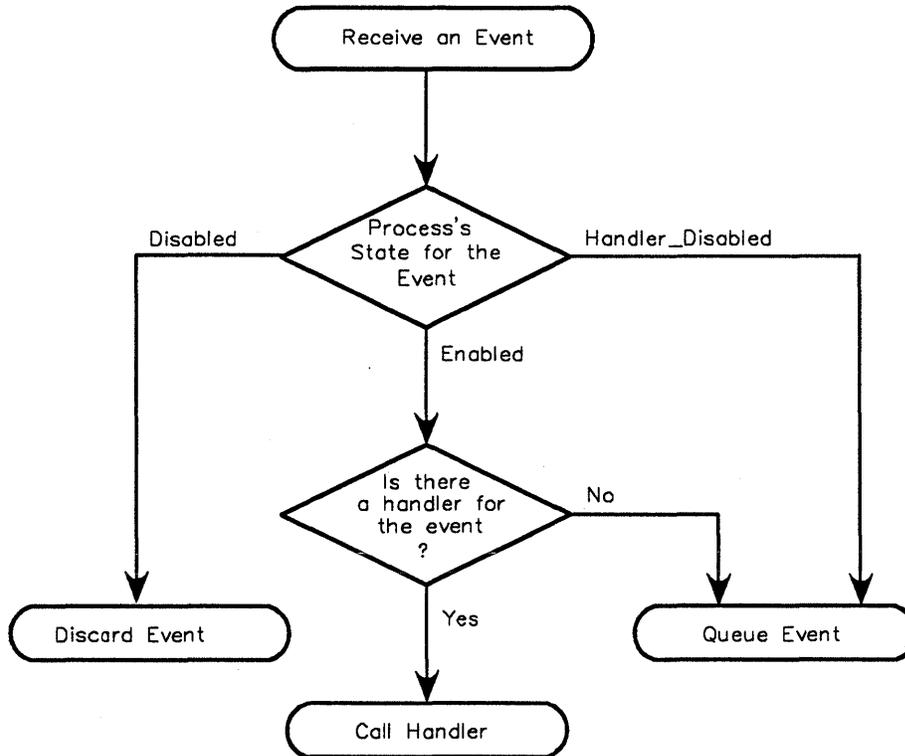


Figure VI-1-8. Events can be Handled, Queued, or Discarded.

### VI-1.5.2 Pipes

A *pipe* is an object that supports one-way I/O transfers between processes.

Figure VI-1-9 shows a pipe used for interprocess communication. One process has the pipe open for output and writes data to the pipe. A second process has the pipe open for input and reads the data written by the first process. The pipe contains a fixed-size buffer used to hold data written by the first process but not yet read by the second process.

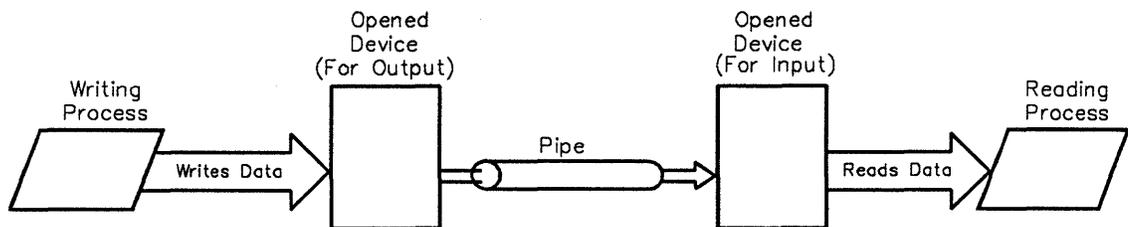


Figure VI-1-9. Pipe I/O

If a process writes to a pipe and there is not enough space in the buffer, then it can block, waiting for space to be freed by the reading process. If a process reads from a pipe but there is no data in the buffer, then it can block, waiting for data to be written by the writing process.

Pipes are one type of OS *device*. Pipes are implemented entirely in software; there are no underlying physical devices, such as terminals or disk drives, that correspond to pipes. Because pipes are software devices, they can be freely created by executing programs, limited only by the amount of virtual memory available to the process.

Pipes are useful because they eliminate the need for intermediate files by allowing the output of one program to be connected to the input of another program. This makes it easier to construct complex programs from smaller existing programs. Both the Command Language and the BiiN™/UX "shell" define an operator for piping, which takes two program invocations and connects them via a pipe. This chapter covers the procedural interface to pipes.

Pipes support the Byte Stream Access Method and the Record Access Method. These I/O access methods provide calls to open pipes for I/O, perform I/O transfers, and close opened pipes. The `Pipe_Mgt` package provides calls to create pipes, check whether pipes are open for input or output, and check whether an arbitrary object is a pipe. The `Pipe_Mgt` package description also describes the pipe implementation of the I/O access methods.

Once created, a pipe exists until no jobs reference it or until it is deallocated by calling `Pipe_Mgt.Destroy`.

### VI-1.5.3 Pipes vs. Events

Both pipes and events provide distributed interprocess or interjob communication. Some comparisons will help you decide which mechanism to use for your application:

- In an application that uses pipes, a subprogram can be given an opened device and use the same code to read or write it whether the opened device is connected to a pipe, a file, or an interactive user.
- An application can send ADs and virtual addresses using events but not using pipes.
- If a message larger than two words is sent with an event, then additional message buffer space must be allocated and managed. Pipes can handle transfers of any size, even transfers larger than the pipe's buffer.
- A pipe keeps the writing process from writing too much unread data, blocking the process (or optionally raising an exception) when the pipe buffer is full. A process signalling an event never blocks and queues of pending events can grow without limit.
- Handlers can be established for both events and for pipe input (using the `Enable_input_notification` I/O access method call).

## VI-1.6 Process Control

This section discusses the creation and control of processes.

### VI-1.6.1 Process States

A program creates a new process within its job by calling `Process_Mgt.Spawn_process`.

Processes are controlled using local events, as described on Page VI-1-16. By sending an event to a process, you can:

- Kill it immediately
- Terminate it "gracefully", giving the process a chance to handle its own termination
- Suspend its execution until a matching resume event is received
- Resume its execution if it is suspended.

After a process has terminated, you can deallocate all storage used by the process by calling `Process_Mgt.Deallocate`.

Figure VI-1-10 shows major process states and the transitions between them.

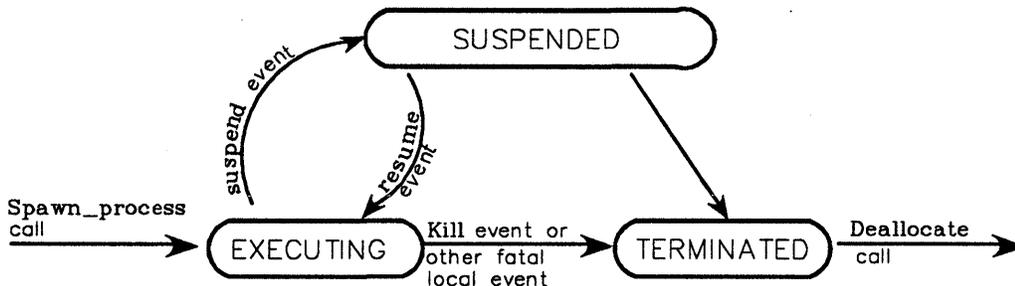


Figure VI-1-10. Major Process States

### VI-1.6.2 Local Event Cluster

To kill, terminate, interrupt, suspend, or resume a process or job, signal the appropriate local event. Table VI-1-2 describes all local event values.

Table VI-1-2. Local Event Values

Value	Description	Modifiable?	Awaitable?	Default
user_1 user_2 user_3 user_4	Available for user. Not used by OS.	Yes	Yes	Enabled. No default handler.
kill	Kills process immediately, even if handling another event.	No	No	Enabled. Default handler kills process.
debug	Requests debugging. Can interrupt any other event but kill.	Event_admin- only	No, unless enabled using Event_Admin	Disabled.
termination	Requests process termination.	Yes	Yes if handler disabled.	Enabled. Default handler kills process.
interrupt	Requests abort of current operation.	Yes	Yes if handler disabled.	Enabled. Default handler kills process.
suspend	Requests suspension of process.	Yes	Yes if handler disabled.	Enabled. Default handler increments suspend/resume count. If count is now one, suspends process.
resume	Resumes process.	No	No	Enabled. Default handler decrements suspend/resume count. If count is now zero, resumes process.

Value	Description	Modifiable?	Awaitable?	Default
hangup	A dialup line connected to one of the process's opened devices has been hung up.	Yes	Yes if handler disabled.	Enabled. Default handler kills process.
io_complete	Available to indicate completion of an asynchronous I/O operation.	Yes	Yes	Enabled. No default handler.
local_xm	Available to signal resolution of a local transaction.	Yes	Yes	Enabled. No default handler.
gcol	Signalled each time a local GCOL run begins in the process's job.	Yes	No	Enabled. Default handler shrinks stacks if unused portions exist.
event_15 to event_32	Reserved by OS.	No	No	Disabled.

## VI-1.7 Semaphores

Processes can share data. But many operations on shared data will only execute correctly if executed by one process at a time. Other processes can be excluded during such an operation by associating a *semaphore* with the shared data structure.

A semaphore is a system object that contains a count and, if the count is zero, a pointer to zero or more processes blocked at the semaphore.

The basic operations on semaphores are P and V. If a semaphore's count is greater than zero, P indivisibly decrements it. Otherwise, P blocks the calling process in the semaphore's prioritized process queue. If processes are blocked at a semaphore, V unblocks and dispatches the highest-priority process. Otherwise, V indivisibly increments the semaphore's count.

A third operation, `Conditional_P`, indivisibly decrements a semaphore's count if the count is greater than zero, returning true. If the semaphore's count is equal to zero, `Conditional_P` does nothing and returns false. A process uses `Conditional_P` to try to acquire a lock, without blocking if the lock is not available.

A semaphore can be used to lock a data structure by interpreting a 1 count to mean that the data structure is available and a 0 count to mean that the data structure is in use. Before accessing the data structure, a process calls P. If the data structure is available, the process continues and the semaphore's count becomes zero, indicating that the data structure is in use. If the data structure is being used by another process, the process calling P blocks in the semaphore's queue. After accessing the data structure, a process calls V. If another process is waiting, V dequeues the highest priority waiting process, leaving the count at zero, indicating that the data structure is still in use by the just dequeued process. If no processes are waiting, V increments the semaphore's count to one, indicating that the data structure is available.

A semaphore used to lock a data structure is called a *binary semaphore*. Figure VI-1-11 shows binary semaphores.

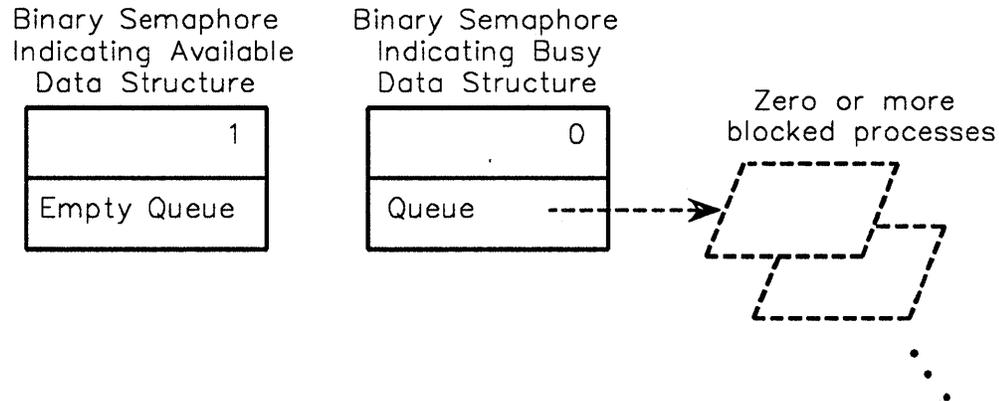


Figure VI-1-11. Binary Semaphores

A semaphore's count can also be used to count units of some resource. For example, a package that manages a buffer pool can use a semaphore's count to indicate the number of free buffers in the pool. P decrements the count and is called when a buffer is allocated; V increments the count and is called when a buffer is released. The semaphore that counts buffers can also be used to block processes that need a buffer when no buffer is available, and then to unblock a process when a buffer is released. In an implementation of the buffer pool package, a second semaphore is needed as a lock on the buffer pool data structure. A semaphore used to count units of some resource is called a *counting semaphore*.

Semaphores are supported directly by the CPU. Semaphore objects are embedded directly in their object descriptors and require no additional active memory. The P, V, and Conditional\_P operations are implemented as single machine instructions and execute very quickly.

Semaphores are not distributed. A process can only use semaphores within its own job or within global objects on its node.

Semaphores used as locks should be held for as short a time as possible, so that other processes are blocked less often and for a shorter time. You can use the Typemgr\_Support package to defer event handling while the process is holding a lock (only for trusted type managers).

A simple but serious bug occurs if a process uses a semaphore as a lock but never releases it for use by other processes. This could occur, for example, if the process executes a return, goto, exit, or raise statement without first calling V, or if an exception is propagated to the procedure in which the process is executing (preventing the process from calling V).

This bug causes all subsequent processes that call P on the lock to block indefinitely, halting all or part of an application. The section "Locking Shared Data Structures" in Chapter VI-2 shows how to write code that ensures that an acquired lock is always released.

Killing or terminating a process that uses semaphores and shared data structures can leave data structures inconsistent and leave binary semaphores with zero counts, preventing other processes from using the data structures. Because semaphores and shared data structures are normally local to a job, this problem can be avoided by killing/terminating an entire job and not just a process within a job.

If an application must acquire multiple locks before executing certain operations, then the locks should always be acquired in the same order. Consider two processes executing an application. Process A acquires semaphore C first and is blocked waiting for semaphore D. Process B acquires semaphore D first and is blocked waiting for semaphore C. Neither process can execute; each waits for resources held by the other. This is a *deadlock* or "deadly embrace" bug that can halt all or part of an application. The bug is avoided if the semaphores are always acquired in the same order, such as <C, D>.

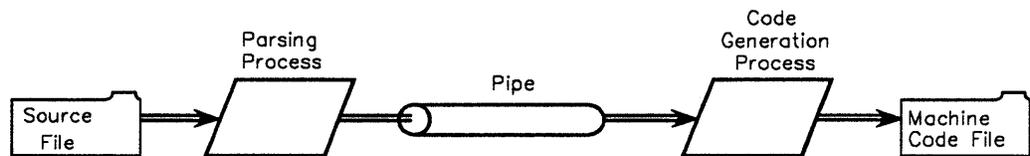
## VI-1.8 Use of Multiple Processes

This section describes three general ways to use multiple processes:

- Processes that do different tasks on data that flows from one process to the next.
- Processes that do identical tasks on different parts of a large data structure.
- Processes that have a client/server relationship in which the client sends a request to the server which sends a reply when the request has been processed.

Some operations on a stream of data can be broken into different sub-operations that can be done by different processes. The entire concurrent program resembles an assembly line where the units of work (or packets of data) flow from one worker to the next, with each doing a special part of the entire operation.

Figure VI-1-12 shows a compiler divided into separate processes to handle parsing and code generation. Data flows through a pipe between the two processes, which can access the pipe using standard I/O access methods.



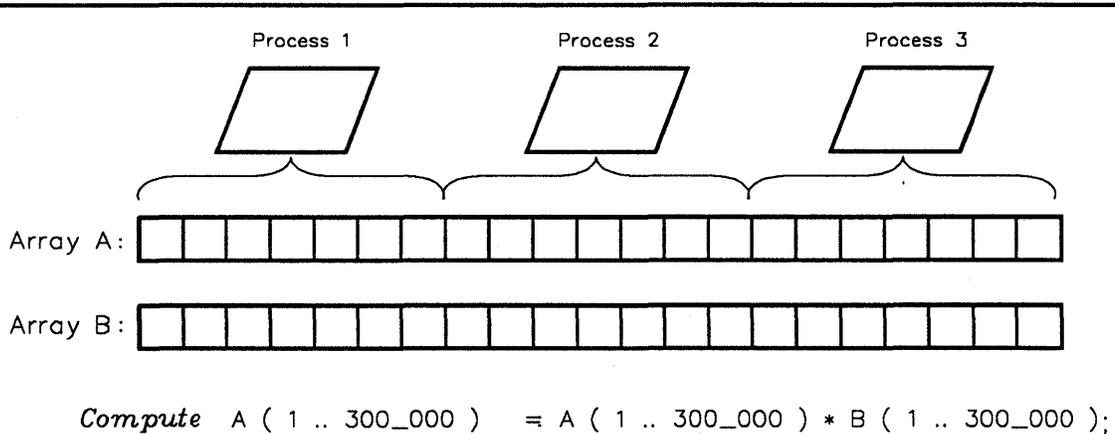
**Figure VI-1-12. Processes Connected by a Pipe Speed Up a Compiler.**

---

Some applications that can use a piped design are:

- Compilers
- Text formatters
- Format converters.

Some computations involve repeatedly doing simple transformations to large arrays of data. Figure VI-1-13 shows how such a computation can be speeded up by dividing it among multiple processes that each perform the identical calculation on a portion of the array.

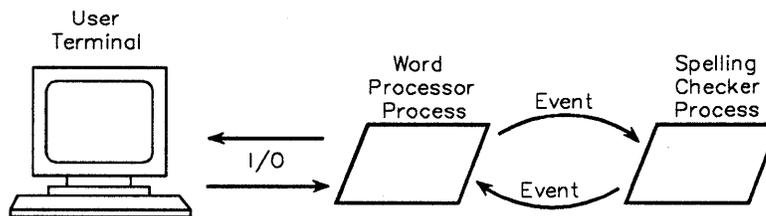


**Figure VI-1-13. Multiple Processes Speed Up a Large Array Calculation.**

Some applications that can use such a design are:

- Image processing
- Advanced computer graphics
- Weather models
- Models of air flow, fluid flow, heat flow, and other engineering properties
- Linear programming
- Monte Carlo simulations
- Programs that examine many possible solutions, such as a chess-playing program or programs that optimize VLSI chip designs.

Breaking an application into client and server processes can be useful when the application both requires interactive or realtime response *and* requires lengthy computations. Tasks that require lengthy processing are relegated to separate server processes. The interactive application sends requests to such server processes and can continue handling user input while the request is being processed. The server process sends a reply to its client when the request has been processed. Figure VI-1-14 shows such a design, used for a word processor with a concurrent spelling checker that checks each word entered by the user.



**Figure VI-1-14. A Separate Spelling Checker Process Preserves Word Processor Responsiveness.**

Server processes can be useful for applications such as:

- Concurrent spelling checking, grammar checking, or style checking.
- Incremental compilation of entered source code.
- Background generation of reports. For example, a process controlling a welding robot may spawn a server process that runs each hour to send operation statistics to a central computer.
- Concurrent language translation: As text is entered in one window in one language, it is translated and displayed in another window in another language. The human translator can edit either window to correct errors in text input or the computer's draft translation.

## VI-1.9 Summary

- The term *program* refers to an *executable program* or *executable image module*.
- A program is a network of objects rooted in a *program object* created by the linker. It consists of a *program object*, a *global debug table*, an *outside environment object*, and one or more *domain objects*. Each domain object references a *static data object*, an *instruction object*, a *stack object* (referenced by a subsystem ID), a *public data object*, a *handler object*, and a *debug object*.
- A program is invoked by CLEX upon user request.
- A session is the collection of jobs executed during a user's interaction with the system.
- A program executes as a job. Each job has its own address space, memory resource, and processing resource. Jobs are grouped into sessions.
- A process is one thread of execution within a job. A job can contain multiple processes running concurrently and sharing data and resources.
- Each process has an execution environment defined by its process globals.
- Events provide flexible interprocess communication.
- Events are used to control processes.
- Pipes support one-way I/O transfers between processes or jobs.
- Semaphores are used to synchronize access to shared data.
- Concurrent processes can improve performance or responsiveness for a variety of applications.



# BUILDING CONCURRENT PROGRAMS **2**

---

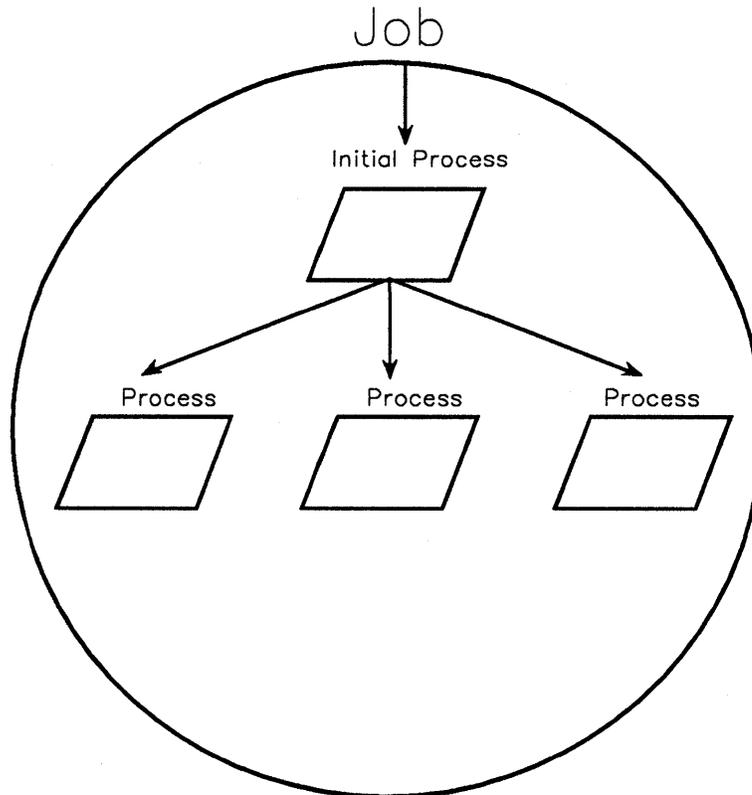
## Contents

Getting a Process Globals Entry .....	VI-2-4
Setting a Process Globals Entry .....	VI-2-4
Creating a Process .....	VI-2-5
Getting Process Information .....	VI-2-7
Suspending and Resuming a Process .....	VI-2-7
Terminating a Process .....	VI-2-8
Signaling an Event .....	VI-2-9
Establishing an Event Handler .....	VI-2-10
Waiting for Events .....	VI-2-11
Connecting Processes with a Pipe .....	VI-2-12
Locking Shared Data Structures .....	VI-2-13

## PRELIMINARY

A concurrent program is one which has multiple processes executing simultaneously within a single job. Concurrent programs are suitable for a wide range of applications and can improve program performance dramatically.

A process is one thread of execution within a job. Processes share the job's resources and cooperate to perform the job's computational task. A job begins with an initial process, which can spawn other processes. See figure VI-2-1.



**Figure VI-2-1. Job and Processes**

---

This chapter shows you some specific techniques for building concurrent programs. You should read chapter VI-1 before this one to understand the concepts underlying programs, processes, and interprocess communication (events, pipes, and semaphores).

**Packages Used:**

<code>Event_Mgt</code>	Manages event clusters. Event clusters provide distributed communications and software interrupts for processes.
<code>Pipe_Mgt</code>	Manages pipes. A <i>pipe</i> is a one-way interprocess or interjob I/O channel. Pipes support byte stream I/O and record I/O.
<code>Process_Mgt</code>	Provides public operations on processes.
<code>Process_Mgt_Types</code>	Declares types and type rights for processes.
<code>Semaphore_Mgt</code>	Manages semaphores. Semaphores can be used to synchronize concurrent access to shared data structures or resources.

This chapter shows you how to:

- Get a process globals entry
- Set a process globals entry
- Create a process
- Get process information
- Suspend and resume a process
- Terminate a process
- Signal an event
- Establish an event handler
- Wait for events
- Connect processes with a pipe
- Lock shared data structures.

Excerpts from the following examples in Appendix X-A are used:

<code>Compiler_Ex</code>	Shows how a compiler can be implemented by dividing parsing and code generation between two processes connected by a pipe.
<code>Process_Globals_Support_Ex</code>	Provides calls to get and set commonly used process globals entries for the calling process.
<code>Symbol_Table_Ex</code>	Shows how a compiler's symbol table manager can synchronize concurrent access using semaphores.
<code>Word_Processor_Ex</code>	Shows how a word processor with a concurrent spelling checker can be implemented using processes and events.

Appendix X-A contains complete listings for these examples.

## VI-2.1 Getting a Process Globals Entry

### Calls Used:

`Process_Mgt.Get_process_globals_entry`  
Gets a process globals entry.

To get a process globals entry, call `Get_process_globals_entry` with the desired entry's name. Entry names are defined by the `Process_Mgt_Types.process_globals_entry` enumeration type.

The following code is excerpted from the `Process_Globals_Support_Ex` package body:

```

45     stdin:          Device_Defs.opened_device;
46     stdin_untyped: System.untyped_word;
47     FOR stdin_untyped USE AT stdin'address;
48     begin
49         stdin_untyped := Process_Mgt.
50             Get_process_globals_entry(
51                 Process_Mgt_Types.standard_input);
52     . . .
62     RETURN stdin;
```

`Get_process_globals_entry` always returns a value of type `System.untyped_word`.

An optional second parameter to `Get_process_globals_entry` allows a caller to retrieve an entry from another process's globals, if the caller has control rights to the other process.

## VI-2.2 Setting a Process Globals Entry

### Calls Used:

`Process_Mgt.Set_process_globals_entry`  
Assigns a value to a process globals entry.

To assign a process globals entry, call `Set_process_globals_entry` with the desired entry's name and its new value. Entry names are defined by the `Process_Mgt_Types.process_globals_entry` enumeration type.

The following code is excerpted from the `Process_Globals_Support_Ex` package body:

```

69     opened_dev: Device_Defs.opened_device)
. . .
79     stdin_untyped: System.untyped_word;
80     FOR stdin_untyped USE AT opened_dev'address;
81     begin
82         if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
83             RAISE Device_Defs.device_not_open;
84
85         elsif not Access_Mgt.Permits(
86             AD => stdin_untyped,
87             rights => Device_Defs.read_rights) then
88             RAISE System_Exceptions.insufficient_type_rights;
89
90         else Process_Mgt.Set_process_globals_entry(
91             slot => Process_Mgt.Types.standard_input,
92             value => stdin_untyped);
93     end if;

```

A value assigned to a process globals entry must have type `System.untyped_word`.

## VI-2.3 Creating a Process

### Calls Used:

`Process_Mgt.Spawn_process`  
Creates a new process in the caller's job.

Creating a new process has two parts:

1. The program must define the initial procedure of the process in a specific way.
2. The program then creates one or more processes that execute that initial procedure.

This section's examples are excerpted from the `Compiler_Ex` package body. The first excerpt shows how a process's initial procedure is defined:

```

44 procedure Parse(
45     param_buffer: System.address;
46     -- Address of connection record.
47     param_length: System.ordinal)
48     -- Not used in this procedure, but required for
49     -- process's initial procedure.
50     --
51     -- Logic:
52     -- Do Pascal parsing using the I/O connections
53     -- specified in the "conn_rec" parameter record.
54 is
55     conn_rec: connection_record; -- Record containing
56     -- parameters.
57     FOR conn_rec USE AT param_buffer;
58     begin
. . .
63     end Parse;
64     pragma subprogram_value(Process_Mgt.Initial_proc, Parse);

```

The initial procedure must have the two parameters shown, `param_buffer` and `param_length`, whether the parameters are used or not. The `subprogram_value` pragma informs the compiler that `Parse` is an instance of the subprogram type `Process_Mgt.Initial_proc`, the type used for a process's initial procedure.

Parameters can be passed between parent and child processes by defining a record type, `connection_record` in this example, that contains the parameters as its fields. The parent process creates a connection record, fills in its fields, and passes its virtual address to the child process. The child process uses the `FOR ... USE AT ...` declaration to specify that its view of the connection record is at the virtual address specified by the parent.

### WARNING

If a parameter buffer specified to a child process is allocated as a local variable (that is, on the stack) of the parent process, then the parent process should not terminate, or return from the call that the buffer is local to, until after the child process terminates (otherwise the buffer would be inaccessible to the child).

There are four different ways to pass information to a child process:

1. Use a parameter buffer local to the parent process. This technique is fine if the parent process does not terminate or return from the call that allocates the buffer until after the child process terminates.
2. Use a parameter buffer allocated as a separate object from the job's heap. The parent process can terminate and the buffer will continue to exist. Such a buffer can be allocated by defining an access type to whatever type is used for the buffer, and then using the Ada `new` operator to create the buffer.
3. Use a parameter buffer allocated in a package's static data area. This technique is undesirable because the buffer cannot be used by concurrent parent processes that each need to communicate with their individual children. If such a parameter buffer *is* used by concurrent parent processes, serious and hard-to-find bugs can result. If this technique is used, access to the parameter buffer should be guarded with a semaphore.
4. Communicate via changes in the child's process globals. Such changes can be specified when the child is spawned. For example, consider a child process that reads its standard input and counts lines, writing the count to its standard output. The child does not need an explicit parameter buffer; it only needs to have its standard input and standard output connected to the desired opened devices. Changes in the child's process globals can be used alone or in combination with a parameter buffer.

The second code excerpt shows how a process is created to execute a particular procedure:

```

146   parse_process: Process_Mgt.Types.process_AD;
147   -- Process executing "Parse".

176   parse_process := Process_Mgt.Spawn_process(
177       init_proc   => Parse'subprogram_value,
178       param_buffer => conn_rec'address,
179       term_action => (
180           event =>      Event_Mgt.user_1,
181           message =>    System.null_address,
182           destination => this_process_untyped));

```

The initial procedure to be executed is specified using the `'subprogram_value` attribute.

The address of the parameter record is specified using the `'address` attribute.

The `term_action` parameter is optional; it indicates the action to signal when the process terminates.

## VI-2.4 Getting Process Information

### Calls Used:

`Process_Mgt.Get_process_state`  
Gets a process's state.

`Get_process_state` produces detailed state information for a process. The process state information is contained in a record of type `Process_Mgt_Types.process_state_rec`. See the `Process_Mgt_Types` package description for more detailed information.

The state information is a snapshot and can change at the same time that the information is being retrieved. For example, `Get_process_state` may indicate that a process is executing even though it blocked while its state information was being retrieved.

## VI-2.5 Suspending and Resuming a Process

### Calls Used:

`Event_Mgt.Signal`  
Signals an event.

`Process_Mgt.Suspend_caller`  
Suspends the calling process. Is normally the last statement in a handler for the `suspend` local event.

An application can suspend a process by signaling the `Event_Mgt.suspend` local event to the process.

An application can resume a suspended process by signaling the `resume` local event to the process.

A suspend or resume event can be signalled to all processes in a job by signaling the corresponding event to the job.

Signaling either event to a process or a job requires control rights.

Each process has a *suspend/resume* count. A positive count is the number of suspend events received without a matching resume event. A negative count indicates the number of resume events that have been received without matching suspend events. Each suspend event received by a process increments the count, and each resume event received decrements the count. The suspend/resume count is zero when a process is created. The process is suspended whenever the count is greater than zero. Note that the resume event that matches a suspend event may be received before the suspend event.

A process can control its response to suspend events, disabling them or establishing a handler for them. A handler for suspend events can simply do whatever cleanup is needed before the process suspends itself, and then call `Process_Mgt.Suspend_caller` to suspend itself.

## VI-2.6 Terminating a Process

### Calls Used:

`Event_Mgt.Signal`  
Signals an event.

`Process_Mgt.Terminate_caller`  
Terminates the calling process.

`Process_Mgt.Deallocate`  
Deallocates the storage used by a process, including the process object and process stacks.

A process can terminate itself by:

- Returning from its initial procedure
- Raising an exception that is not handled within the process
- Calling `Terminate_caller`.

A process can terminate another process or a job by signaling the `termination` or `kill` local event to the process or job. (Recall that control rights are required to signal any event to a process or job.) The difference between the two events is that processes can control their response to termination events but not to kill events.

A process may establish a handler for the `termination` event that does some cleanup and then calls `Terminate_caller`.

A process cannot modify or establish a handler for `kill` events, which terminate a process as soon as they are received; `kill` events can interrupt other event handlers.

When a process terminates, it may be desirable to free the memory that it used, by calling `Process_Mgt.Deallocate`. There is no way for a process that terminates itself to deallocate itself, so deallocation is usually handled by the parent process. If a terminated process is not deallocated, its memory can still be reclaimed by garbage collection or at job termination.

When a process creates a child process, it can specify an event to be signalled when the child terminates. The parent process can wait for that event or establish a handler for it. When the child terminates, the parent receives the termination event and deallocates the child's storage.

The following excerpt from the `Word_Processor_Ex` package body shows how the word processor signals a concurrent spelling checker process to terminate, waits for the termination event, and then deallocates the spelling checker process.

```

306     Event_Mgt.Signal(Event_Mgt.action_record' (
307         event      => Event_Mgt.termination,
308         message    => System.null_address,
309         -- No message.
310         destination => Conversion_Support_Ex.
311             Untyped_from_process(
312                 spelling_checker_process));
313     Event_Mgt.Wait_for_any(
314         events => {
315             child_termination_event_value => true,
316             others => false),
317         action => child_termination_event);
318     Process_Mgt.Deallocate(spelling_checker_process);

```

## VI-2.7 Signaling an Event

### Calls Used:

`Event_Mgt.Signal`  
Signals an event.

To signal an event, call `Signal` with an *action record* that describes the event.

The `destination` and `event` fields specify which event to signal. The `message` field can be used to send a message with an event, formatted as a virtual address.

The following excerpt is from the `Word_Processor_Ex` package body. A spelling checker process has received the location of a word to check via a "word" event. If the word is misspelled, the spelling checker signals a "spelling error" event to the client process.

```

162         if word_misspelled then
163             Event_Mgt.Signal(Event_Mgt.action_record' (
164                 event      => spelling_error_event_value,
165                 message    => (
166                     offset => word_event.message.offset,
167                     AD     => System.null_word),
168                 destination => word_event.message.AD));
169         end if;
```

The `message.offset` field of a spelling error event contains the word location, exactly as received earlier from the client process. The `message.AD` field is not used. The `destination` field is an AD to the client process being signalled. The "word" event received earlier from the client process contained this AD in its `message.AD` field.

A BiiN™ Ada representation specification can be used to pack several fields into the `message.offset` field. An excerpt from the `Word_Processor_Ex` package body illustrates this technique:

## PRELIMINARY

```
84  type word_record is record
85  -- This type encodes a word location into 32 bits,
86  -- allowing a word location to be transmitted
87  -- using the "message.offset" field when an event
88  -- is signalled. The word processor and spelling
89  -- checker are presumed to share a two-dimensional
90  -- array containing the text being edited. Words
91  -- are presumed to not break across lines of the
92  -- array. A word location can thus be specified
93  -- as a line number, a starting column number, and
94  -- an ending column number. The encoding limits
95  -- line numbers to the range 0 .. 65 535 and
96  -- column numbers to the range 0 .. 255.
97  line:      System.short_ordinal;
98  start_col: System.byte_ordinal;
99  end_col:   System.byte_ordinal;
100 end record;
101
102 FOR word_record USE
103   record at mod 32;
104     line      at 0 range 0 .. 15;
105     start_col at 0 range 16 .. 23;
106     end_col   at 0 range 24 .. 31;
107   end record;
. . .
143   word_event:      Event_Mgt.action_record;
144   -- Receives each word to be checked.
145   current_word:    word_record;
146   FOR current_word USE AT word_event.
147     message.offset'address;
148   -- Overlay used to extract word location.,
```

## VI-2.8 Establishing an Event Handler

### Calls Used:

`Event_Mgt.Establish_event_handler`  
Assigns handler and state for an event. Returns previous handler and state.

Establishing an event handler has two parts:

1. The program must define the handler procedure in a specific way.
2. The program must call `Establish_event_handler` to connect the handler to the event.

This section's examples are excerpted from the `Word_Processor_Ex` package body. The first excerpt shows how a handler procedure is defined:

## PRELIMINARY

```
178 procedure Spelling_error_handler(  
179     action: Event_Mgt.action_record)  
180     --  
181     -- Operation:  
182     -- Handler invoked for each 'spelling error'  
183     -- event.  
184 is  
185     misspelled_word: word_record;  
186     FOR misspelled_word  
187         USE AT action.message.offset'address;  
188         -- Overlay used to extract word location.  
189 begin  
190     -- Code to handle misspelled word goes here. For  
191     -- example, this code could highlight the  
192     -- misspelled word on the display and ring the  
193     -- terminal's bell.  
194  
195     null;  
196 end Spelling_error_handler;  
197 pragma subprogram_value(  
198     Event_Mgt.Event_handler,  
199     Spelling_error_handler);
```

A handler procedure must have the `action` parameter shown, which is the event that invokes the handler. The `subprogram_value` pragma informs the compiler that `Spelling_error_handler` is an instance of the subprogram type `Event_Mgt.Event_handler`, the type used for all event handlers.

The second excerpt shows how the word processor process establishes this handler:

```
250     old_event_status: Event_Mgt.event_status;  
251     -- Saves previous event status for the  
252     -- spelling_error local event, so the previous  
253     -- status can be restored before exit.  
  
271     old_event_status := Event_Mgt.  
272         Establish_event_handler(  
273             event => spelling_error_event_value,  
274             status => (  
275                 handler =>  
276                     Spelling_error_handler'  
277                     subprogram_value,  
278                     state => Event_Mgt.enabled,  
279                     interrupt_system_call => false));
```

When a subprogram establishes an event handler, and the subprogram is not the initial procedure or final procedure for its process, then it is good manners for the subprogram to restore the previous event status before returning to its caller:

```
320     old_event_status := Event_Mgt.  
321         Establish_event_handler(  
322             event => spelling_error_event_value,  
323             status => old_event_status);  
324     -- Reestablish previous event status.  
325     -- Value returned is never used.
```

## VI-2.9 Waiting for Events

**Calls Used:**

`Event_Mgt.Wait_for_all`  
 Wait for all of a set of events within a cluster.

`Event_Mgt.Wait_for_any`  
 Wait for any of a set of events within a cluster.

`Wait_for_any` is used to wait until any of a set of events within a cluster is received. The first event in the set that is received is assigned to an action record output parameter. The following excerpt from the `Word_Processor_Ex` package body shows the spelling checker process waiting for a word to be checked.

```

143     word_event:      Event_Mgt.action_record;
144     -- Receives each word to be checked.
145     current_word:    word_record;
146     FOR current_word USE AT word_event.
147         message.offset'address;
148     -- Overlay used to extract word location.,
.
.
.
152     Event_Mgt.Wait_for_any(
153         events => (word_event_value => true,
154                 others => false),
155         action => word_event);

```

`Wait_for_all` is used to wait until all of a set of events within a cluster have been received. The received events are assigned to an array of action records. The following excerpt from the `Compiler_Ex` package body shows a parent process waiting for two child processes to terminate.

```

152     term_events: Event_Mgt.action_record_list(2);
153     -- Array that receives termination events of the
154     -- two child processes.
.
.
.
192     Event_Mgt.Wait_for_all(
193         events =>
194             (Event_Mgt.user_1 .. Event_Mgt.user_2 =>
195              true,
196              others => false),
197         action_list => term_events);

```

## VI-2.10 Connecting Processes with a Pipe

**Calls Used:**

`Pipe_Mgt.Create_pipe`  
 Creates a pipe.

`Byte_Stream_AM.Ops.Open`  
 Opens a device.

The following excerpt from the `Compiler_Ex` package body shows how a pipe is created and opened.

```

134 compiler_pipe: Pipe_Mgt.pipe_AD;
135     -- Pipe that connects "Parse" and "Code_gen"
136     -- processes.
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157 compiler_pipe := Pipe_Mgt.Create_pipe;
158
159 conn_rec := (
160     source_code => source_code,
161     machine_code => machine_code,
162     listing     => listing,
163     parse_out   => Byte_Stream_AM.Ops.Open(
164         Pipe_Mgt.Convert_pipe_to_device(
165             compiler_pipe),
166         Device_Defs.output),
167     code_gen_in => Byte_Stream_AM.Ops.Open(
168         Pipe_Mgt.Convert_pipe_to_device(
169             compiler_pipe),
170         Device_Defs.input));

```

The opened device ADs for the two open ends are stored in a "connection record" that is passed by address to each child process. Each child process can read the connection record and use the opened devices in it.

The Parse process writes the results of its parsing to the `conn_rec.parse_out` opened device, the output end of the pipe. The Code\_gen process reads the same parse results from the `conn_rec.code_gen_in` opened device, the input end of the pipe.

## VI-2.11 Locking Shared Data Structures

### Calls Used:

`Semaphore_Mgt.Create_semaphore`  
Creates a semaphore.

`Semaphore_Mgt.P`  
Enters/locks/waits at a semaphore. If the semaphore's current count is greater than zero, indivisibly decrements it. Otherwise, blocks the caller in the semaphore's prioritized process queue.

`Semaphore_Mgt.V`  
Unlocks/leaves/signals a semaphore. If processes are blocked at the semaphore, unblocks and dispatches the highest-priority process. Otherwise, indivisibly increments the semaphore's current count.

A data structure shared by multiple processes can be locked by locking an associated semaphore. To ensure that all processes observe the locking protocol, the data structure can be managed by a BiiN™ Ada package that handles all access to it. The `Symbol_Table_Ex` package manages a symbol table using such a locking protocol.

The package body creates the symbol table at package initialization; the associated semaphore is created in the same code block:

## PRELIMINARY

```
58     lock: Semaphore_Mgt.semaphore_AD;
59     -- Used to lock symbol table while a process
60     -- is accessing it.
. . .
221 . -- PACKAGE INITIALIZATION
222 . begin
. . .
229 .     symbol_table.lock := Semaphore_Mgt.
230 .         Create_semaphore;
231 .     -- Lock initially indicates table is available.
232 .     -- First "P" on lock will succeed.
```

Each operation provided by the `Symbol_Table_Ex` package locks the semaphore at the beginning of the operation and unlocks the semaphore on all return and exception paths. The following excerpt is from the `Read_symbol_data` implementation in the package body. Note that the semaphore is locked once, but unlocked at each of several different exit paths.

```
184     begin
185
186         Semaphore_Mgt.P(symbol_table.lock);
. . .
194         for i in 1 .. symbol_table.length loop
195             if symbol_table.value(i).name =
196                 fixed_width_name then
197                 Semaphore_Mgt.V(symbol_table.lock);
198                 RETURN symbol_table.value(i).data;
199
200             end if;
201         end loop;
202         RAISE no_such_symbol;
203
204     end if;
205
206     -- This call to "V" is never reached in the
207     -- current implementation. The call is included
208     -- as a safeguard in case code changes make it
209     -- reachable.
210     Semaphore_Mgt.V(symbol_table.lock);
211
212     exception
213     when others =>
214         Semaphore_Mgt.V(symbol_table.lock);
215         RAISE; -- Reraise exception
216             -- that entered handler.
217
218     end Read_symbol_data;
```

# SCHEDULING 3

---

## Contents

What the Scheduler Is .....	VI-3-2
The Scheduler's Objectives .....	VI-3-2
The Scheduler's Task .....	VI-3-2
CPU Scheduling .....	VI-3-3
CPU Scheduling Model .....	VI-3-3
Scheduling Service Objects (SSOs) .....	VI-3-6
Resource-Driven Priorities .....	VI-3-7
Memory Scheduling .....	VI-3-9
I/O Scheduling .....	VI-3-9
Summary .....	VI-3-9

This chapter explains how jobs and processes are scheduled. It discusses the scheduler's objectives and tasks, scheduling service objects (SSOs), CPU scheduling, memory scheduling, and I/O scheduling.

### VI-3.1 What the Scheduler Is

The scheduler is a collection of hardware and software entities whose purpose is to schedule the execution of jobs (and thus processes).

The scheduler is designed for multi-user systems, provides support for real-time applications, and withholds explicit control of scheduling from the user.

The scheduler is not intended to be replaceable; instead, the system administrator can tailor a job's scheduling parameters to suit specific requirements.

### VI-3.2 The Scheduler's Objectives

The scheduler's general objective is efficient use of the system's resources. Specifically, it seeks to:

- Maximize resource utilization
- Maximize system throughput
- Minimize response time for interactive users
- Avoid starvation of jobs
- Degrade gracefully under load
- Minimize thrashing.

To accomplish these objectives, the scheduler is designed to favor:

- Interactive jobs
- I/O-bound jobs
- Jobs with small working sets
- Short jobs.

and to handicap:

- Noninteractive jobs
- CPU-bound jobs
- Jobs with large working sets.

### VI-3.3 The Scheduler's Task

A job needs three resources to execute: physical memory, processor time, and I/O devices. The scheduler attempts to balance the job's need for these resources against their availability and maximize resource utilization for all jobs in the system.

Thus, the scheduler's task is threefold: CPU scheduling, memory scheduling, and I/O scheduling. These are discussed in the following sections.

## VI-3.4 CPU Scheduling

This section discusses CPU scheduling.

### VI-3.4.1 CPU Scheduling Model

When a job is invoked (see Chapter VI-1), it is enqueued on a *scheduling port* served by a *scheduling daemon*. Thereafter, scheduling occurs at three different levels:

- *High-level scheduling* schedules jobs.
- *Medium-level scheduling* assigns priorities to processes.
- *Low-level scheduling* dispatches processes for execution on a processor.

#### VI-3.4.1.1 High Level Scheduling

When the scheduling daemon is activated, it removes a job from the scheduling port and schedules it by enqueueing the job's initial process at the end of one of the queues in a *dispatching port*. The port has 32 queues, ordered in priority from 0 (lowest) to 31 (highest). (Note: Priorities 16-31 are reserved by the OS and never used by user processes.) A process enqueued in this manner is said to be *in the mix*. Putting a process in the mix is called *high-level scheduling*. See Figure VI-3-1.

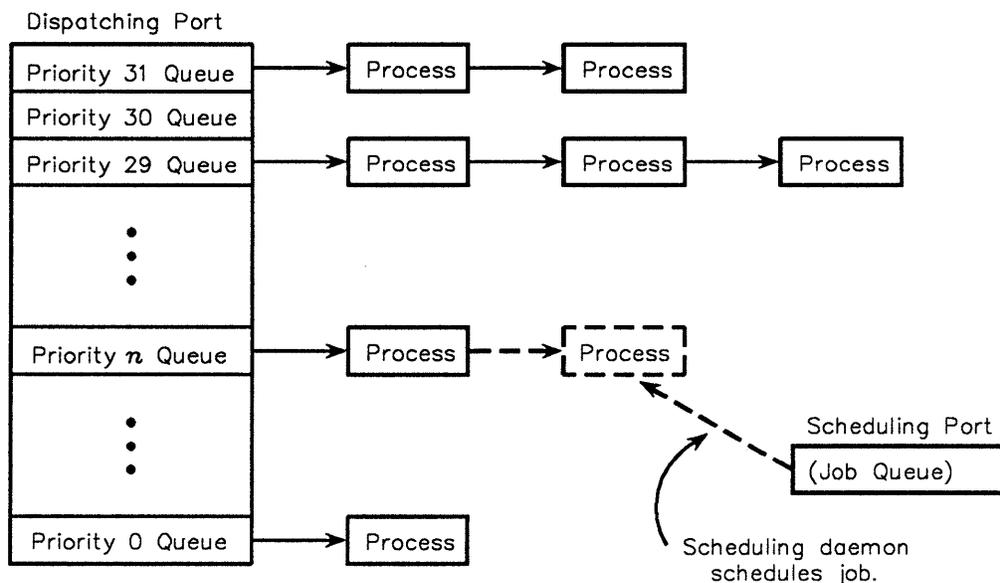


Figure VI-3-1. High-level Scheduling

### VI-3.4.1.2 Low Level Scheduling

Each processor has a pointer to the dispatching port. When a processor is available to execute a process, it dequeues the first process from the highest numbered, non-empty queue in the port, and executes it. This is called *low-level scheduling* or *dispatching*; it is done by microcode, not software. See Figure VI-3-2.

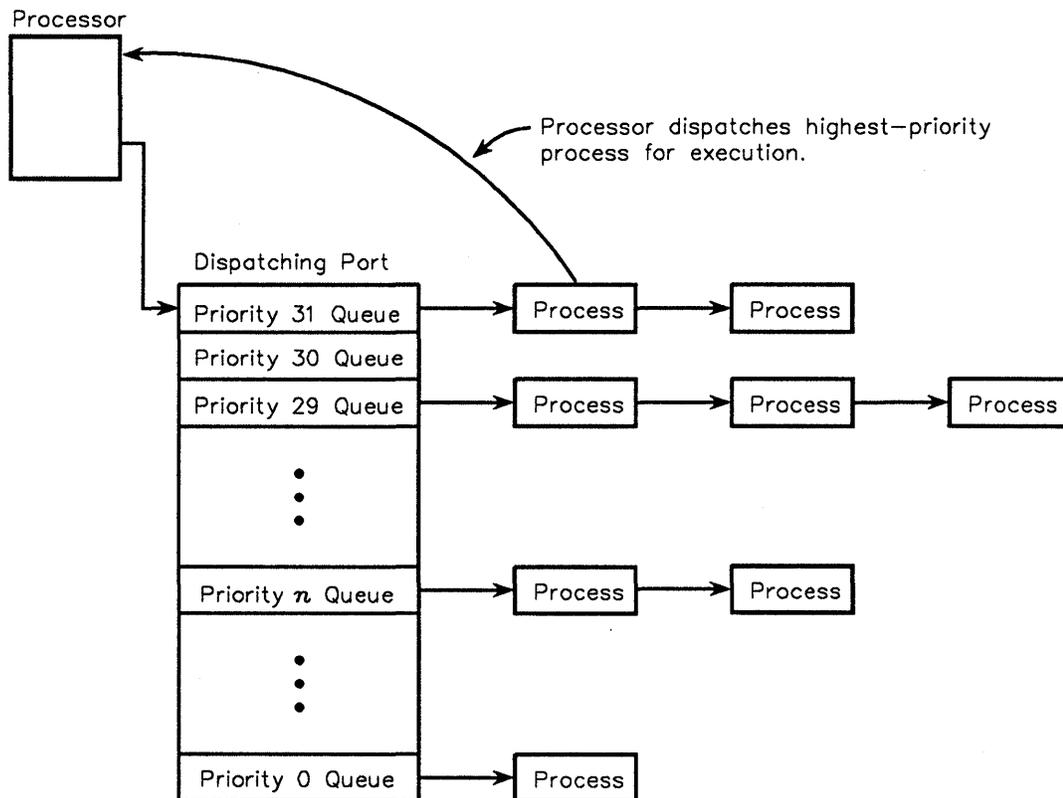


Figure VI-3-2. Low-level Scheduling

### VI-3.4.1.3 Processor Preemption

It is possible for a running process to be preempted (forced to relinquish the processor) by a process waiting in the dispatching port. Whether this occurs depends on the processes' relative priorities and the system's *preemptive threshold*. Currently the threshold is 8: if an interrupt handler or a process with a priority greater than or equal to 8 is ready to run, it will preempt a handler or process running with a lower priority.

Note that the preemptive threshold may change.

See Pages VI-3-6 and VI-3-7 for further information about process priorities.

### VI-3.4.1.4 Classes and Priorities

Each job has a *scheduling service object (SSO)* that determines the type of scheduling service the job receives. Among other things, the SSO defines the job's *service class* and *priority*.

There are four service classes: *real-time*, *time-critical*, *interactive*, and *batch*. All the processes in a job have the job's service class; a job's service class never changes.

There are 32 priorities, corresponding to the priorities in the dispatching port.

See Page VI-3-6 for further information about service classes, priorities, and SSOs .

#### VI-3.4.1.5 Processor Claim and Job Time Limit

Each job has a *processor claim* that defines the *number* of time slices available to the job's processes in a scheduling cycle and a *time limit* that defines the total *processing time* available to the job (and its descendant jobs).

All jobs have the same processor claim, but the *length* of the time slice given to a process is determined by the process's priority.

A job's time limit is determined by the `time_limit` parameter in the `Job_Mgt.Invoke_job` function. The exact interpretation of `time_limit` is subtle; see `Invoke_job` for further information.

When a time slice occurs, a *time-slice fault-handler* checks the processor claim:

- If it is nonnegative, the time-slice fault-handler reduces it by one and gives the process another time slice by putting it at the tail of its priority queue in the dispatching port.
- If it is negative, the time-slice fault-handler triggers a *resource-exhaustion fault-handler*, which checks the job's time limit. If the limit has been exceeded, the job is terminated; if not, the resource-exhaustion fault-handler replenishes the processor claim (charging it against the job's *Resource Control Object (RCO)*), and continues job execution.

#### VI-3.4.1.6 Medium Level Scheduling

The scheduling daemon puts real-time, time-critical, and interactive jobs into the mix immediately, but puts batch jobs in a waiting queue until system load allows them to be put in the mix. Once a process is in the mix, its scheduling depends on its priority, service class, and dynamic behavior. This is called *medium-level scheduling*, and is performed by hardware and the time-slice fault-handler. The following summarizes medium-level scheduling after a job has been put in the mix:

- **Real-time processes:**
  - A real-time process is not subject to *time slice* faults; that is, it executes until it terminates or blocks for I/O.
  - If it blocks for I/O, hardware returns it to the front of its priority queue in the dispatching port when the I/O completes.
  - It is up to the software designer to ensure that a real-time process does not starve other real-time processes and keep them from executing for too long a period.
- **Time-critical processes:**
  - A time-critical process is subject to *time slice* faults. When a time slice fault occurs, it is handled as described in Section VI-3.4.1.5 on Page VI-3-5.
  - If a time-critical process blocks for I/O, it is treated like a real-time process.
- **Interactive and batch processes:**

- An interactive or batch process is subject to *time slice* faults like a time-critical process and is treated in the same way, with one exception: if it receives an additional time slice, the time-slice fault-handler *lowers* the process's priority and places it at the tail of its new (lower) priority queue in the dispatching port.
- If an interactive or batch process blocks for I/O, the time-slice fault-handler *raises* the process's priority to the priority of the requested I/O device, and places it at the tail of its new (higher) priority queue in the dispatching port when the I/O completes. This allows the process to issue several I/O requests for the device at the higher priority.
- Note that the scheduling discipline for real-time and time-critical jobs is based on fixed priorities, but the scheduling discipline for interactive and batch jobs is based on dynamic, resource-driven priorities. See Page VI-3-7 for further information.

### VI-3.4.2 Scheduling Service Objects (SSOs)

A *Scheduling Service Object* (SSO) is associated with a job when the job is invoked. The SSO determines the type of scheduling the job receives.

The system administrator is responsible for creating different types of SSOs and controlling access to them, thus controlling the type of service granted to different jobs (see the `SSO_Admin` package).

The SSO determines the job's service class, SSO priority, time slice, memory type, initial age, and age factor.

#### VI-3.4.2.1 Service Classes

*Service class* denotes the general class of service a job is to receive. Four service classes are defined: realtime, time-critical, interactive, and batch.

**Real-time jobs** are executed in real time. They have very high priority and an infinite time limit. They run in frozen memory, and are not subject to the scheduling process. They are preemptive (given the current preemptive threshold) and always in the mix. If they block for I/O, the hardware reschedules them as soon as the I/O completes.

**Time-critical jobs** have less stringent time constraints than real-time jobs. They have the same priority as real-time jobs, but a finite time limit (when a time slice expires, they are rescheduled or terminated). They need not run in frozen memory, since their time constraints can tolerate page faults. Like real-time jobs, they are preemptive (given the current preemptive threshold) and always in the mix.

**Interactive jobs** involve interaction between a user and a job (an editing session, for example). Interactive jobs run in normal memory, have a finite time limit, and have a lower priority than real-time and time-critical jobs.

**Batch jobs** are background jobs with no attached user. Like interactive jobs, they run in normal memory, have a finite time limit, and have a lower priority than real-time and time-critical jobs.

#### VI-3.4.2.2 SSO Priority

*SSO Priority* is the job's SSO priority. SSO priorities are defined as follows (higher values indicate higher priority):

16 - 31	Reserved for interrupt handlers; not available for program execution.
15	Timing daemon.
12 - 14	Real-time and time-critical jobs.
11	Scheduler and other well-behaved system jobs.
0 - 10	Interactive and batch jobs.

As noted earlier, a handler or process with a priority greater than or equal to the *preemptive threshold* will preempt a processor from a handler or process running at a lower priority. A handler or process with a priority lower than the preemptive threshold cannot preempt a processor. The current preemptive threshold is 8; it may change in the future.

#### VI-3.4.2.3 Time Slice

*Time slice* is the amount of processing time assigned to each process in the job in each dispatching cycle. (It does not include time spent on such incidents as interrupts, processor preemption, or waiting at a port or semaphore).

When a process exhausts its time slice, it is handled as described in Section VI-3.4.1.5 on Page VI-3-5.

For additional information about how time slices are interpreted for different classes of jobs, see `time_slice_enabled`, `time_slice_reschedule`, and `time_slice` in `SSO_Types.SSO_Object`.

#### VI-3.4.2.4 Memory Type

*Memory type* is the type of memory in which the associated job should run. There are two types of memory: *frozen* and *normal*. Frozen memory is nonswappable, nonrelocatable memory; it is used for jobs that cannot tolerate page faults (real-time jobs, for example). Normal memory is swappable and relocatable.

#### VI-3.4.2.5 Initial Age

*Initial age* is a job's age when it first enters the scheduler's waiting queue of swapped-out jobs (see page VI-3-9). Larger values indicate *older* jobs. The job at the head of the queue is the oldest job and will be scheduled next. Giving a job a large initial age helps move it to the head of the queue more rapidly.

#### VI-3.4.2.6 Age Factor

*Age factor* is the rate at which a job ages in the scheduler's waiting queue. On every scan of the waiting queue, the age factor is added to the job's age to determine a new age. The larger the aging factor, the faster a job ages, and the sooner it rises to the front of the waiting queue.

Note that care should be used before assigning an age factor of 0 to a job. Such a job will never age, and may therefore starve in a busy system.

### VI-3.4.3 Resource-Driven Priorities

A single, fixed priority (SSO priority) is used to schedule real-time and time-critical jobs, and their priority is unaffected by resource usage. In contrast, scheduling for interactive and batch jobs uses several priorities and is dynamically driven by resource usage.

### VI-3.4.3.1 Priorities Used

The priorities used in scheduling interactive and batch jobs are:

*SSO priority*      The priority defined in the job's SSO.

*Base priority*      The lowest priority a process can have.

A process's base priority is set when the process is created. The base priority of an initial process in a job is the job's SSO priority. The base priority of a spawned process is the base priority of its parent process.

The System Administrator can change a process's base priority to any value; a user can change it to a value less than or equal to the job's SSO priority.

Changing a job's base priority is accomplished by changing the base priorities of all the job's processes.

*Resource priority*      The priority assigned to a particular resource.

When a process blocks on a resource, its priority is raised to the resource priority (unless its priority is already higher, in which case its priority remains unchanged).

After using a resource, a process must return to its base priority. Each resource class specifies the amount of time in which this must occur. The process's priority is decreased linearly from the resource priority to the base priority in the specified amount of time.

*Running priority*      The priority at which an interactive or batch process is currently running.

Running priority is determined by the other priorities.

### VI-3.4.3.2 An Example

Consider I/O resources as an example (but note that the discussion is applicable to any resource managed by the scheduler).

I/O resources are divided into different classes and each class is assigned a priority; for example, terminals might have priority 10, disks priority 9, and communication lines priority 8. (To keep process priorities less than or equal to 10, all resources have priorities less than or equal to 10).

A process begins executing at its base priority (say, 5) and stays there until it blocks on an I/O resource (say, disks). While blocked, its priority is raised to the disk's priority (9). After the I/O, its priority is decreased linearly (by the same amount at each time slice) until it returns to its base priority (5).

As the process alternates between CPU usage and I/O requests, its priority fluctuates between its base priority and the priority of the I/O resources it requests (these may be different resources with different priorities). The process terminates at some priority level between its base priority and the priority of the I/O resource it last requested.

The presumption behind raising a process's priority to the resource's priority is that if the process issues one request for the resource, it is likely to issue another soon. The overall effect of the model is to favor I/O-bound jobs and penalize CPU-bound jobs, thus maximizing the use of system resources.

## VI-3.5 Memory Scheduling

This section discusses memory scheduling.

Before a process can compete for CPU time, some of its instructions and data must be present in physical memory. (Invoking a job causes a series of faults that bring the program object, domain object, and other objects into primary memory; see Chapter VI-1). Thus, physical memory is as important a resource as the CPU, and memory scheduling is an important part of the scheduler.

The major goal of memory scheduling is to implement the *working set* model of memory management. The working set of a job is dynamically defined as the set of primary memory pages referenced by the job in the last time quantum,  $T$ , measuring backwards from a given time  $t$ . These are the pages which the job used most recently; identifying them and keeping them in memory reduces page fault rates and contributes to system efficiency. (See any standard operating system text for more information about the working set model).

Memory scheduling uses the following model:

- The system maintains a pool of free pages of primary memory.
- As long as there are enough pages in the pool, all the jobs in the mix are allowed to remain there and new jobs are allowed to enter the mix.
- To guard against the depletion of the pool, the scheduler periodically examines memory usage by all the jobs in the mix and transfers back to the pool any pages that are not in the working set of some job. This is done by examining each job's *Storage Resource Object (SRO)*. The SRO references a list of the pages each job has in primary memory. Any page that has not been accessed or modified in the last time quantum,  $T$ , can be returned to the pool. This is known as *SRO page replacement*.
- When the number of free pages in the pool falls below a *low water mark*, the scheduler tries to get more free pages by triggering SRO page replacement more often. If that doesn't succeed, the scheduler then pulls jobs out of the mix and releases their pages. The pages are given to the pool, and the jobs are swapped out to secondary memory. The scheduler keeps a waiting queue of swapped-out jobs.
- In order to achieve fair treatment for all jobs, the scheduler periodically examines the waiting queue and puts the job at the head of the queue in the mix. This ensures that no job starves while waiting for memory. The *aging* parameters in a job's SSO (*initial\_age* and *age\_factor*) determine the job's position in the waiting queue.
- The scheduler also periodically triggers global SRO page replacement, which attempts to free pages from the normal global SRO (pages in the frozen global SRO are not replaced).

## VI-3.6 I/O Scheduling

I/O scheduling is done implicitly through the mechanism of resource-driven priorities, as described above.

## VI-3.7 Summary

- The scheduler is a collection of hardware and software entities whose purpose is to schedule the execution of jobs (and thus processes).

## PRELIMINARY

- The scheduler's general objective is efficient use of system resources.
- The scheduler's task is to perform CPU scheduling, memory scheduling, and I/O scheduling.
- The type of CPU scheduling a job receives is determined by the SSO associated with the job when it is invoked. The SSO determines the job's service class, priority, time slice, memory type, initial age, and age factor.
- The scheduling daemon puts real-time, time-critical, and interactive jobs into the mix immediately, but puts batch jobs in a waiting queue until system load allows them to be put in the mix. Once a process is in the mix, its scheduling depends on several factors.
- The scheduling discipline for real-time and time-critical jobs is based on a fixed priority, but the scheduling discipline for interactive and batch jobs is based on dynamic, resource-driven priorities.
- The major goal of memory scheduling is to implement the *working set* model of memory management.
- I/O scheduling is done implicitly through the mechanism of resource-driven priorities.

# Part VII

## Type Manager Services

---

This part of the *BiiN™/OS Guide* shows you how to build *type managers*, software modules that implement new object types and their attributes.

The chapters in this part are:

### Understanding Objects

Explains objects and their characteristics.

### Understanding Memory Management

Explains how the OS manages memory.

### Building a Type Manager

Shows you how to design and implement a simple type manager.

### Using Type Attributes

Shows you how to define and implement type-specific *attributes*, packages or data structures supported by multiple object types.

### Managing Active Memory

Shows you how to control object allocation and deallocation, and control object reclamation via garbage collection.

### Building Type Managers for Stored Objects

Shows you how to design and implement type managers for objects stored on disk.

### Understanding System Configuration

Explains how a BiiN™ node is configured as a collection of type managers that have configuration requirements. Each such type manager implements the *configuration attribute*.

Type Manager Services contains the following services and packages:

#### *TM object service:*

Countable\_Object\_Mgt  
Global\_SRO\_Defs  
Lifetime\_Control  
PSM\_Trusted\_Attributes  
SRO\_Mgt  
Unsafe\_Object\_Mgt

#### *TM transaction service:*

Local\_Transaction\_Defs  
Local\_Transaction\_Mgt  
TM\_Transaction\_Mgt

#### *TM concurrent programming service:*

Job\_Resource\_Reclamation  
Port\_Mgt  
Typemgr\_Support  
Unsafe\_Port\_Mgt  
Unsafe\_Semaphore\_Mgt

#### *configuration service:*

Configuration

## PRELIMINARY

### *custom naming service:*

Customized\_Name\_Mgt  
Link\_Mgt  
Standalone\_Directory\_Mgt

### *backup service:*

Backup\_Support *not implemented in this release*  
Trusted\_Log\_Mgt *not implemented in this release*

# UNDERSTANDING OBJECTS 1

---

## Contents

Why Use Objects? .....	VII-1-2
Data Abstraction .....	VII-1-2
Memory Protection .....	VII-1-3
Secure and Dynamic Memory Management .....	VII-1-4
Support for Complex and Extensible Applications .....	VII-1-4
Uniform Storage Model for Permanent and Volatile Memory .....	VII-1-4
Distributed Storage Model .....	VII-1-5
How Objects Work .....	VII-1-5
Object Sizes .....	VII-1-5
Types .....	VII-1-5
Object Protection .....	VII-1-6
Attributes .....	VII-1-6
The Inside View of an Object .....	VII-1-8
Address Space Protection .....	VII-1-10
Access Descriptors .....	VII-1-12
Type Managers .....	VII-1-13
Domains .....	VII-1-14
Passive Objects .....	VII-1-15
Active Memory .....	VII-1-16
Passive Store .....	VII-1-16
Passive ADs .....	VII-1-17
Passive Store Protection -- Authority Lists .....	VII-1-18
IDs .....	VII-1-19
Updating Stored Objects .....	VII-1-20
Summary .....	VII-1-20

This chapter explains concepts related to objects and access descriptors. You can find most of this information elsewhere in the BiiN™ document set, but you would have to look in many different places. This chapter is the place where all pieces are brought together, so that you can understand the building blocks of the BiiN™ architecture.

The BiiN™ system has an object-oriented architecture; objects are the building blocks of the system. This is not the first system based on object-oriented programming. The difference between the BiiN™ system and other systems is the rigor with which object-orientation is implemented.

## VII-1.1 Why Use Objects?

Objects are used in the BiiN™ system for the following reasons:

- Data abstraction
- Memory protection
- Secure and dynamic memory management
- Support for complex and extensible applications
- Uniform storage model for permanent and volatile memory
- Distributed storage model.

Each point above will be briefly explained in the following sections.

### VII-1.1.1 Data Abstraction

In most cases your program will not be concerned with the inner workings of objects. An object appears like a black box to the programmer. The box has “jacks” and “buttons”. As you press certain buttons the box takes things from the input jacks and sends something to its output jacks. Or the box performs some other operation. The two important points in the analogy are:

- The box’s buttons do certain things and those things only.
- How the box performs its operations or how it looks on the inside is unknown. (See Figure VII-1-1)

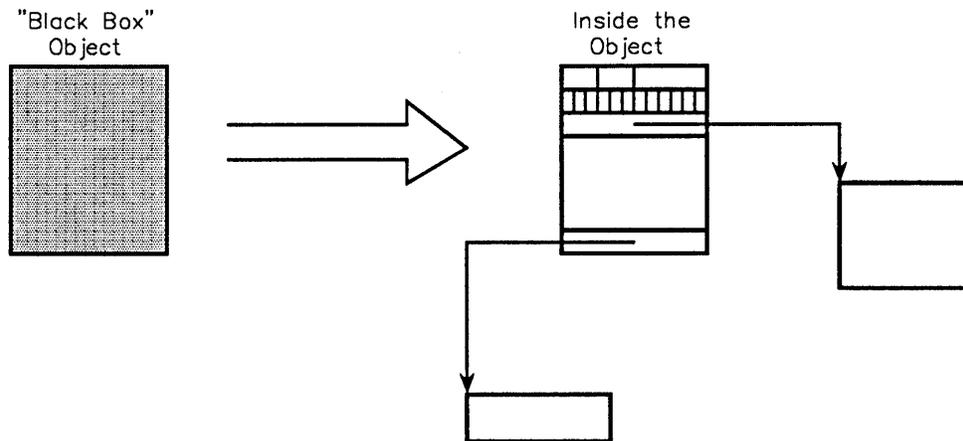


Figure VII-1-1. An Object as a Black Box

Objects present a well defined outside view. That means that their functionality is defined “on their front panel”. How the object works is hidden from view. Data abstraction of this type has two advantages:

- A programmer can use an object without having to know what goes on inside just as you may use a television set without having studied the intricacies of electromagnetism.
- The inside of an object may be altered without affecting programs that depend on the outside of the object.

You can compare objects to Ada packages. The outside view of an object corresponds to the specification of the package. The representation of the object corresponds to the body of the package.

### VII-1.1.2 Memory Protection

Objects are the unit of protection in a BiiN™ system. The memory of a BiiN™ system should not be viewed as an array of bytes but as a network of objects. The way the objects are connected can change at any time as the system runs. Each connection consists of a pointer called the *object index* and a list of access rights. These connections are called *access descriptors* (AD). The both provide and limit access. Connections can be made based on a strict “need to know” basis. Connections can only be made (ADs created) by the BiiN™ Operating System. The BiiN™ Operating System uses special hardware instructions to manipulate ADs. Every access to memory involves checking

- that an AD presented is a valid AD,
- that an AD has proper access rights,
- that the reference falls entirely within the referenced object.

While objects are protected by ADs, ADs are protected by the hardware. Special instructions are required to create and copy ADs. Nobody, not even the operating system, can circumvent this protection mechanism.

### VII-1.1.3 Secure and Dynamic Memory Management

Objects are dynamic. They can be of any size from zero to four Giga bytes. They can be dynamically created, resized, and destroyed. Unneeded objects are automatically removed. For example you can create an object, change its size as many times as you want over the lifetime of the object and then simply abandon it. The operating system will pick up after you. Long running or very large programs can also explicitly control garbage collection. This relieves the operating system considerably.

### VII-1.1.4 Support for Complex and Extensible Applications

Complex programs can never be entirely free of bugs. In a complex system a constant concern is that one program module not corrupt another. This problem is particularly hard to handle in conventional architectures: The instructions or data that have been corrupted may not even be related to the corrupting module.

This is a particularly acute problem when you want to extend important, sensitive, and complex applications, or maybe the OS itself. The traditional solution to the problem is to adopt a two-view scheme. In a two-view scheme the address space is divided into two levels, one level reserved for the operating system, and one level for the user. The interaction between the two levels is severely limited. The two-view scheme restricts functionality.

If address space is shared between user and operating system one risks major breakdowns of the combined system.

In the object-oriented architecture of a BiiN™ system addressing errors are confined to their origin: A wrong address will also always be an invalid address. This is done with a multiple-view scheme. Every application program, every system routine, in fact, every job runs in its own protected address space. All jobs execute at the same level. The important ingredient in the multiple-view scheme is an efficient call/return mechanism that allows communication between protected address spaces.

For example, extensions to the OS run at the same level as the OS and are therefore able to use its full functionality. The same applies to applications. Any program can be easily extended without compromising reliability of the original program.

### VII-1.1.5 Uniform Storage Model for Permanent and Volatile Memory

The BiiN™ system extends its model of protection and its object-oriented architecture to permanent storage. Objects in permanent memory (such as magnetic disks) are called *passive objects*. Objects in volatile memory are termed *active objects*. Permanent memory is termed *passive store*. There can be multiple active versions of an object but only one passive version at any time. In order to read the contents of an object or to write an object, the object has to be *activated* first. When a change to an object should become permanent, the object will be *passivated*. That means that either a new passive object will be created, or an existing passive version of the object will be updated. When multiple active versions of an object are present, the BiiN™ Operating System ensures that obsolete active versions cannot corrupt the passive object.

## VII-1.1.6 Distributed Storage Model

Passive store is distributed -- spread over multiple BiiN™ nodes and transparently accessible from any node. One can view passive store as the glue that holds a distributed BiiN™ system together. Passive store is divided into volume sets. Passive objects are stored on volume sets. Along with each passive object, a *master AD* is stored on the same volume set. That passive AD contains a *unique identifier* (UID), unique for all times and on all BiiN™ nodes. Even if a disk is moved to another BiiN™ node or BiiN™ system, the passive objects stored on that disk will still be uniquely identified.

## VII-1.2 How Objects Work

In the previous section you have learned what objects are, namely typed and protected memory segments. In this section you will learn how objects function in the BiiN™ architecture.

An object is characterized by a number of properties such as size, lifetime, type and a list of attributes. Objects can also be active or passive. In the following sections you will learn about these properties in more detail.

### VII-1.2.1 Object Sizes

Objects can have sizes ranging from zero to four Giga bytes. Object sizes are rounded. (How object sizes are rounded is explained in chapter VII-5.) Objects can be created resized and destroyed at runtime (see Figure VII-1-2).

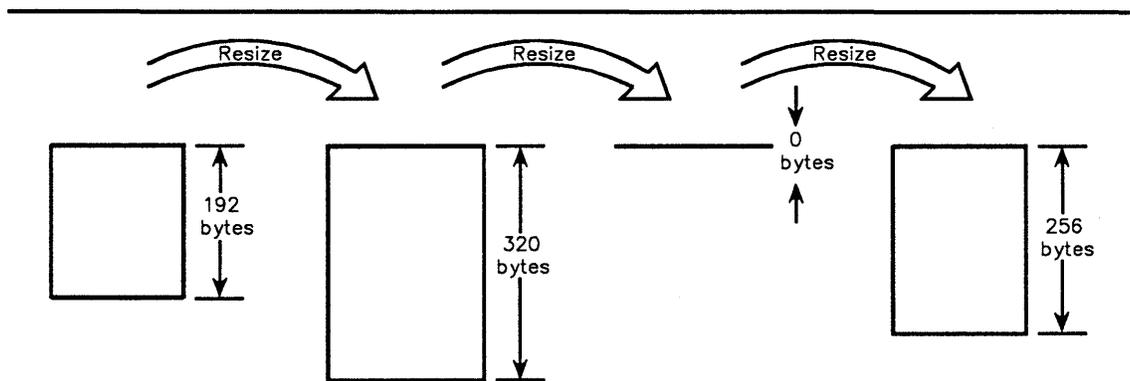


Figure VII-1-2. An Object Can be Resized

### VII-1.2.2 Types

You probably know what typing is from programming languages such as Ada or Pascal. In one sense object types in a BiiN™ system are no different than data types in Ada. Since most of the BiiN™ Operating System is written in BiiN™ Ada, object types are implemented to a certain degree as Ada types. In another sense object types are very different from Ada types. Data in Ada is typed only at compile-time while objects are also typed at runtime. Whenever a software module attempts an operation on an object in a BiiN™ system, the OS first checks whether the operation is allowed for the object. While you can get around compile-time typing by using conversion functions or type overlays, there is no way to circumvent runtime typing

There are a number of predefined *system types* such as *disk*, *file*, *job*, or *program*. (For a complete list of system types refer to the Appendix of the *BiiN™/OS Reference Manual*.) On top, there is one peculiar type of objects called *generic* objects. Generic objects are untyped although, strictly speaking, they have a defined type, the so-called *generic* type.

You are not limited to the system types. Just as in Ada, you can define your own types and implement them on the system.

Object typing is complete and pervasive, more so than typing in programming languages. There are no backdoors that let you bypass the typing mechanisms.

### VII-1.2.3 Object Protection

Typing protects an object from operations that are not defined for the object. There is another mechanism that protects the contents of the entire address space. This protection is provided by protected pointers called *access descriptors* (AD). As the name indicates, ADs provide access to objects. At the same time ADs limit access. Protection by ADs is complete. No object can be accessed without an AD. You can go so far as to identify an AD with the object.

Figure VII-1-3 illustrates the relationship between an object and an AD in a simplified way.

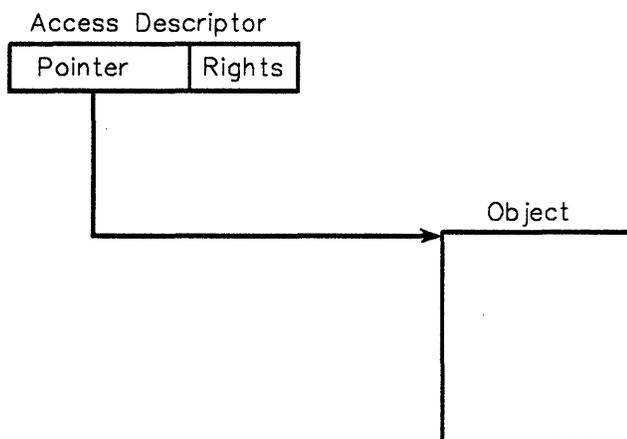


Figure VII-1-3. Object and Access Descriptor

---

### VII-1.2.4 Attributes

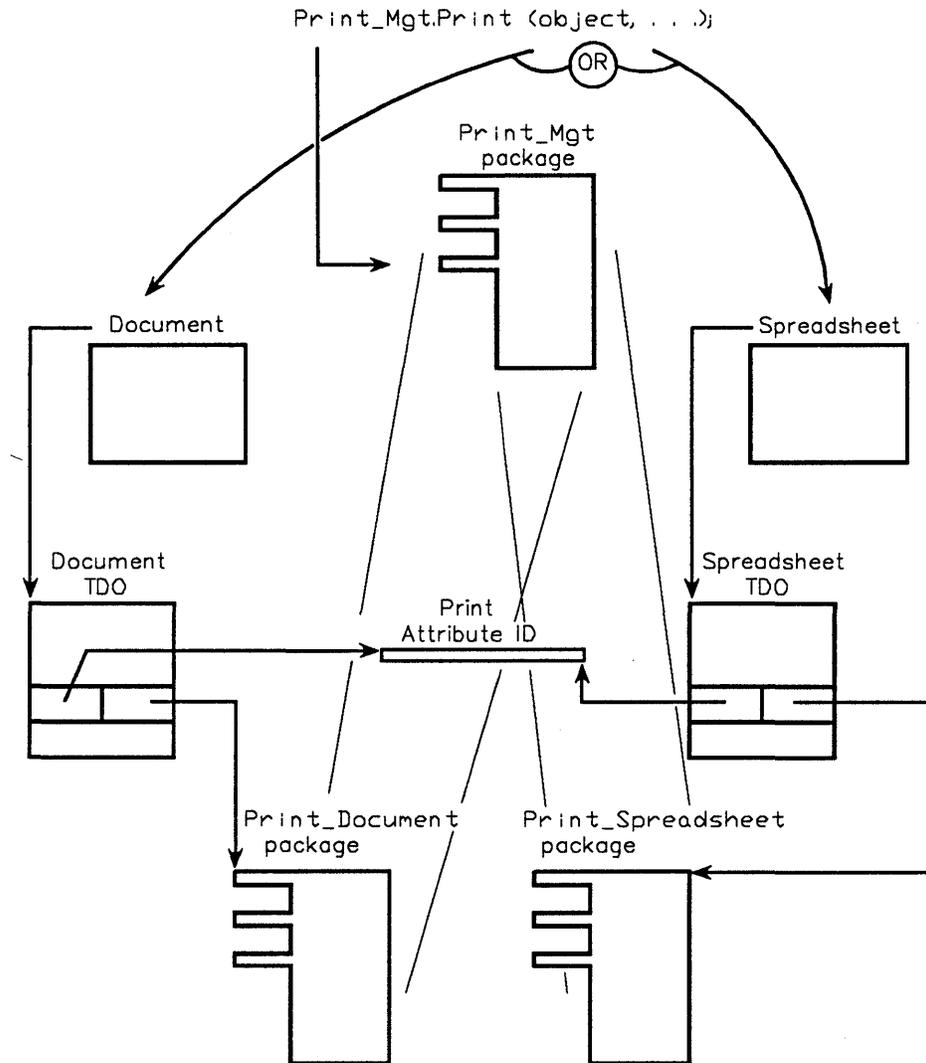
While typing of objects serves two functions, namely protection and data abstraction, the same applies to attributes. Attributes are the means by which the prime capability of objects is realized; objects describe the operations that can be performed on them. An attribute is itself an object that acts as a label. The label typically describes an operation such as `Byte_Stream_AM.ops.Read`. All objects that allow `Byte_Stream_AM.ops.Read` carry a reference to this attribute. The mechanism works like this:

Objects have an attribute list that consists of `<attrib-ID,attrib-value>` pairs. The attribute-ID part references the attribute while the attribute value is typically an AD to a routine that implements the operation for the type.

## PRELIMINARY

All attributes contained in a particular object's attribute list apply to that object. In addition to these attributes an object inherits all attributes defined for its type. Those type-specific attributes are defined in the object's TDO.

For an example and an illustration of these dependencies see Figure VII-1-4.



**Figure VII-1-4. How Attributes Work**

In Figure VII-1-4 there are two objects, a spreadsheet object and a document object. Both have inherited the attribute "printable" from their respective TDOs: The attribute lists of the two TDOs contain a reference to the same attribute "printable". The attribute values however are different: The document TDO has an AD to a package that implements printing of documents (named Print\_Document) while the spreadsheet TDO has an AD to a package that is capable of printing spreadsheets (named Print\_Spreadsheet).

Before concluding this section on attributes we shall briefly touch upon the general protocol of how attributes are implemented in a BiiN™ system.

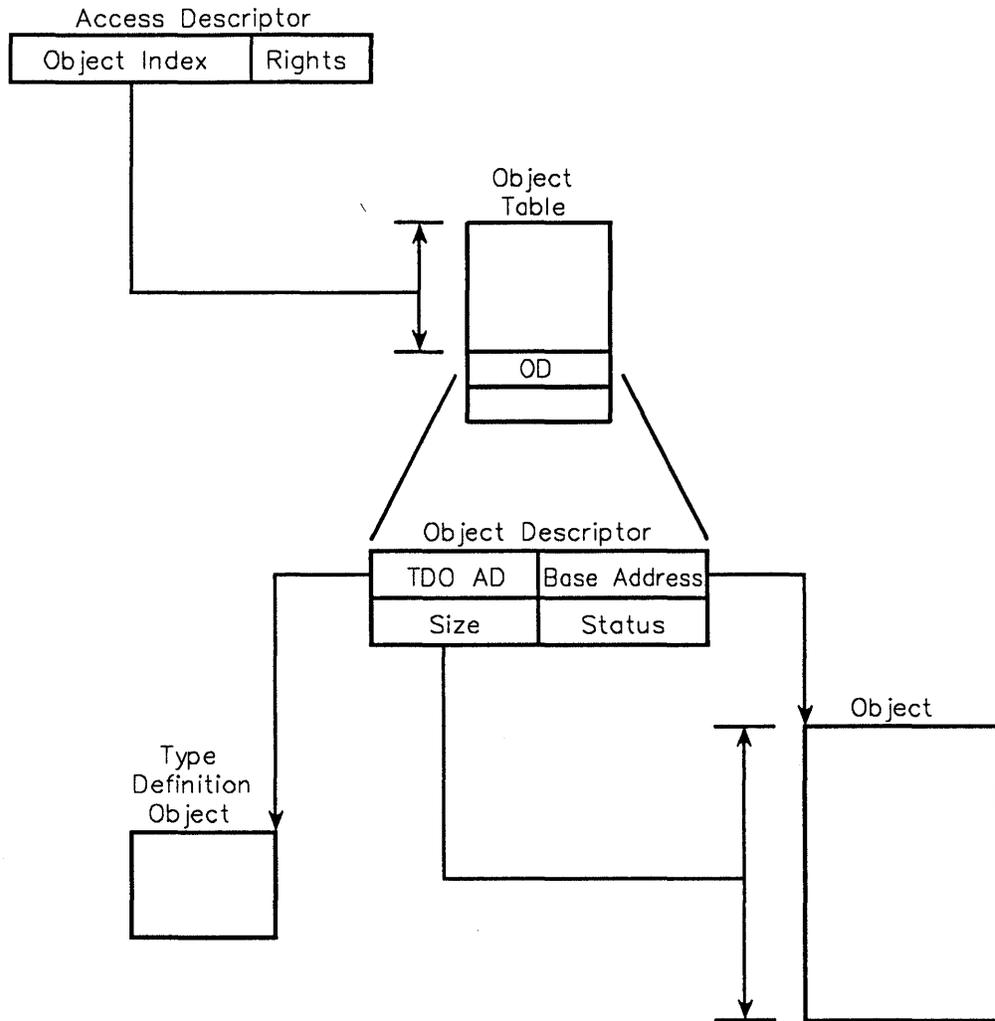
Generally an implementor will establish a 1:1 correspondence between Ada attribute packages and attributes. There will be one attribute package for each attribute. The attribute package only contains subprograms and no other declarations. However, an attribute package can be nested inside another package that provides data declarations and subprograms common to all types. An attribute package must also have the Ada `package_type` pragma. This marks the package as an attribute package and binds it to the attribute ID, which is identified by its pathname. The body of an attribute package is empty.

As the next step, the implementor of an attribute will define various *instances* of the attribute package. These instances are the type- or object-specific implementations of the attribute package. In Figure VII-1-4 `Print_Spreadsheet` and `Print_Document` are such instances of one attribute package `Print`.

Instances have their own package specifications which all match the specification of the attribute package. The instances are bound to the attribute package by the `package_value` Ada pragma. Every instance has its own specific body and runs in its own domain. Instances cannot be merged into one domain with other packages.

### VII-1.2.5 The Inside View of an Object

After having learned about the characteristics of an object, we proceed to explore how these concepts are implemented in the memory of a BiIN™ system. Figure VII-1-5 illustrates the inside view of an object. We have already learned about objects and ADs. Here we see that there are some more details to the picture:



**Figure VII-1-5. Objects Are Typed and Protected**

An object consists of two parts, the object descriptor (OD) and the object's representation. When we talk of the size of an object, we refer to the size of its representation. The representation holds the contents of the object. The object descriptor on the other hand holds important information about the object, such as the physical address of its representation and its size. As Figure VII-1-5 indicates, an AD to an object points to the object descriptor not the object's representation. All object descriptors on one BiiN™ node are held in a one place, the *object table*. An object's representation may be moved around in memory by the BiiN™ Operating System but the object descriptor always stays in the same place.

The object's type is defined in the object descriptor by an AD stored there that points to a *type definition object* (TDO). There is one TDO for each distinct type. That means that two objects have the same type if their object descriptors reference the same TDO.

This model of objects with its two parts, object descriptor and object representation allows for a peculiar object, an object of length zero. Such an object has no representation and therefore really has zero length. This means that all information that pertains to the object is contained in the object descriptor. Objects of length zero are very useful as unforgeable identifiers. They

can be compared to license plate numbers. The significance of a license plate number is not the information contained in it but the fact that it is different from all other license plate numbers.

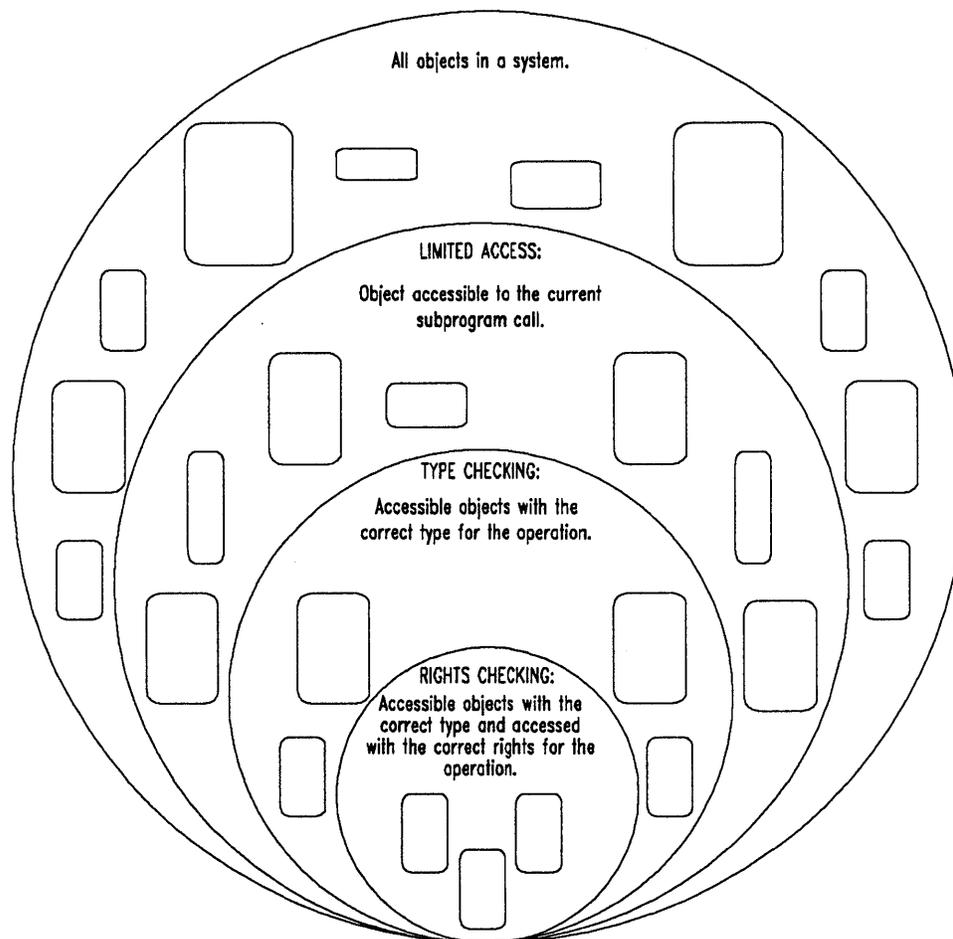
### VII-1.3 Address Space Protection

As software grows more and more complex, bugs become impossible to eradicate. No software engineer, nor any company can guarantee that their software products will not fail under any circumstances. Such software failures can have disastrous results as processors pervade our daily lives. It has therefore become imperative that failures be detected at their origins and that their influence be confined.

The most dangerous types of errors are addressing mistakes. By making such a mistake, a routine can corrupt data or programs anywhere in a computer's memory. Such a mistake may go unnoticed for a while until the corrupted data or programs are used. When the fault is finally discovered, it is almost impossible to locate its origins and prevent it from happening again.

Address space protection should not be monolithic as different programs require different levels of protection. A well tested routine running as a separate process would only suffer in performance if it had to drag along the same protection mechanisms that are needed for a recently implemented extension to the operating system.

The BiiN™ architecture provides a flexible and efficient protection scheme that addresses this problem. The unit of protection in a BiiN™ system is the object. An object is protected on three levels. (For an illustration, see Figure VII-1-6.)



**Figure VII-1-6. Threefold object protection**

The entire memory of a BiiN™ system is organized in terms of objects. Objects can only be accessed by protected pointers, the access descriptors. An AD contains the information where the object it references is stored. But the AD limits the access to the object by way of *access rights* that are stored in it. Access descriptors are manipulated in controlled ways by the hardware. If a routine attempts to manipulate an AD, such as changing the address or tampering with the rights, the AD will automatically be invalidated. This is the basic protection that applies to all objects in a system.

ADs are given out on a strict “need to know” basis. Any subroutine therefore has access only to the objects that it needs to reference. Thus the set of objects accessible to any one call is strictly controlled. In Figure VII-1-6, this set is represented by the second outermost circle.

Objects are further protected by *typing*. Operations are tied to object types; an implementor defines what operations are permissible. This level of protection is represented in Figure VII-1-6 by the third outermost circle.

Finally the strictest protection is provided by the *type manager model*. A *type manager* is a routine that implements all operations on a certain type. Any routine that wants to perform an operation on the object protected by a type manager has to do so using a call to the object’s

type manager. This mechanism strongly confines any error that may occur in an operation on an object: Only the type manager can physically get to its objects. And only it is responsible for the objects' integrity. This level of protection is represented in Figure VII-1-6 by the innermost circle.

In a BiiN™ system not all levels of protection have to be used at all times. Trusted routines can trade in protection for performance.

### VII-1.3.1 Access Descriptors

Previously, we have characterized the memory of a BiiN™ system as a network of objects and access descriptors as connections in the network. Access descriptors are protected pointers; pointers, because they contain a physical address; protected, because only the BiiN™ Operating System can create ADs. You may even identify an AD with the object because there is no way to get to the object except by AD.

Words on a BiiN™ system are 33 bits long. The 33<sup>rd</sup> bit of every word is a tag bit. If the tag bit is set, the hardware recognizes the word to be an AD. The information in an AD, address and rights together is 32 bits long. Figure VII-1-7 shows an AD.

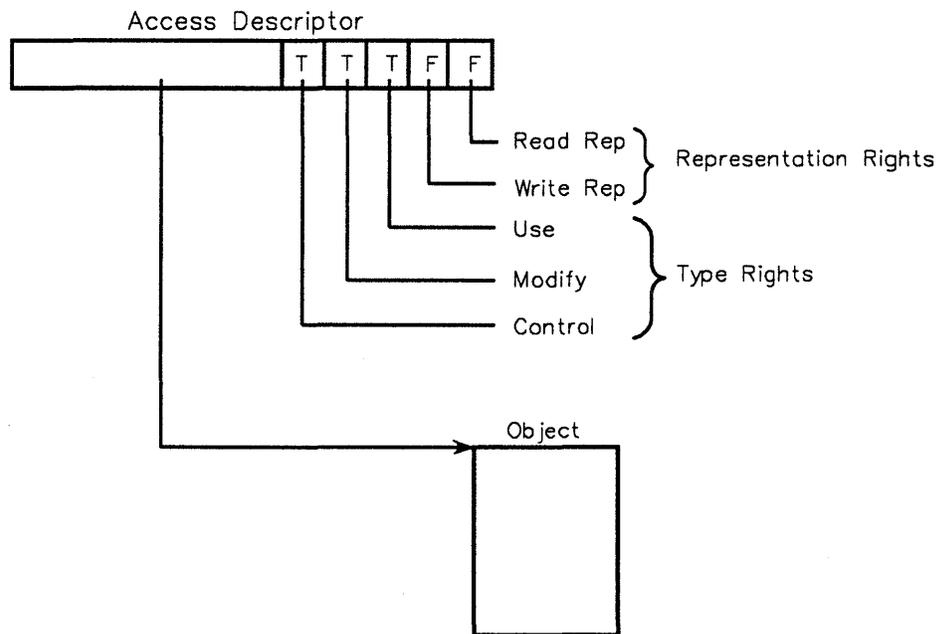


Figure VII-1-7. An Access Descriptor

The first 26 bits contain the object index, then a *local bit* follows, and the next 5 bits are the rights. (There can be  $2^{26}$  different objects on one BiiN™ node at any time.)

There are five rights, three type rights and two representation rights. Type rights, as their name indicates are specific to object types. Their names may vary with the types they apply to. However, there is a naming convention for those three rights: They are called *use*, *modify* and *control*. In the case of a device, they may be renamed to *read*, *write* and *control* and in the case of a directory to *List*, and *Store*. There are no control rights in the case of directories.

Type rights give access to an object's logical structure. For example, if you have modify rights to a file you may write to this file record by record. Representation rights are different. There are *read* and *write* representation rights. They give access to an object's physical layout in memory. In the type manager model no routines are granted representation rights except the type manager. (See Figure VII-1-8)

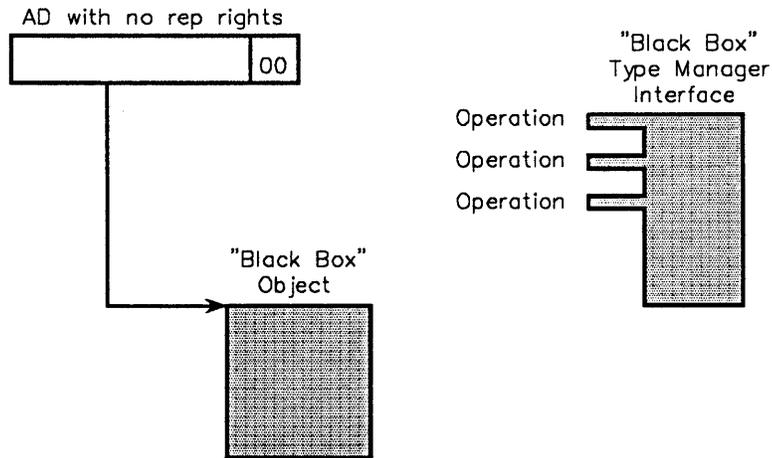


Figure VII-1-8. A Type Manager Makes the Object Appear as a Black Box

It is important to understand the difference between type rights and representation rights. For example, take read rights and read representation rights for a file. A file may have a very complicated layout in memory. It may sometimes be moved around by the operating system and it does not even have to be stored in a contiguous way. Having read rights you would never be aware of the way the file exists in memory. You could read the logical content of the file, however, and you could copy it. Having read representation rights to the file, on the other hand, you could read it bit by bit and find out precisely how it is stored in memory. Here we can go back to our black box analogy; type rights give you access to a black box's front panel. Representation rights are like a mechanic's license. They allow you to take a screwdriver, open up the box, and dig around inside.

### VII-1.3.2 Type Managers

Type Managers provide the strongest protection in a BiiN™ system.

That protection is provided by the following mechanism: Any operation on an object protected by a type manager is a call to the object's type manager. The type manager is the only routine that operates directly on objects of its type: Only the type manager can create new instances of its type and only the type manager can remove those instances.

To use an analogy: In rare book libraries, users are not allowed into the stacks. Type managers act like librarians in such a library. Users of the library fill out request cards, and the librarians bring the books out of the stacks.

Type managers implement two paradigms of the BiiN™ architecture:

- Error confinement

- Independence of implementation details.

A well defined functionality is associated with objects of a given type. This functionality is provided by one module, the type manager. The type manager concept hides implementation details in the the type manager module and confines all errors to that same module.

As a new type is created, the system returns an AD for the type's TDO. That AD has *amplify* and *create* rights. It will be confined to the new type's type manager. A routine may now call the type manager and pass an AD with certain type rights to it. The type manager will use its AD to the TDO as a key and add representation rights to the passed AD. After performing the requested operation, the type manager strips off the representation rights and returns the AD to the calling program. By definition any routine that holds an AD with Create and Amplify rights to a TDO is a type manager for that type. ADs with representation rights should never be passed outside a type manager. There is one exception to this rule; the rule does not apply to *generic objects*.

Generic objects are untyped in the sense that there is no type manager for generic objects. The operating system functions as the type manager for generic objects and gives out ADs *with* representation rights. Generic objects, however, are the only objects for which there are ADs with representation rights outside a type manager.

Generic objects are used whenever an untyped memory segment is needed. Representation rights are needed to write an untyped memory segment.

### VII-1.3.3 Domains

Domains provide protected address space for program execution. A domain is represented by an object of type domain. How a program is split up over different domains is specified at link-time. The modules that make up a program may be linked into separate domains or some or all may be merged into one single domain. When calling a routine in a different domain address space is switched to the called routine's domain. Upon return, address space is switched back to the calling domain. The inter-domain calling mechanism mutually protects caller and callee.

A separate stack may be associated with any set of domains. A set of domains that share one stack is called a subsystem. Subsystems are completely isolated from one another. The address space of a subsystem looks very much like an independent computer all by itself.

Figure VII-1-9 illustrates the details of a domain object.

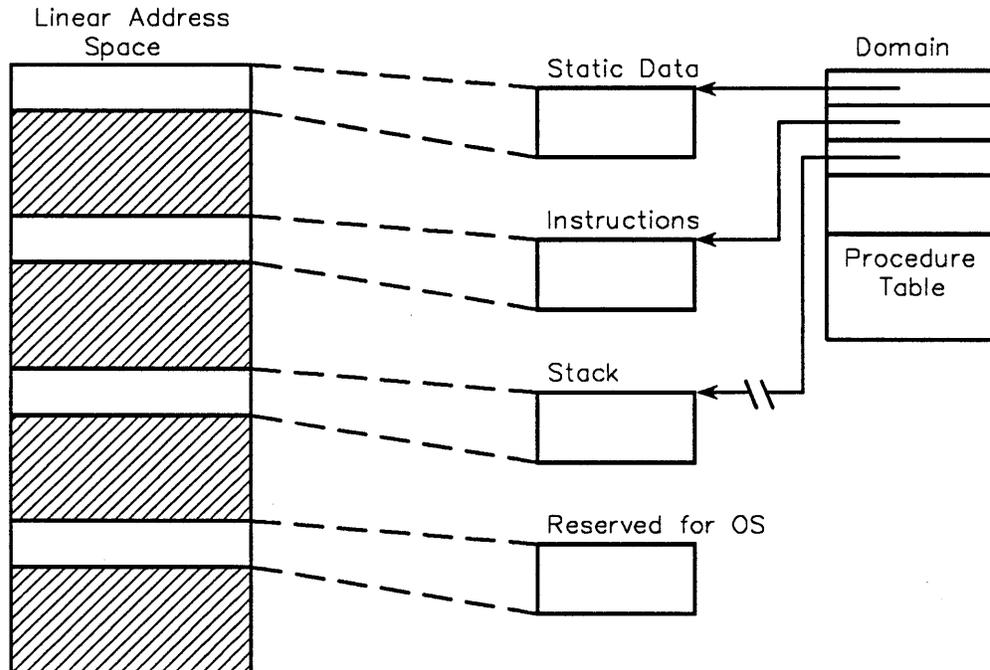


Figure VII-1-9. Linear Address Space and Domain

A domain holds ADs to the static data object, the instruction object, a subsystem ID, and an object reserved for use by the BiiN™ Operating System.

The static data object contains data that cannot be referenced outside the current domain. If a program has only one domain, the static data object contains all variables with global lifetime. The static data object also contains ADs to other domains whose external procedures can be called from this domain.

The instruction object contains the code for all subprograms defined in this domain.

The subsystem ID references a local stack object that contains parameters, local variables and housekeeping information used in subprogram calls. All domains in one *subsystem* and one job share a stack object. If you want to have a process executing with its own stack you have to put the process in its own subsystem.

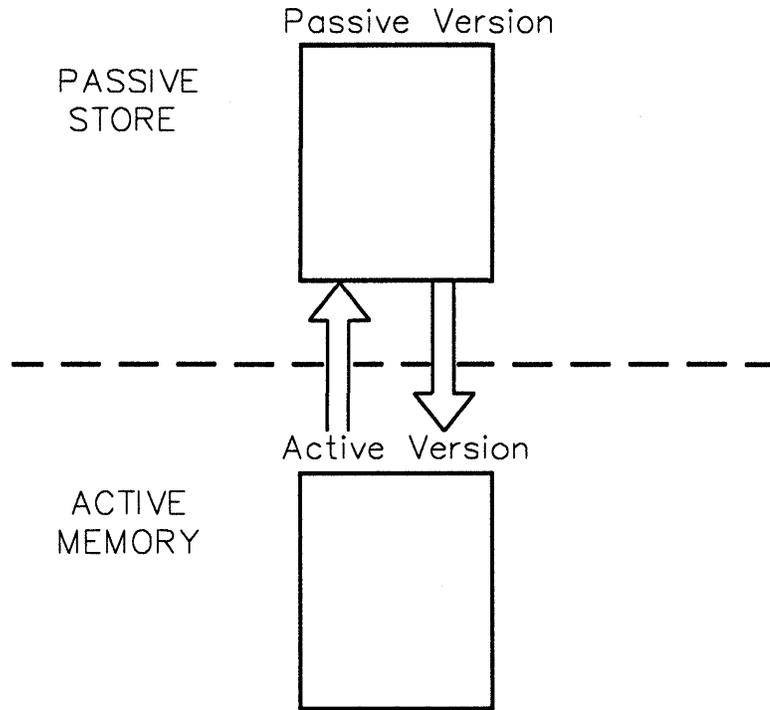
There is a performance penalty attached to inter-domain calls. Only those modules that need the added protection should therefore be linked into separate domains.

## VII-1.4 Passive Objects

We have mentioned before that there can be active and passive versions of an object. Most of our previous discussion applied to active objects. Although passive objects are very similar to active objects, there are a number of differences that you will need to understand. This section explains how objects act as the building blocks of *passive store*, a BiiN™ system's permanent memory.

### VII-1.4.1 Active Memory

*Active memory* is the collection of objects in virtual memory on a particular BiiN™ node. An object can have versions in both active memory and passive store (Figure VII-1-10).



**Figure VII-1-10. An Object's Active and Passive Version**

Only active versions can be directly read or written. Reading or writing an object with no active version causes the object to be *activated*. Objects are activated on demand, transparently, just as pages of virtual memory are swapped in when needed. Both operations are invisible to your application. Changing an object's active version does not change the object's passive version. An explicit *update* call is needed to copy an object's active version to its passive version.

### VII-1.4.2 Passive Store

While active memory is entirely part of one BiiN™ node, passive store is completely distributed in a BiiN™ system. Passive store is the glue that holds a distributed BiiN™ system together. (See Figure VII-1-11)

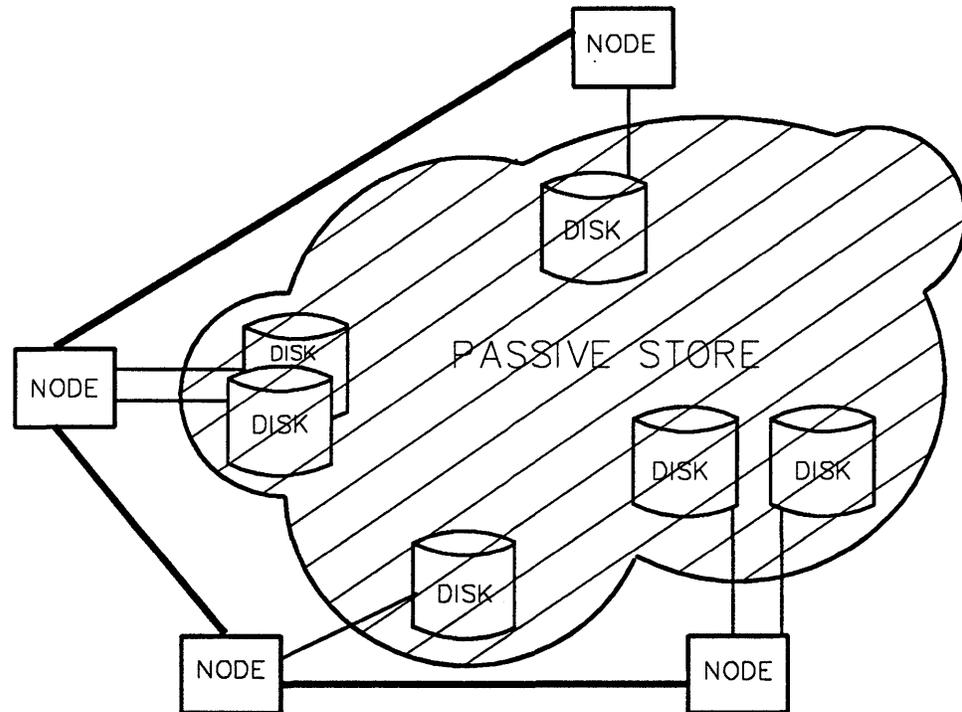


Figure VII-1-11. Passive Store Unifies All Nodes in a BiiN™ System.

Passive store wraps around an indefinite number of disks in a distributed BiiN™ system. Logically it is divided up into *volume sets*. Volume sets are associated with individual nodes. However, that association is transparent to the user.

### VII-1.4.3 Passive ADs

When an object is first stored, passive store creates a passive AD for the object. A passive AD is a much bigger entity than an active AD. The reason is that a passive AD is a unique reference on an entire distributed system, while an active AD is valid only on a particular BiiN™ node.

Whenever an AD crosses the boundary between active and passive store or between different nodes of a distributed system, it has to be converted from its active to its passive form.

Just as there can be multiple active ADs to one object, there may be more than one passive AD to an object. (There may also be active ADs to passive objects.) One of the passive ADs is the *master AD*. All other passive ADs are called *alias ADs*. The master AD plays a crucial role. An object cannot be stored until a master AD exists. If there is no longer any master AD for an object that object will be removed. There are the following exceptions to that rule:

- If the master AD is stored in a directory and other directory entries on the same volume set reference the object. One of these alias ADs then becomes the new master AD.
- If the master AD is stored in another object and other ADs in that object reference the object. One of those alias ADs then becomes the new master AD.

#### VII-1.4.4 Passive Store Protection -- Authority Lists

Naming of and references to passive objects are slightly different than for active objects. The reason for this is simple: An AD once given out is irrevocable. That means that rights once granted by giving out an appropriate AD cannot be taken back. Generally this poses no problem in active memory since usually active objects only exist for short time periods. Objects on disk, however, exist indefinitely.

The model for protecting objects in passive store is different from the address space protection provided by ADs in active memory. Protection requirements are different for passive objects than for active objects.

In active memory a program should execute as much as possible in a secluded cell. Thus the segment of memory that can be affected by an erring program is kept to a minimum size.

This protection philosophy is inadequate for passive store for two reasons.

- Passive store is distributed. The view that any one job has of passive store should as wide as possible without opening up protection holes.
- Objects in passive store exist indefinitely. Information of who may access an object stays with the object. This allows the owner of the object to alter access over the lifetime of the object. (The philosophy behind active memory protection is to attach the information of who may access an object not to the object but to the requesting job. In this model it is difficult to revoke access once it has been granted.)

The difference explained in the second point above can be likened to the difference between a key lock and a combination lock. A key will always open the key lock just as an AD will always grant access to its object. But a combination can be made invalid when the lock is reset.

The protection provided for stored objects is based on the concept of an authority list. An authority list consists of *<ID, Type Rights>* pairs. When an object is first stored, an authority list can be specified by the storing process. If no authority list is given, the object will receive the default authority list of the directory in which it is stored. If there is no default authority list for the directory, the object receives the storing process's default authority list defined in the process globals. A passive object may also have no authority list.

An authority list is a vehicle for granting access to different users, user groups and programs. The owner can grant or revoke access at any time by specifying a new authority list. (Figure VII-1-12 shows how authority lists fit into the organization of passive store.)

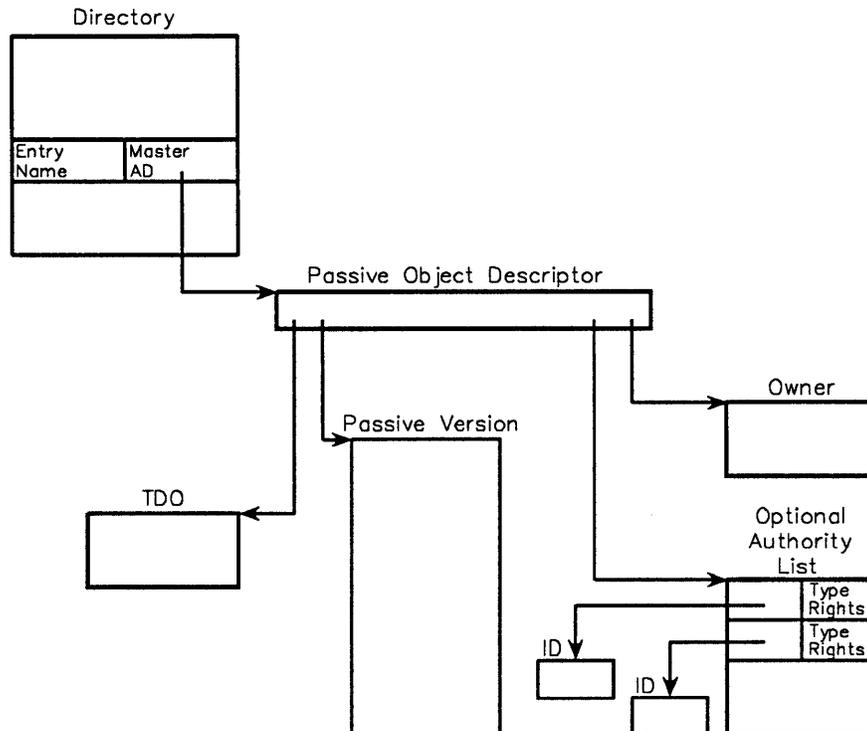


Figure VII-1-12. A Stored Object

Authority lists define access in two operations and for both in slightly different ways: Firstly, when a passive object is explicitly retrieved, the retrieving job's list of IDs is compared to the authority list and an AD is returned with the combined rights of all matching IDs. Secondly, when an AD is transparently activated, the activating process's ID list is checked against the authority list of the container and against the authority list associated with the AD proper. This ensures that stored ADs cannot be activated unless their rights are current. Should rights have been revoked since the AD was given out, the AD will lose those rights when it is activated. Note that an object's owner always has access to the object even if his ID does not appear in the authority list. For more details, see Chapter III-3.

#### VII-1.4.5 IDs

As you have seen in the previous section IDs are central to the protection concept used for passive store. It is therefore necessary to tell some more details about IDs.

IDs are maintained centrally in a BiiN™ system, namely in the *Clearinghouse*. To get back to our previous example of the two different locks: Each ID is like the combination for a combination lock. (The analogy is a little bit weak at this point since combination locks usually only have one combination. Let's however disregard this for the moment and assume that there are combination locks that open by more than one combination.)

As IDs are the keys to stored objects, they in turn have to be protected. This is achieved by way of *protection sets* and passwords. Protection sets are similar to authority lists. They consist of <ID, Rights> pairs. The two rights defined for IDs are *portray* and *control*. The portray right grants the holder permission to add this ID to an *ID list*. Control rights allow the holder

to alter the password on an ID. By specifying the proper password, one can obtain an AD to an ID with portray rights.

### VII-1.4.6 Updating Stored Objects

Most calls to passive store are *transaction-oriented*. In particular, updates on stored objects can be included in a transaction. (A transaction ensures that all the operations included in it are executed as a unit: Either all the operations inside a transaction will be executed or none of them.) With the help of a transaction, you can prevent incomplete updates. Including calls to passive store in a transaction also prevents clashes between multiple jobs attempting an operation on the object. While the older of two transactions executes, it reserves the object. The younger transaction simply waits until the older one finishes.

Another problem arises when multiple active versions of an object exist. An obsolete active version could be used to update the passive version. Two situations can arise:

*Multiple Activation Model:*

There are multiple active versions of a passive object. Passive store keeps track of all active versions and refuses updates from obsolete versions.

*Single Activation Model:*

A *single activation object* is only activated in one *home job*. Other jobs that activate the object receive a token active version of the object called *homomorph*. Jobs that want to update the object have to communicate with the home job. For all operations on the object the job communicates with the home job of the object.

Both models are supported by the BiiN™ system. Depending on the needs of an application, the programmer can decide which one to use. In this context it is only important to note how updates are handled in these two models.

## VII-1.5 Summary

After having read this chapter you should understand the following concepts:

- All information in a BiiN™ system is contained in objects.
- Objects are typed and protected memory segments.
- Objects are the unit of protection.
- Access descriptors are protected pointers. Objects can only be accessed with access descriptors.
- Objects can be dynamically allocated, resized, and destroyed.
- Objects may “know” what operations can be performed on them and how.
- Objects can have passive and active versions.
- Objects can be local to a job or global to a particular node.
- Passive objects are uniquely identified on all BiiN™ nodes and for all time.
- Access descriptors can pass freely between the nodes of a BiiN™ system.

If you understand all these concepts, you can go on to the next chapter which explains memory management.

# UNDERSTANDING MEMORY MANAGEMENT **2**

---

## Contents

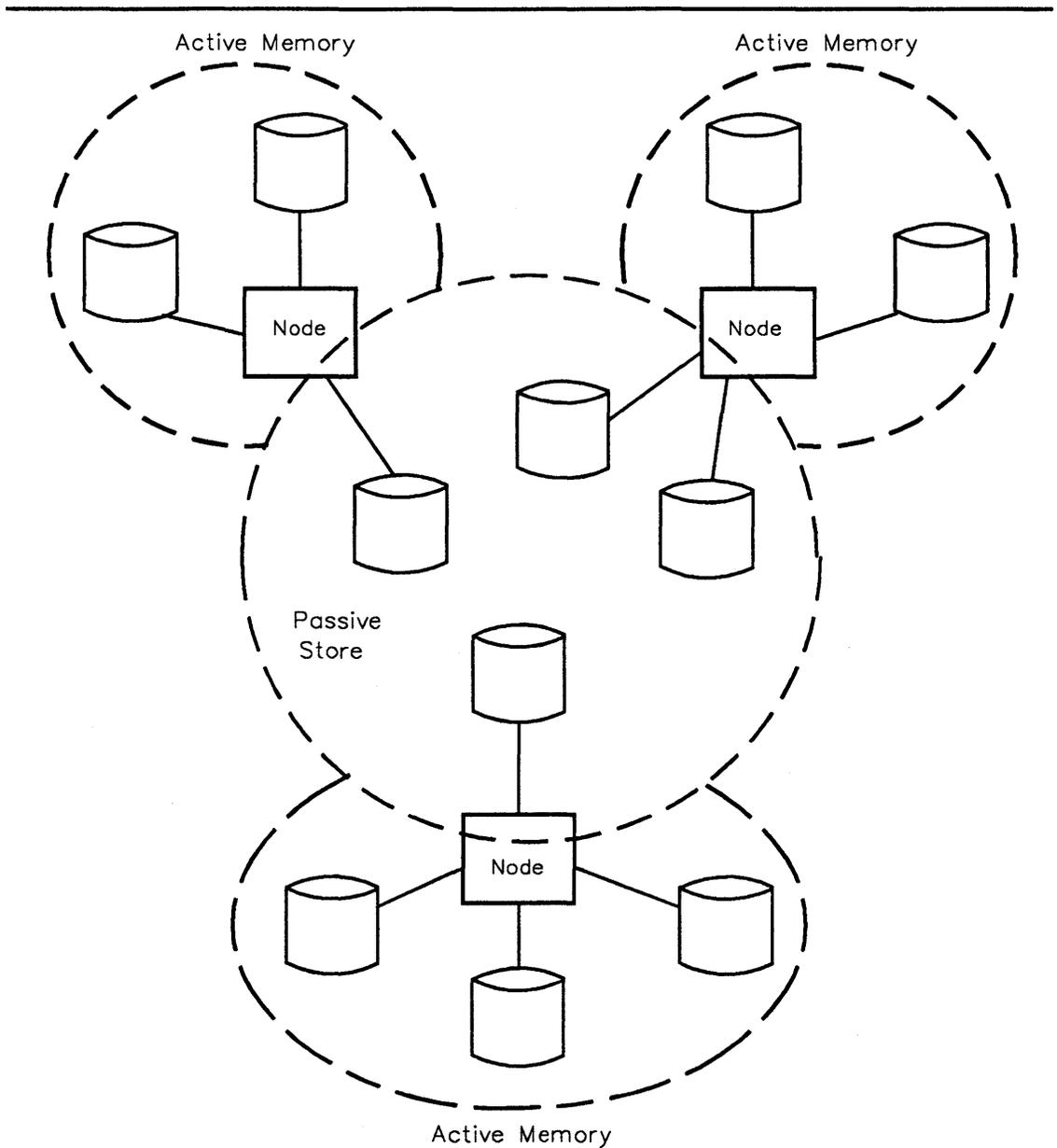
Physical Memory Organization .....	VII-2-2
Virtual Memory Organization .....	VII-2-5
The Object Table .....	VII-2-5
Object-Based Address Translation .....	VII-2-7
Storage Resource Object .....	VII-2-7
Object Representations .....	VII-2-8
Frozen and Normal Memory Types .....	VII-2-9
Different Allocation Policies .....	VII-2-9
Object Lifetimes .....	VII-2-9
Object Deallocation Strategies .....	VII-2-10
Controlling and Accounting for Memory Resources .....	VII-2-12
Object Activation .....	VII-2-12
Virtual Memory Paging .....	VII-2-12
Global Garbage Collection .....	VII-2-13
Compaction .....	VII-2-13
Optimized Handling of Instruction Objects .....	VII-2-13
User-Transparent Memory Management Functions .....	VII-2-12
Summary .....	VII-2-13

Objects are abstract constructs. Just as you cannot understand the concept of an automobile by studying metallurgy, you cannot understand objects by looking at their representation in memory. However, if you want to design a car, you will probably have to understand some metallurgy. Similarly, you will have to understand how memory is managed in a BiiN™ node if you are going to do some system programming, because objects are “made out of memory”.

This chapter describes how a BiiN™ node manages its memory. It covers the underlying concepts of *virtual memory* and of the allocation and deallocation of objects. It discusses how objects are laid out in memory, when they can be moved around by the system and when not. And finally, it shows the forms of addresses in a BiiN™ system and how they are resolved. This chapter does not give a detailed description of passive store. However, where passive store concepts are relevant to active memory management, they will be explained briefly. This chapter builds on the previous chapter (Chapter VII-1). You should either read that chapter or have a good understanding of objects and how they function in the BiiN™ architecture, before reading this chapter.

## VII-2.1 Physical Memory Organization

Physical memory consists of a node’s RAM and all disks that are mounted on the node. Physical memory is divided into *active memory* and *passive store*. Figure VII-2-1 shows how memory is organized in a BiiN™ system.



**Figure VII-2-1. The Organization of Memory in a BiiN™ system**

Active memory, as its name indicates, is the immediate “working space” of the processor. Active memory is also volatile. Its contents are lost whenever the system is turned off. Passive store on the other hand is permanent storage. Its contents cannot be lost unless a disk is damaged. (See Figure VII-2-2.)

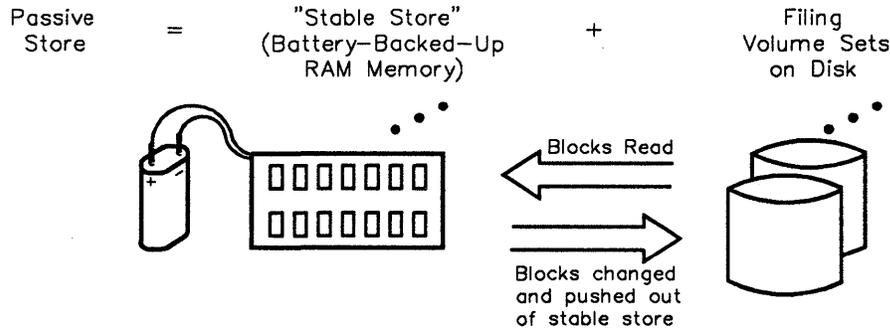


Figure VII-2-2. Passive Store

The memory pool on all disks of a node is partitioned into volume sets. Volume sets in turn consist of from 1 to 254 volumes. A volume set can span multiple disks. A single volume always resides on one particular disk. However, there can be more than one volume on a single disk. A volume set can be either a *swapping volume set* in which case it is part of the active memory, or a *filing volume set* and part of passive store. Swapping volume sets are invisible to the user. They appear as part of active memory, and from a user's point of view, the memory in a swapping volume set looks identical to the RAM.

The physical memory that underlies all other memory is partitioned into 4K byte page frames. Each page frame is uniquely identified by a page number. (See Figure VII-2-3.) A page frame is simply an empty page. A page is the unit of abstraction of memory management. The smallest unit that memory management recognizes is 64 bytes.

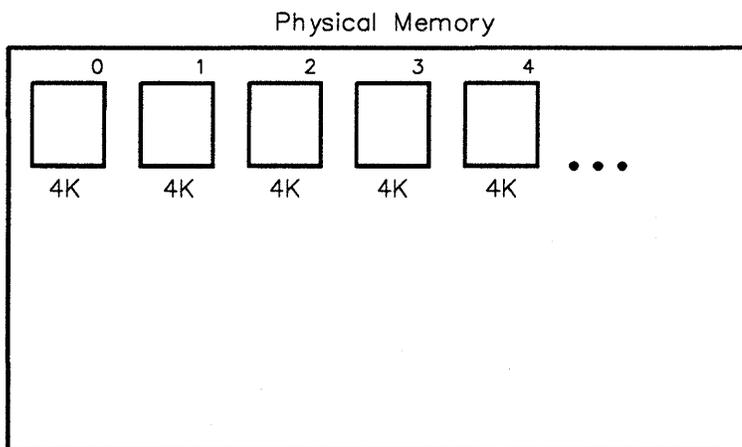
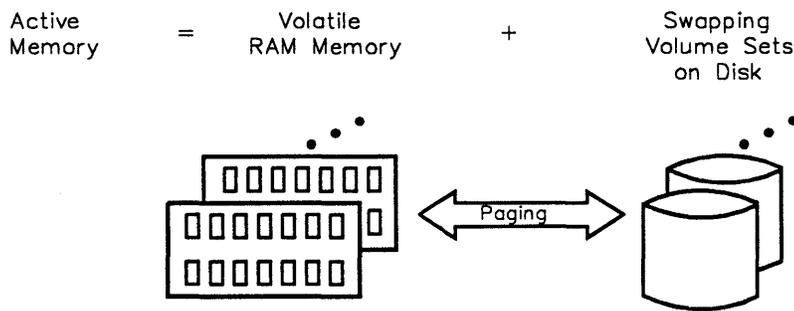


Figure VII-2-3. Physical Memory is Divided into Pages

Private to memory management is a central *page frame table* (PFT) where information about the contents of all page frames is stored. Since a single page frame may contain different information as time progresses, the contents of the *page frame table entry* will change as well. (There is a parallel here between physical and logical memory organization: Object table and page frame table and object descriptor (object table entry) and page frame table entry play similar roles. An important difference between the two is that the object table is recognized by the hardware, while the page frame table is purely a software concept.)

## VII-2.2 Virtual Memory Organization

Active memory is organized according to the *virtual memory concept*. This means, the part of memory that is directly accessible to the node may span parts or all of the node's RAM and mass storage devices such as disk drives as well. The processor's total physical address space is  $2^{32}$  bytes. (That is about 4G bytes.) (See Figure VII-2-4.) The total virtual address space permissible is  $2^{58}$  bytes, consisting of  $2^{26}$  objects and  $2^{32}$  bytes per object. The virtual memory concept frees the system from the limitations imposed by relatively scarce primary memory.



**Figure VII-2-4. Active Memory Uses Both RAM and Disk.**

---

Virtual memory management takes advantage of the fact that the entire address space of the node is not used simultaneously at all times. The processor can only directly address pages that are available in RAM. This part of memory is called *primary memory*. Memory management moves pages in and out of primary memory in such a way that the user has the illusion that all the information is contained in primary memory. Pages are swapped in as they are referenced and swapped out when they are no longer needed. A page is either *accessible* or not. If the page is accessible, it means, the page resides in primary memory and the process can get to it directly. If the page is not accessible, memory management retrieves it from its location in *secondary memory* (on disk, in the swapping volume set) and places it in primary memory.

There is a *common page pool* that is a list of free pages in primary memory. When a job requests space in RAM, pages from the common page pool are allocated to it. When a page that is not altered is returned to the common page pool, then, if a process references the page, it can be reclaimed from the pool, thereby avoiding a swap-in. In essence, the common page pool represents a cache of pages in the swapping volume set. If a page is not available in the common page pool, it is swapped in from disk. That means, its contents is copied into a newly allocated page frame.

### VII-2.2.1 The Object Table

Physical memory is organized in terms of pages. On the other hand logical organization of memory is in terms of objects. The page frame table (PFT) centralizes important information about pages. Analogous to the PFT in the organization of physical memory is the object table in the logical organization of memory. (The object table is a hardware defined and hardware recognized data structure, while the page frame table is a purely software defined data structure.) The PFT consists of page frame table entries, and the object table consists of object descriptors. (See Figure VII-2-6.)

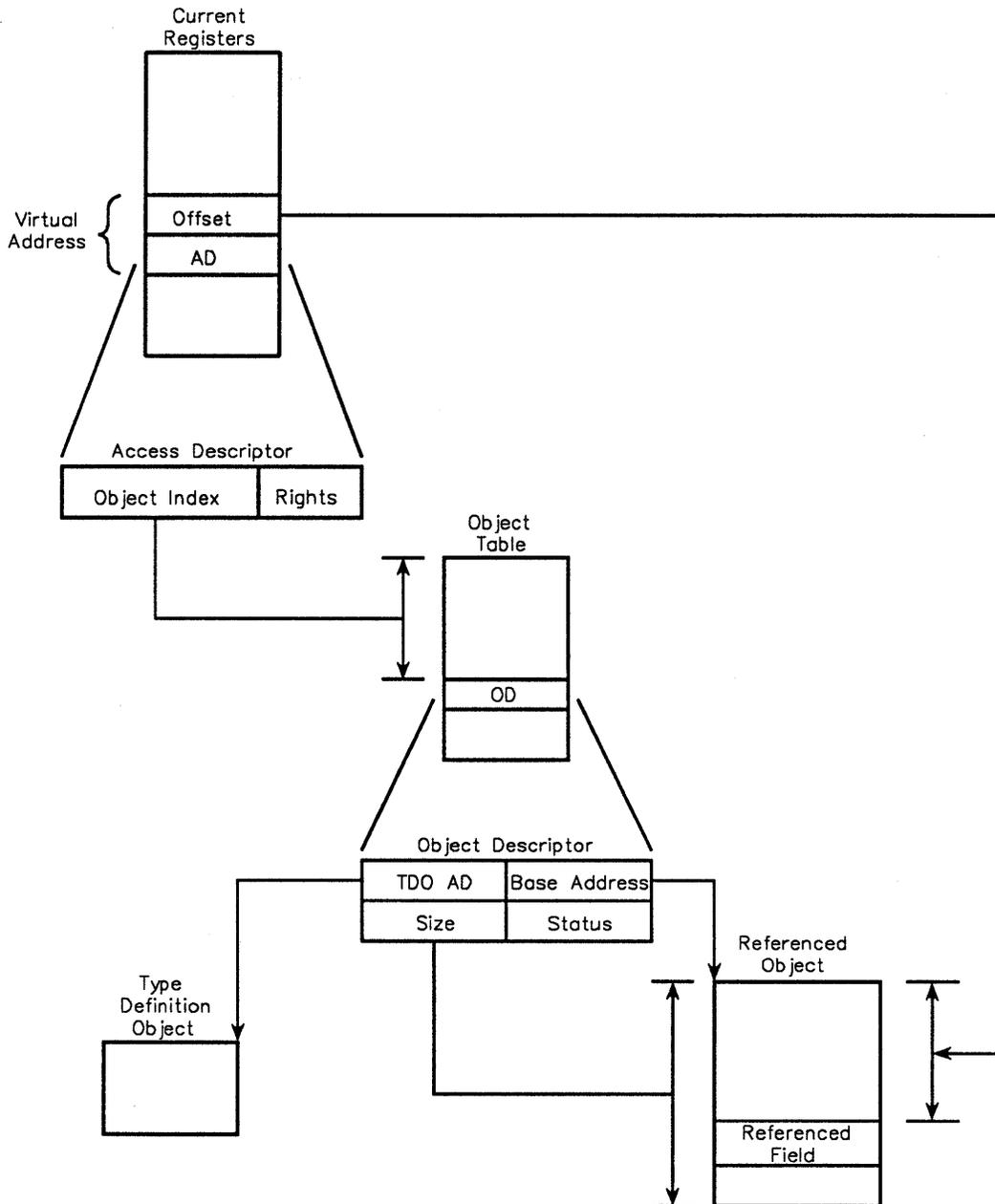


Figure VII-2-5. The Object Table and Object Based Address Translation

Objects can only be referenced by access descriptors (ADs). There can be a multitude of ADs to any single object. It is necessary to have one single place where important information about the object is stored, such as its physical address. Otherwise all ADs to the object would have to be updated if some of the information changes. For this reason, there is exactly one object table per node.

### VII-2.2.2 Object-Based Address Translation

Figure VII-2-5 also illustrates the addressing mechanism. The BiiN™ system recognizes two types of addresses, *linear* and *virtual* addresses. Linear addressing is faster than virtual addressing, but is restricted to a single domain. Linear addresses are used for programs that execute entirely inside a linear address space. This would typically be the case with FORTRAN and Pascal programs. In order to access arbitrary objects in the system you *have* to use virtual addresses. Figure VII-2-6 shows a valid virtual address.

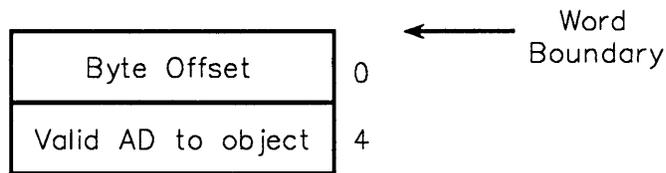


Figure VII-2-6. A Valid Virtual Address

---

Virtual addressing is an object-based addressing scheme. Figure VII-2-5 illustrates the virtual addressing scheme. A virtual address consists of two parts, an AD to the object that contains the field that you want to access, and an offset into the object that specifies where the field is located inside the object. A linear address is an offset by itself, without an AD.

As mentioned previously, the AD does not reference the object directly but rather it refers to the object descriptor in the object table. The object descriptor holds the physical address of the object.

### VII-2.2.3 Storage Resource Object

There is one *storage resource object* (SRO) associated with each job. It represents a pool of storage local to the job and all its processes. When an SRO is first created, a certain *storage claim* is assigned to it. As storage is allocated from the SRO the storage claim is debited, and if storage that had been allocated from the SRO is deallocated, the claim is credited with the proper amount. A job's local SRO is a global object which is removed once its controlling job terminates. In addition to local SROs there are two global SROs for each BiiN™ node, one controlling *normal memory* allocation and the other one controlling *frozen memory* allocation. Global SROs can only be referenced by administrative users and trusted type managers. Global SROs have unlimited storage claims. SROs are *active-only* objects: That means that SROs cannot be passivated. (For a discussion of normal and frozen memory, see section VII-2.2.5.) Figure VII-2-7 illustrates SROs in a node's virtual memory.

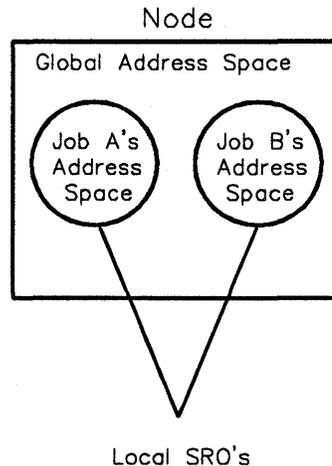


Figure VII-2-7. Active Virtual Memory, Jobs, Nodes and SROs

#### VII-2.2.4 Object Representations

An object's representation is an area in virtual memory that holds the contents of the object. An object's representation has a certain size that can range from 0 to  $2^{32}$  bytes. However, object sizes are rounded depending on the size of the object:

1. If  $\text{size} = 0$  bytes, or if the object is a *semaphore*, then the object's representation is entirely contained within the object descriptor. These objects are called *embedded objects*.
2. If  $0 < \text{size} \leq 4\text{K}$  bytes, then  $\text{size}$  is rounded up to the next multiple of 64 bytes. These objects are called *simple objects*.
3. If  $4\text{K} < \text{size} \leq 4\text{M}$  bytes, then  $\text{size}$  is rounded up to the next multiple of 4K bytes. These objects are called *paged objects*.
4. If  $4\text{M} < \text{size} \leq 4\text{G}$  bytes, then  $\text{size}$  is rounded up to the next multiple of 4M bytes. These objects are called *bipaged objects*.

The reason for the rounding outlined above stems from the paged structure of the underlying physical memory. The following paragraph outlines the mechanism. For more details refer to *BiiN™ Systems CPU Architecture Reference Manual*.

Simple objects can share a page frame with other simple objects. If an object's size is equal to 4K bytes, it will occupy a page all by itself. In the case of a paged object the object descriptor references a *page table* (PT). A page table is simply a list of all pages that are part of the object's representation. The page table is located on a page frame itself, possibly together with other object's page tables. If a paged object's size is equal to 4M bytes, the page table will occupy an entire page by itself. The object descriptor of a bipaged object references a *page table directory* (PTD). This is a list of page tables which in turn are lists of page frames. Instead of having one very long page table there are two levels of page tables (hence the name bipaged objects) -- many 4K page tables, and one level up, a table of those page tables. In the extreme case of an object occupying 4G bytes, the page table directory itself occupies an entire page.

The object table is a paged or bipaged object. It is handed out in units of single pages which can contain up to 256 object descriptors. Whenever possible, the object table is kept down to a paged object to keep down address translation times. Only when necessary will the object table become bipaged.

### VII-2.2.5 Frozen and Normal Memory Types

In certain cases, such as real-time or time-critical applications the virtual memory mechanism of swapping pages in and out of primary memory may cost too much time. Upon request, a job can run in *frozen memory*. The job's SRO will then allocate objects that will not be moved between primary and secondary memory but will reside entirely within primary memory. A local SRO that has a *frozen memory type* has an infinite storage claim. The designer of the application will have to take care that there is sufficient primary memory to run the program. Furthermore, in order for all pages to be allocated before the program runs, the user must have *allocate-on-creation* rights for the SRO.

Most other programs will run in *normal memory*. They have an SRO with a *normal memory type*. The SRO then has a given fixed storage claim.

### VII-2.3 Different Allocation Policies

Two policies are used when paged objects are allocated in primary memory. The standard policy for SROs with a normal memory type is *allocate-on-reference*: First, only the page table directory is allocated for a bipaged object and the page table of a simply paged object. Second level page tables of bipaged objects and pages of paged objects are physically allocated in memory only when they are directly referenced.

The second policy, called *allocate-on-creation*, is reserved for SROs with frozen memory type. The SRO also needs to have *allocate-on-creation-rights*. Allocate-on-creation can be explicitly enabled and disabled for such an SRO. If an SRO with allocate-on-creation enabled allocates an object, the entire representation of the object will be allocated. This technique is useful for time-critical and real-time applications.

### VII-2.4 Object Lifetimes

There are *local* and *global* objects in the BiiN™ system. Local objects are local to a particular job. That means that the active version of a local object is removed when the controlling job finishes.

A local object can however be passivated, and the passive version will survive when the controlling job finishes. When the passive version is again activated, its active version will again be a local object and will automatically disappear, once the job that activated the object finishes. A local object that has never been passivated will disappear completely once its controlling job finishes. Global objects exist outside any particular jobs. There are two types of global objects, *unbounded* global objects and *countable* global objects.

An unbounded global object's active version can exist indefinitely, or more precisely, until it is explicitly removed by global garbage collection. Global objects can also be passivated and thus survive system crashes and explicit garbage collection.

Countable global objects behave very much like unbounded global objects. However, unbounded global objects have one distinct disadvantage that countable global objects avoid: Unbounded global objects can only be removed by global garbage collection. Global garbage collection is a very expensive process because it may involve extensive disk traffic. It is desirable that it not be used too often. Countable global objects can be deallocated without global garbage collection. This is done with the following technique.

For countable global objects, there is a mechanism that keeps track of all references to a particular object. Whenever an AD is given out to a job for the first time, the reference count is incremented by one. Also, whenever a job terminates that held an AD to the countable global object, the reference count is decremented by one. If the reference count equals zero, object management is notified and then removes the object. Note that the reference count keeps track of how many jobs hold references to the object, not how many ADs have been given out. A job can also *logically delete* its AD to an object. The job then continues to run but forfeits its access to the particular object. This causes the count of logically deleted references to be incremented. When the count of logically deleted references is equal to the reference count, deletion of the object also results. The BiiN™ Operating System and the hardware work together to prevent lifetime violations.

ADs can also be local and global. On the simplest level, this means, ADs to a local object will always be local ADs. If this were not so, global ADs to a local object could outlive the object. For that same reason local ADs are confined to one job. Global objects can have local and global ADs. Countable global objects, however, have only local ADs. This ensures that all ADs that belong to one job will disappear once the job terminates.

## VII-2.5 Object Deallocation Strategies

There are various ways of removing, or deallocating, objects that are no longer needed. This is an important task. Without it, memory would be exhausted in a very short time period. The way objects are deallocated depends on the object and on the needs of the job that uses them. In particular, there are these methods for deallocating objects:

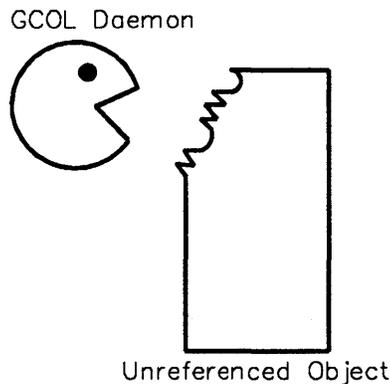
- Explicit Deallocation
- Local Garbage Collection
- Global Garbage Collection
- Reference Counting
- Deallocating Passive Versions.
- Job Termination

Explicit deallocation (using `Object_Mgt.Deallocate`) is the simplest, most direct method to remove an object. It is used whenever a job “knows” that an object that it has created is no longer needed. Note, however, that such deallocation removes only the object’s active representation. The object descriptor will still be there. If an AD is used to access an object whose representation has been deallocated and which has no passive version, the exception `System_Exceptions.object_has_no_representation` is raised. If there exists a passive version of the object, it is transparently activated. Note, however, that when you deallocate an object’s representation, the object’s passive version is not updated automatically. If you want to save any changes on the object, you have to specifically update the passive version.

## PRELIMINARY

There is an operation available to trusted routines called `Unsafe_Object_Mgt.Unsafe_deallocate`. This operation removes not only the object's representation but the object descriptor as well. This operation is unsafe because if there are any ADs to the object after the object has been completely removed from the system, a use of this AD will result in a dangling reference. A routine that uses `Unsafe_deallocate` has to ensure that there are no ADs left to the object outside the routine itself. Failure to do so can cause fatal system behavior.

Local objects for which there are no more ADs can be reclaimed by local garbage collection. The purpose of local garbage collection is to enable long-running jobs to periodically clean up their address spaces. Garbage collection can be started and then runs as a daemon. When run as a daemon it will wake up periodically whenever the storage claim of the job falls below a certain adjustable percentage. A minimum delay between runs of the garbage collector (GCOL) can also be specified. This is to prevent GCOL from running permanently when a job's storage claim becomes low.



**Figure VII-2-8. Garbage Collector**

---

GCOL finds each object with no reference and labels it as garbage. It then starts to remove these objects. Differently from an explicit `Deallocate`, GCOL also removes an object's object descriptor. It can do so because it has previously made sure that no ADs to the object exist.

When a job finishes all objects local to the job are removed completely, representation, local ADs, and object descriptors.

Besides the local garbage collection, there is also a global garbage collection mechanism. Global garbage collection works for global objects the same way local garbage collection works for local objects. Global garbage collection is invoked periodically by the system and removes all unreferenced objects. Global garbage collection is an expensive process: It may involve a lot of disk traffic. Therefore, global garbage collection should run as infrequently as possible.

As mentioned previously, countable global objects can be removed without the overhead of garbage collection.

## VII-2.6 Controlling and Accounting for Memory Resources

Jobs are dispatched to the processor by a scheduler. The scheduler recognizes four different classes of jobs: *batch*, *interactive*, *time-critical* and *real-time*. What class a particular job belongs to, depends on what SRO the user specifies when the job is started. (A user has to have the necessary rights to an SRO in order to run a job from it.) Depending on the type of the job, a storage claim of a certain size is defined in the job's SRO by the scheduler.

When an object is allocated from an SRO, the job's storage claim is charged. Accounting is done for the number of object descriptors allocated from the SRO and for the size of the representation of the object. If a local SRO gets to the bottom of its claim, local garbage collection is automatically invoked. In most cases this will result in enough memory space being reclaimed to be able to satisfy the job's allocation request. However, if the garbage collection cannot reclaim enough space to handle the job's allocation request, the job is terminated with a message that states that resources have been exhausted. Accounting is done on a per job and per node basis.

In addition, the class of a job has a more subtle influence on memory allocation than just setting upper limits on the allowed space. In particular, it specifies whether a job is subject to virtual memory paging or not. In the extreme case, a job can run in frozen memory. That means, all of its virtual memory is primary memory. Thus all the job's objects are immediately accessible without swapping pages. This increases performance considerably.

## VII-2.7 User-Transparent Memory Management Functions

Most of the functions of memory management are executed transparently to the user. In particular this includes the following:

- Object Activation
- Virtual Memory Paging
- Global Garbage Collection
- Compaction
- Optimized Handling of Instruction Objects.

### VII-2.7.1 Object Activation

This section describes the mechanism behind transparent object activation. Typically, an object's representation is deallocated and a process holds an AD to the object. When the process touches the object, the BiiN™ Operating System finds that the object has no representation. At that point it attempts to find the object in passive store. If it succeeds, the passive version is copied into active memory and becomes directly available to the requesting process. Otherwise, activation fails.

### VII-2.7.2 Virtual Memory Paging

The virtual memory concept solves the problem that primary memory is scarce. A large part of virtual memory is secondary memory; that is disk. When a process touches a page that is presently held in secondary memory it will be swapped into primary memory. Secondary memory that is part of virtual memory is called *swapping memory*. Swapping memory is divided into volume sets, just as passive store. Swapping pages between swapping volume sets

and primary memory is invisible to the requesting processes. Extensive page swapping, however, slows down program execution. For that reason real-time jobs have all their memory requirements satisfied in primary memory. (In this case the programmer has to make sure that there is enough primary memory available to satisfy the job's demands.)

### VII-2.7.3 Global Garbage Collection

The system periodically invokes a global garbage collector daemon. The daemon is responsible for cleaning up a node's global memory. It removes all global objects for which no AD exists on that node. Garbage collection runs in the background and is invisible to the user. Global garbage collection involves a great amount of overhead. This is because the objects that garbage collection is looking for are unreferenced objects. Objects that have not been referenced in a while tend to move to secondary memory. Finding all those objects and removing them involves a lot of disk traffic. Remember also that garbage collection has to search all objects on a node for references.

### VII-2.7.4 Compaction

The representation of a simple object usually takes up less than one page of memory (4K bytes). When pages are swapped out, compaction is transparently invoked. Compaction takes simple objects and optimizes memory use by placing multiple simple objects on one memory page. Swapping always happens page by page. When a user requests a simple object that is presently on a swapping volume set and shares a page with other simple objects, the entire page that holds the object is swapped in.

### VII-2.7.5 Optimized Handling of Instruction Objects

As their name indicates instruction objects hold processor instructions and constants necessary for program execution. Program execution is optimized in three ways:

- Pages of instruction objects are directly paged in from the file. You do not need to explicitly activate (or load) the instruction object.
- The representation of a (local multiple activation) instruction object is physically shared by all jobs using it whenever possible. This avoids having multiple identical copies in active memory.
- When a job terminates, pages of the instruction object may remain reclaimable for some time. That means, another job that runs later and uses the same instructions can reclaim those pages without having to copy them from disk.

## VII-2.8 Summary

After having read this chapter you should now have a basic understanding of how active memory is managed in a BiiN™ node. In particular, you should have grasped the following concepts:

- Physical memory organization
- Virtual memory
- The object table
- Storage resource object

## PRELIMINARY

- Objects representation
- Granularity of object sizes
- Memory types
- Object allocation
- Object lifetimes
- Object deallocation
- Control of memory resources
- Transparent memory functions
- Addressing

# BUILDING A TYPE MANAGER **3**

---

## Contents

Concepts .....	VII-3-2
The Type Manager Defines All Calls for a Type of Object .....	VII-3-2
Type Managers Hide Data Representation .....	VII-3-3
Only the Type Manager Has the Key to Access the Type's Objects .....	VII-3-3
One Module Can Manage Multiple Types .....	VII-3-3
Techniques .....	VII-3-3
Defining the Public Type .....	VII-3-4
Defining Type Rights .....	VII-3-5
Defining Exceptions .....	VII-3-6
Defining the Type's Calls .....	VII-3-6
Defining the Private Types .....	VII-3-7
Defining Needed BiiN™ Ada Type Overlays .....	VII-3-7
Creating the TDO .....	VII-3-8
Binding to a Stored TDO .....	VII-3-8
Implementing the Is_account Call .....	VII-3-8
Implementing the Create_account Call .....	VII-3-9
Implementing the Create_stored_account Call .....	VII-3-9
Implementing Calls that Require Type Rights .....	VII-3-10
Implementing Calls that Do not Require Type Rights .....	VII-3-11
Implementing the Destroy Call .....	VII-3-11
Making Operations Atomic .....	VII-3-12
Initializing the Type Manager .....	VII-3-13
Protecting the Type Manager from Other Services .....	VII-3-14
Summary .....	VII-3-15

A *type manager* is a program module that defines a particular object type and all calls for objects of that type. This chapter shows you how to build a type manager.

#### Packages Used:

Access\_Mgt      Interface for checking or changing rights.  
Object\_Mgt      Provides basic calls for objects.

The example for this chapter, `Account_Mgt_Ex`, is a simple, general-purpose type manager written as a Ada package. The complete listing of this example can be found in Appendix X-A.

## VII-3.1 Concepts

A type manager provides both data abstraction and protection for the objects of its type. It does so by defining all calls for its objects. No operations but the ones defined by the type manager are possible on the objects protected by it. It is therefore important that you provide all necessary calls when building your type manager.

The type manager holds a key that allows it to create objects of its type and to add representation rights to ADs that are handed to it by calling programs. The key is an AD to the TDO with *amplify* and *create* rights. It is given out when the TDO is first created.

### VII-3.1.1 The Type Manager Defines All Calls for a Type of Object

A type manager defines all basic calls for an object type. For example, the `Account_Mgt_Ex` type manager defines calls for *account* objects:

`Is_account`      Checks whether an AD references an account.  
`Create_account`      Creates an account with an initial balance.  
`Create_stored_account`      Creates and stores an account.  
`Get_balance`      Returns an account's balance.  
`Change_balance`      Changes an account's balance.  
`Transfer`      Moves an amount between accounts.  
`Destroy_account`      Destroys an account.

Callers must use the type manager `Account_Mgt_Ex` to do any of the above calls on an account. More complex calls must be composed from the type manager's basic calls. Again, it is important that the list of basic operations be complete, or else there is no way to do the operation on an account. For example, if you forgot the `Destroy_account` call, there would be no way to eliminate unneeded accounts.

### VII-3.1.2 Type Managers Hide Data Representation

Type managers provide *data abstraction*, concealing the representation of data from callers. For example, `Account_Mgt_Ex` provides the calls `Create_account` and `Change_balance` that affect the data in an account. To other services, an account is an *abstract data type*; the caller doesn't need to know or care how data in the account is represented.

Data abstraction makes software more:

- |                     |   |
|---------------------|---|
| <i>reliable</i>     | Only the type manager accesses the representation of a particular type of data. If the type manager is correct, then no outside program error can corrupt data of the type.   |
| <i>maintainable</i> | Data representation can be changed as long as the correctness of the basic calls is preserved.  |
| <i>extensible</i>   | Changes in functionality can easily be implemented as long as they are compatible with the existing interface. In our example, operations on accounts could be realized using transactions without any other program but the type manager having to be changed. |

### VII-3.1.3 Only the Type Manager Has the Key to Access the Type's Objects

The type of an object is uniquely defined by the object's TDO. A TDO for a new type of object can be created with `Object_Mgt.Create_TDO`. `Object_Mgt.Create_TDO` returns an AD to the new TDO. This AD has *create* and *amplify* rights. Those are necessary to create new instances of the managed object, and to add access rights to ADs of managed objects. Any module that has a TDO with *create* rights and *amplify* rights is by definition a type manager for that type.

In order to protect a newly created type, the AD to the TDO that has *create* and *amplify* rights should be confined to your type manager.

### VII-3.1.4 One Module Can Manage Multiple Types

The type manager model provides a flexible way of protecting objects. In particular, one module can manage as many types as you choose. However, it is obvious that the number of types that a type manager manages should be strongly limited. Otherwise the concept defeats itself. For example, it is common that one type manager manages closely related objects such as files and opened files.

## VII-3.2 Techniques

This section shows you each step in building a type manager. After reading this section, you will be able to:

- Define the Public Type
- Define Type Rights
- Define Exceptions
- Define the Type's Calls
- Define the Private Types

- Define Needed Type Overlays
- Create the TDO
- Bind to a Stored TDO
- Implement the Is Call
- Implement the Create Call
- Implement Calls that Require Type Rights
- Implement Calls that Don't Require Type Rights
- Implement the Destroy Call
- Make Operations Atomic
- Initialize the Type Manager
- Protect the Type Manager from Other Services.

The first four techniques describe the type manager's package specification, the public interface used by outside callers.

The next eleven techniques describe the type manager's package body, the package implementation, which is hidden from outside callers.

The last technique describes how to use BiiN™ Ada pragmas and the BiiN™ Systems Linker to completely protect your type manager from other services.

The `Account_Mgt_Ex` example is a type manager for *accounts*, each containing a long integer balance. It is a general-purpose type manager and could be used for inventory accounts, bank accounts, or other accounting applications. Appendix X-A contains complete listings for the `Account_Mgt_Ex` package. Various implementations of this type manager are described in this chapter and in Chapters VII-6 and VIII-2. The implementation described in this chapter is the simplest and supports active-only accounts.

### VII-3.2.1 Defining the Public Type

The type manager's package specification defines the *public type*, the type used by outside callers to reference an account. The `account_AD` access type is the public type for accounts. It references a private type `account_object` that is defined as a null record.

The package specification for `Account_Mgt_Ex` defines the public type:

```

114   type account_object is limited private;
115
116   type account_AD is access account_object;
117       pragma access_kind(account_AD, AD);
118       -- User view of an account.
```

The null record is defined in the private part of the specification:

```

295 private
296
297     type account_object is
298         -- Empty dummy record. The real object
299         -- format is defined in the package body.
300         record
301             null;
302         end record;
303
304 end Account_Mgt_Ex;

```

A dummy record format is defined because the BiiN™ Ada compiler requires a record layout in the package specification, but it is still desirable to conceal the actual object representation in the package body. The `account_object` type is never actually used, because account ADs lack rep rights and cannot be used to read or write account objects. Actual reading and writing is done within the package body with types defined there.

### VII-3.2.2 Defining Type Rights

Type rights allow a type manager to differentiate between users. The implementer of the type manager can require certain type rights for certain calls. It may also permit certain calls without any type rights. In the example presented here, the `Is_account` call is an example of a call that requires no type rights. (For more details, see Section VII-3.2.9.)

#### Declarations Used:

```

Object_Mgt.rights_mask
    Access rights type.

Object_Mgt.modify_rights
    Modify type right.

Object_Mgt.control_rights
    Control type right.

```

The type manager's package specification typically gives type-specific names to the type rights that it uses. The type manager's calls can check for needed rights before performing the call. A type manager does not always have to define all three rights. By convention, unused type rights should always be left turned on; otherwise a higher level routine will not be able to use them.

`Account_Mgt_Ex` defines two type rights:

```

121 change_rights: constant
122     Object_Mgt.rights_mask :=
123     Object_Mgt.modify_rights;
124     -- Required to change an account's balance.
125
126 destroy_rights: constant
127     Object_Mgt.rights_mask :=
128     Object_Mgt.control_rights;
129     -- Required to destroy an account.

```

If an account call is made without needed rights, then `System_Exceptions.insufficient_type_rights` is raised.

### VII-3.2.3 Defining Exceptions

The type manager's package specification defines any type-specific exceptions raised by its calls. `Account_Mgt_Ex` defines these exceptions:

```

94  insufficient_balance:  exception;
95      pragma exception_value(insufficient_balance,
96      insufficient_balance_code'address);
97      -- An operation failed because it would
98      -- cause a negative account balance.
99
100  balance_not_zero:      exception;
101      pragma exception_value(balance_not_zero,
102      balance_not_zero_code'address);
103      -- "Destroy_account" was called on an account
104      -- with a nonzero balance.

```

Text messages to be displayed by CLEX when an exception occurs can be bound to these exceptions at compile-time. These messages can be displayed on a terminal, for example.

```

71  insufficient_balance_code:
72      constant Incident_Defs.incident_code :=
73      (0, 1, Incident_Defs.error, System.null_word);
74
75  --*D* manage.messages
76  --*D* store :module=0 :number=1 \
77  --*D* :msg_name=insufficient_balance_code \
78  --*D* :short= \
79  --*D* "An account operation failed because it\
80  --*D* would create a negative balance."
81
82  balance_not_zero_code:
83      constant Incident_Defs.incident_code :=
84      (0, 2, Incident_Defs.error, System.null_word);
85
86  --*D* store :module=0 :number=2 \
87  --*D* :short= \
88  --*D* "An account cannot be destroyed because\
89  --*D* it has a non-zero balance."
90  --*D* exit

```

### VII-3.2.4 Defining the Type's Calls

The type manager's package specification defines all calls available to outside callers of the type.

Calls typically provided for a type *T* are:

<code>Is_T</code>	Checks whether an object is of type <i>T</i> . Only the type manager can reference <i>T</i> 's TDO and make this check.
<code>Create_T</code>	Creates a <i>T</i> object. Only the type manager can create and initialize <i>T</i> objects.
<code>xxx_T</code>	Any calls that need to read or write <i>T</i> objects. Only the type manager can read from or write to the object's representation.
<code>Destroy_T</code>	Destroys a <i>T</i> object. Only the type manager can explicitly deallocate <i>T</i> objects.

`Account_Mgt_Ex` defines all the typical calls:

```

Is_account
Create_account
Create_stored_account
Get_balance
Change_balance
Transfer
Destroy_account

```

It might appear at first glance that the `Transfer` call is not necessary since it can be composed of two calls to `Change_balance`. The problem with this solution is that it could happen that the calling program fails before it completes the transfer. Thus an amount may be deducted from the source account and not be deposited in the target account. The `Transfer` call is set up to be an atomic operation. It can only succeed as a unit and not partially. This concludes the type manager's package specification. The following techniques are done in the first body of `Account_Mgt_Ex`.

### VII-3.2.5 Defining the Private Types

The type manager's package body defines the *private types* used inside the type manager to reference the accounts. The `account_rep_object` type defines the object's representation. The `account_rep_AD` type is used for ADs with rep rights, allowing the type manager to read and write the representation:

```

38   type account_rep_object is
39     record
40       balance: Long_Integer_Defs.long_integer;
41       -- Current balance.
42     end record;
43
44   type account_rep_AD is access account_rep_object;
45     pragma access_kind(account_rep_AD, AD);
46     -- Private view of an account.

```

### VII-3.2.6 Defining Needed BiiN™ Ada Type Overlays

The `Account_Mgt_Ex` package body requires three different BiiN™ Ada types to represent the AD to one of its objects:

```

account_AD      Public AD without rep rights.
System.untyped_word
                Type required for Access_Mgt and Object_Mgt calls.
account_rep_AD  Private AD with rep rights.

```

Instead of instantiating `unchecked_conversions` type overlays are used here to the same goal. This is done using a BiiN™ Ada *address clause*. (Refer to the *BiiN™ Ada Language Reference Manual* for more details.)

```

180     account_rep: account_rep_AD;
181     FOR account_rep USE AT account'address;
182     account_untyped: System.untyped_word;
183     FOR account_untyped USE AT account'address;

```

Note that this technique has no runtime cost.

### VII-3.2.7 Creating the TDO

The package body described in this chapter is an *active-objects-only* package body, so every time the package initializes it creates a TDO. This poses no problems as long as objects of the type are not passivated or do not outlive their TDO or type manager. (This is explicitly enforced -- refer to Section VII-3.2.16 in this chapter for more details.)

```

48     account_TDO: constant Object_Mgt.TDO_AD :=
49                     Object_Mgt.Create_TDO;

```

A stored object should use a stored TDO as its type, as described in the next section.

### VII-3.2.8 Binding to a Stored TDO

If objects of the type can outlive a particular job, then the TDO should be a stored object, created once by the system administrator.

The type manager's package body then uses the BiiN™ Ada `bind` pragma to obtain the needed TDO AD with all type rights. The following example is excerpted from the second body of `Account_Mgt_Ex` package body in Appendix X-A. In this example, the `account_TDO` is first assigned a null value, then used in the pragma `bind`:

```

52     account_TDO: constant Object_Mgt.TDO_AD := null;
53     -- This is a constant AD but not really null; its
54     -- filled in with an AD retrieved by the linker.
55     pragma bind(account_TDO,
56                "account");
57     -- Bind to TDO for accounts.

```

This technique declares a BiiN™ Ada access type variable which is initialized with `null` at compile-time. The BiiN™ Ada pragma `bind` is an instruction to the BiiN™ Systems Linker to retrieve an AD from the directory entry that is named by the second argument of `pragma bind`. (For more details on BiiN™ Ada pragmas refer to the *BiiN™ Ada Language Reference Manual*.) The linker reinitializes the variable with the activated AD.

### VII-3.2.9 Implementing the `Is_account` Call

The `Is` call checks whether an object has the type managed by the type manager.

#### Calls Used:

```

Object_Mgt.Retrieve_TDO
    Retrieves object's TDO.

```

`Is_account` returns true if `obj`'s type equals `account_TDO`, false if `obj` is null or has another type:

```

70     begin
71         return obj /= System.null_word and then
72             Object_Mgt.Retrieve_TDO(obj) = account_TDO;
73     end Is_account;

```

### VII-3.2.10 Implementing the Create\_account Call

The `Create` call allocates an object of the right size and type, initializes the representation, and returns an AD with no rep rights.

#### Calls Used:

`Object_Mgt.Allocate`  
Allocates an object with specified size and type.

`Access_Mgt.Remove`  
Removes rights.

The `Create_account` call creates an account with a specified `starting_balance`:

```

94   begin
95     if starting_balance < Long_Integer_Defs.zero then
96       RAISE insufficient_balance;
97     else
98       account_untyped := Object_Mgt.Allocate(
99         size => Object_Mgt.object_size(
100           (account_rep_object'size + 31)/32),
101         -- Expression computes number of words
102         -- required to hold the number of bits
103         -- in an account.
104         tdo => account_TDO);
105
106     account_rep.all := account_rep_object'(
107       balance => starting_balance);
108
109     account_untyped := Access_Mgt.Remove(
110       AD      => account_untyped,
111       rights => Object_Mgt.read_write_rights);
112     RETURN account;
113
114
115   end if;
116 end Create_account;
```

The BiiN™ Ada `new` operator cannot be used here to allocate the object, because `new` by default allocates a generic object instead of an object with the desired type `account`. However, if we had made use of the Ada pragma `allocate_with` we could have specified a TDO to be used with the `new` operator. Thus we would obtain objects of the proper type when using `new`.

The size specified to `Allocate` is the number of 32-bit words. The BiiN™ Ada attribute `size` yields the number of bits required for the object's representation. The expression `(account_rep_object'size + 31)/32` yields the smallest number of 32-bit words with at least the required number of bits.

### VII-3.2.11 Implementing the Create\_stored\_account Call

Our particular example provides two `Create` calls, one that simply creates an object and returns an AD, and another that also stores the object with a pathname. The implementation discussed in this chapter does not support stored objects, however. For this reason the `Create_stored_account` function simply raises the `System_exception.operation_not_supported` exception as shown in the following excerpt from this implementation:

## PRELIMINARY

```
119 function Create_stored_account(  
120     starting_balance:  
121         Long_Integer_Defs.long_integer :=  
122         Long_Integer_Defs.zero;  
123     master: System_Defs.text;  
124     authority:  
125         Authority_List_Mgt.authority_list_AD := null)  
126     return account_AD  
127     --  
128     -- Logic:  
129     -- This call is not supported by this implementation.  
130     --  
131     is  
132     begin  
133         RAISE System_Exceptions.operation_not_supported;  
134         RETURN null;  
135     end  
136     end Create_stored_account;
```

### VII-3.2.12 Implementing Calls that Require Type Rights

For calls that require type rights, the type manager checks the rights on the caller's AD before performing the requested operation. The usual way to do this is with `Access_Mgt.Import`, which checks type rights before adding rep rights. `Import` raises `System_Exceptions.insufficient_type_rights` if needed rights are not present.

#### Calls Used:

`Access_Mgt.Import`  
Checks for rights and adds rep rights.

#### Declarations Used:

`System_Exceptions.insufficient_type_rights`  
Raised when the AD does not have the type rights needed for the call.

In `Account_Mgt_Ex`, the call `Change_balance` requires that the caller have *change rights* on the passed AD:

```

190     begin
191         account_untyped := Access_Mgt.Import(
192             AD      => account_untyped,
193             rights => change_rights,
194             tdo     => account_TDO);
195
196         new_balance := account_rep.balance + amount;
197
198         if new_balance < Long_Integer_Defs.zero then
199             RAISE insufficient_balance;
200
201         else
202             begin
203                 old_balance := account_rep.balance;
204                 account_rep.balance := new_balance;
205                 RETURN new_balance;
206             exception
207                 -- An exception in this inner block means
208                 -- that something has gone wrong with the
209                 -- update. The old balance is restored.
210                 when others =>
211                     account_rep.balance := old_balance;
212                 RAISE;
213             end;
214
215         end if;
216     end Change_balance;

```

The call `Access_Mgt.Import` checks the AD for *change rights* before adding rep rights.

### VII-3.2.13 Implementing Calls that Do not Require Type Rights

Calls that don't require type rights don't need to check the type rights before performing the call. As a result, the type manager can use `Access_Mgt.Amplify`, which adds rights without doing a check for type rights.

#### Calls Used:

`Access_Mgt.Amplify`  
 Adds rights without checking type rights.

An example of a call that doesn't require type rights is `Account_Mgt.Get_balance`. In this case, read rep rights are amplified:

```

151     begin
152         account_untyped := Access_Mgt.Amplify(
153             AD      => account_untyped,
154             rights => Object_Mgt.read_rights,
155             tdo     => account_TDO);
156         return account_rep.balance;
157     end Get_balance;

```

### VII-3.2.14 Implementing the Destroy Call

A type manager's `Destroy` call usually checks type rights for this destructive act, then deallocates the object's representation.

**Calls Used:**`Access_Mgt.Import`

Checks for rights and adds rep rights.

`Object_Mgt.Deallocate`

Deallocates the object's representation.

In the following example from `Account_Mgt_Ex`, the call `Object_Mgt.Import` checks for the appropriate type rights, then adds rep rights to the AD in order to be able to check the balance. If the balance in the account is zero, the account will be deallocated using

`Object_Mgt.Deallocate`:

```

326  begin
327      account_untyped := Access_Mgt.Import (
328          AD      => account_untyped,
329          rights => destroy_rights,
330          tdo     => account_TDO);
331
332      if account_rep.balance /= Long_Integer_Defs.zero then
333          RAISE balance_not_zero;
334
335      else
336          Object_Mgt.Deallocate (account_untyped);
337
338      end if;
339  end Destroy_account;
```

**VII-3.2.15 Making Operations Atomic**

Although the `transfer` call can in principle be composed of two successive calls to `Change_balance` there is a considerable disadvantage to this method; the process that performs the two calls could encounter an exception after performing the first call and before the second. If that happened, one account would be charged (or credited) but not the other one.

**Calls Used:**`Access_Mgt.Import`

Checks for rights and adds rep rights.

## PRELIMINARY

```
265     begin
266         source_untyped := Access_Mgt.Import (
267             AD      => source_untyped,
268             rights => change_rights,
269             tdo     => account_TDO);
270         dest_untyped := Access_Mgt.Import (
271             AD      => dest_untyped,
272             rights => change_rights,
273             tdo     => account_TDO);
274
275         new_source_bal := source_rep.balance - amount;
276         new_dest_bal := dest_rep.balance + amount;
277
278         if new_source_bal < Long_Integer_Defs.zero
279             or else
280             new_dest_bal < Long_Integer_Defs.zero then
281             RAISE insufficient_balance;
282
283         else
284             old_source_bal := source_rep.balance;
285             old_dest_bal := dest_rep.balance;
286             -- Old balances are recorded here
287             -- in case the update will have to be
288             -- rolled back.
289             begin
290                 source_rep.balance := new_source_bal;
291                 dest_rep.balance := new_dest_bal;
292             exception
293                 -- An exception in this inner block means
294                 -- that something has gone wrong with
295                 -- the update. Restore the old balances to make
296                 -- this operation atomic, then
297                 -- reraise the exception.
298                 when others =>
299                     source_rep.balance := old_source_bal;
300                     dest_rep.balance := old_dest_bal;
301                     RAISE;
302
303             end;
304             RETURN;
305
306         end if;
307     end Transfer;
```

The new balances of both the source and the destination account are computed. If either one is less than zero, the `insufficient_balance` exception is raised. Before the balances in the accounts are physically changed, they are stored. Any exception that is raised while the new balances are assigned causes the update to be rolled back and the original balances to be restored.

### VII-3.2.16 Initializing the Type Manager

The example that we discuss in this chapter manages accounts that cannot be passivated. In order to make sure that accounts cannot be passivated, the account TDO must contain the passive store attribute, bound to an instance that refuses requests for passive store operations.

#### Calls Used:

`Passive_Store_Mgt.Set_refuse_filters`  
Sets a type manager's passive store attributes object to refuse all outside requests for passive store operations.

`Attribute_Mgt.Store_attribute_for_type`  
Stores an attribute entry in a TDO.

## PRELIMINARY

```
350  begin
351      Passive_Store_Mgt.Set_refuse_filters(
352          passive_store_impl);
353      Attribute_Mgt.Store_attribute_for_type(
354          tdo      => account_TDO,
355          attr_ID  => Passive_Store_Mgt.PSM_attributes_ID,
356          attr_impl => passive_store_impl_untyped);
357  end;
```

Note that this piece of code is executed every time this package is initialized. Also, a new TDO is created at that time. The TDO and all the objects of the type manager are deallocated when the job that uses this package finishes.

A more general package body would be able to handle objects that can be passivated. In this case the TDO should only be created once and stored. This can be done by the system administrator using the `create.TDO` command in the `configure` utility. (For more details see the *BiiN™ Systems Administrator's Guide*.) You could also write a program that will execute only once, create a TDO and store it. The `Stored_Account_TDO_Init_Ex` procedure in Appendix X-A is an example of such a program.

### VII-3.2.17 Protecting the Type Manager from Other Services

Finally, a type manager may want to protect its address space from other services so that it and its objects are safe from accidental destruction or modification. Protecting the type manager's address space involves:

1. Creating a distinct address space with the BiiN™ Systems Linker.
2. Protecting the type managers address space from calling services via `pragma protected_return`.

The idea is to link the type manager into its own separate domain. In addition it might be desirable to put the type manager into its own subsystem. That means that the type manager will not share stacks with other services.

Refer to the *BiiN™ Systems Linker Guide* for information on how to create the type manager's own address space at link time. You will need to create a distinct domain and a distinct subsystem ID.

The BiiN™ Ada `pragma protected_return` ensures that all global registers will be cleared before control is returned to the calling process. This is to protect ADs that may have been left in the global registers by the call. Refer to the *BiiN™ Systems Linker Guide* for more information on these topics. (`pragma protected_call` is similar to `protected_return`; however it protects the calling routine from the routines it calls. `Account_Mgt_Ex` only calls OS routines. Therefore `protected_call` could be used here but is not really necessary.)

There is a performance penalty involved when you create a protected address space for a type manager. You will use extra memory for the type manager's distinct stack. There is also a time penalty when performing calls to a distinct domain.

## VII-3.3 Summary

- A *type manager* defines an object type and all basic calls for the type.
- Only the type manager can read from or write to the type's objects.
- A type is represented by a TDO.
- Type managers provide *data abstraction*, enhancing software reliability and maintainability.



# USING ATTRIBUTES **4**

---

## Contents

Concepts .....	VII-4-3
Techniques .....	VII-4-5
Defining a New Attribute .....	VII-4-5
Defining an Attribute Instance .....	VII-4-6
Initializing the Type's TDO .....	VII-4-7
Initializing an Objects Attribute List .....	VII-4-8
Summary .....	VII-4-8

An *attribute* is a package or data structure that can be defined for multiple objects or object types. Such packages or structures can be used independent of an object's type and without calling its type manager.

An attribute usually defines a set of operations that is supported by multiple objects, or object types, such as an I/O access method.

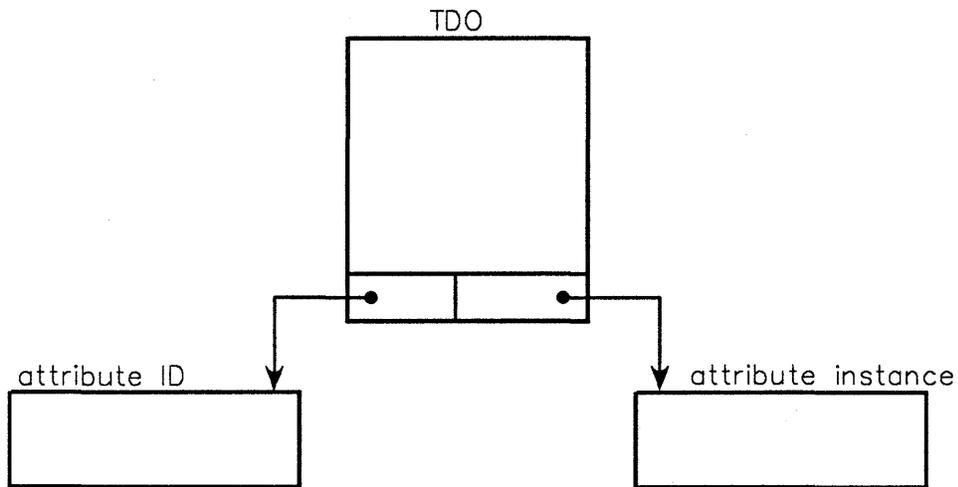
**Packages Used:**

`Attribute_Mgt` Manages attribute IDs and provides calls to store and retrieve attribute instances.

`Object_Mgt` Provides basic calls on objects.

An attribute can be defined either for an object or for an object type. In case of type attributes, an attribute list is contained in the Type Definition Object (TDO). In the case of object attributes, an attribute list is attached to the object proper. Whether in the TDO or attached to an individual object, an attribute list contains one or more *<attribute ID, attribute instance>* pairs. The attribute ID in the pair identifies the attribute (for example, the Byte Stream Access Method). The attribute instance in the pair references the object- or type-specific attribute value (for example, the type-specific implementation of the access method for the particular device type). An example of an object-specific attribute is `execute`. An executable object can be a CLEX script, a BiiN™/UX script, or an executable program. The attribute instances in this case specify how the object is to be executed.

Figure VII-4-1 shows the attribute data structure for a type-specific attribute.



**Figure VII-4-1. Attribute Structure**

In this chapter you will find an example of how to use type-specific attributes. Using object-specific attributes is very similar to what is shown in the example. In addition, in each section you will find information on how to achieve the particular step for an object-specific attribute.

In a later release we may have an example of an object-specific attribute.

## VII-4.1 Concepts

The attributes described in this chapter should not be confused with BiiN™ Ada attributes, used to indicate properties of declared entities in that language.

Even though *using* an attribute is independent of the object or its type, *defining* the attribute instances supported by an object or a type is specific to an object or a type. In the case of a type attribute, only the type manager can store attributes in the TDO, normally at system or program initialization when the TDO is created. In the case of an object attribute, anyone with control rights can store an attribute. But type-specific attributes cannot be overridden by object-specific attributes.

Though in most cases an attribute value is an AD to a package, an attribute value can be any `System.untyped_word`, either an AD to an object or a 32-bit data value. The attribute value can reference any object, not just a package. An example of an attribute value that does not reference a package is `Passive_Store_Mgt.PSM_attributes_object` where the attribute value is an AD to a record.

If an attribute is a package, invoking the attribute package's calls uses a fast *attribute call* mechanism supported by the OS and BiiN™ Ada. This mechanism uses the object type of the *first parameter* to a call to choose the appropriate type-specific instance of the package. This mechanism is used by many OS attributes, including all I/O access methods. If an attribute call is made on an object that does not support the attribute, then the `Standard.constraint_error` exception is raised. **The opinions vary on what exception will actually be raised. Also in the running are**  
`System_Exceptions.bad_parameter` and  
`System_Exceptions.operation_not_supported`.

Figure VII-4-2 shows an OS attribute, the Byte Stream Access Method, defined by the `Byte_Stream_AM` package, that is supported by different object types, such as opened files and opened pipes. Each object type has a type-specific implementation of the access method but applications need only call `Byte_Stream_AM` and their call is efficiently switched to the right implementation by the attribute call mechanism.

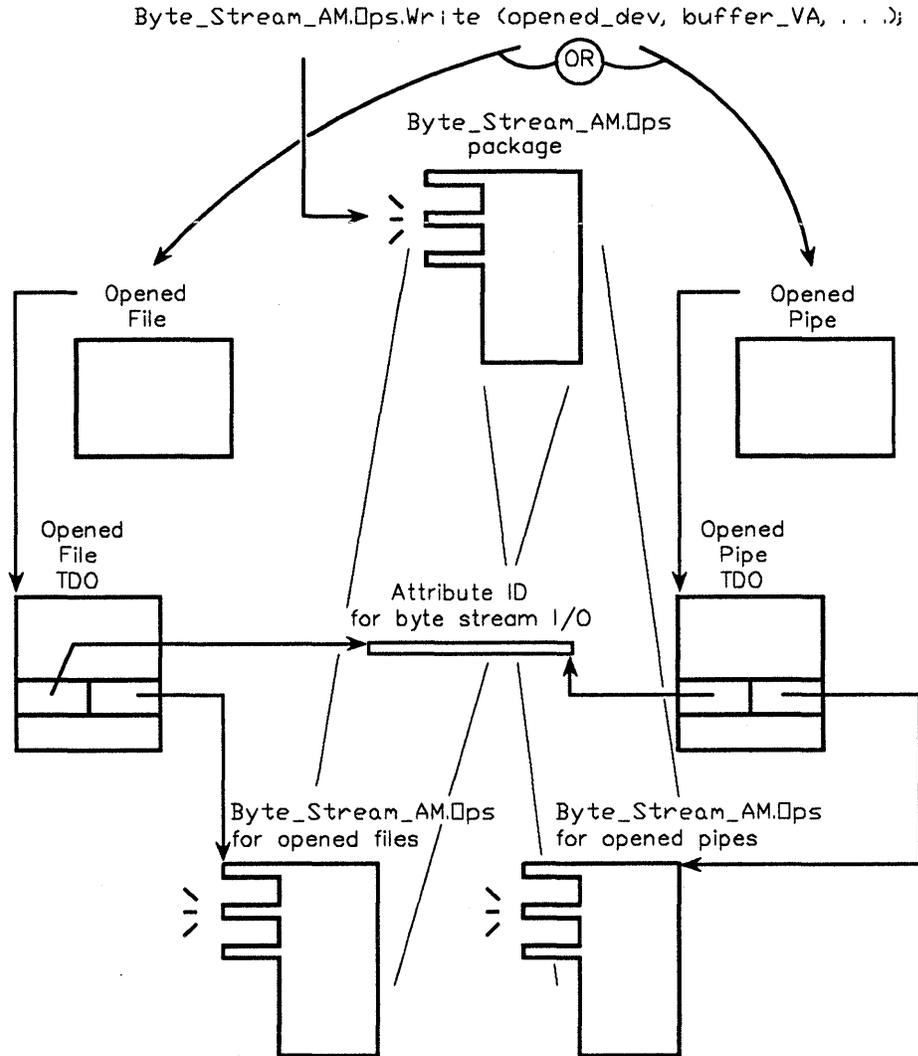


Figure VII-4-2. An OS Attribute

The OS defines many attributes used by type managers to customize System Services for their particular types. Every OS attribute appears to an application as another System Service. At the same time, implementers of new services can define type-specific instances of these OS attributes, without modifying, recompiling, or relinking the OS. *You can use attributes to extend and customize the OS -- without accessing its internals in any way.*

The "OS Attributes" appendix in the *BiiN™/OS Reference Manual* summarizes all OS attributes. Some commonly used OS attributes are:

- Byte stream I/O, specified by the `Byte_Stream_AM.Ops` package.
- Record I/O and record keyed I/O, specified by the `Record_AM.Ops` and `Record_AM.Keyed_Ops` packages.
- Character display I/O, specified by the `Character_Display_AM.Ops` package.

- Passive store, specified by the `Passive_Store_Mgt.PSM_attributes_object` record type.
- The execute attribute, specified by `Execution_Support.Ops`, an example of an attribute that can be object-specific.

## VII-4.2 Techniques

There are three techniques in using attributes:

- Defining a new attribute
- Defining a type-specific attribute instance for a type
- Initializing the type's TDO to refer to the attribute and instance.

Because attributes are most often packages, this section uses a simple package attribute for all three examples. This attribute contains a single call, which returns a type-specific type name. For example, for account objects, the type-specific instance will return the string "account". This example is not as useful as many attributes, such as I/O access methods, but its simplicity allows you to easily understand programming with attributes.

### VII-4.2.1 Defining a New Attribute

You will more often define attribute instances than define new attributes. We begin with defining an attribute because the example attribute is used by the subsequent techniques.

#### Calls Used:

`Attribute_Mgt.Create_attribute_ID`  
Creates a new attribute ID.

You create a new attribute by calling `Attribute_Mgt.Create_attribute_ID`. In this call you can specify whether the new attribute is type-specific or not. Type-specific attributes can only be stored in a TDO and not in an object's attribute list. The newly created attribute ID should be stored in the `aid` directory in the node's root directory.

The `Type_Name_Attribute_Ex` example package assumes that the attribute has already been created and stored. It binds the previously created ID to an attribute package using the BiiN™ Ada pragma `bind`.

```

7   type_name_attr_ID: constant
8       Attribute_Mgt.attribute_ID_AD := null;
9       pragma bind(type_name_attr_ID,
10                  "typnamattr");
11   -- Attribute ID is retrieved at link time using the
12   -- specified pathname. Should have store rights.
```

The attribute package `Type_Name_Attribute_Ex` defines two functions: one to get the attribute ID and one to return a type's name.

The `Get_type_name_attr_ID` function returns the new attribute's ID, required to store an instance of the type-name attribute:

## PRELIMINARY

```
14  function Get_type_name_attr_ID
15      return Attribute_Mgt.attribute_ID_AD;
16      -- Type name attribute ID, with type rights.
17      --
18      -- Function:
19      -- Returns the type name attribute's attribute ID.
```

The nested `Ops` package contains the calls to be defined by each type-specific instance. Only subprograms can be declared in such a package. The `package_type` pragma declares the nested `Ops` package to be a package type.

```
23  package Ops is
24      pragma package_type("typnamattr");
25      --
26      -- Function:
27      -- Provide "Type_name" attribute call.
28
29
30      function Type_name(
31          obj: System.untyped_word)
32          -- Any object that supports
33          -- the type name attribute.
34          return string; -- Name of the object's type.
35          pragma interface(value, Type_name);
36          --
37          -- Function:
38          -- Returns a printable name for an object's type.
39
40
41  end Ops;
```

Calls to any operations declared in the `Ops` package are switched to the proper instance, using the the *first* parameter to the call to select the instance.

The `Ops.Type_name` function body is empty. An empty subroutine body is allowed here due to the `package_type` pragma:

```
23  package body Ops is
24      --
25      -- Logic:
26      -- Attribute packages have null bodies.
27
28
29  end Ops;
```

Defining the attribute is done no differently for an object-specific attribute. In fact, an attribute that is not labeled as type-specific can be added to the attribute list of an object.

### VII-4.2.2 Defining an Attribute Instance

An attribute instance is simply a package that matches ("conforms to") the attribute's `Ops` package template and that is bound to that template using the `package_value` pragma:

## PRELIMINARY

```
1 with System,
2   Type_Name_Attribute_Ex;
3
4 package Account_Type_Name_Ex is
5   pragma package_value(Type_Name_Attribute_Ex.Ops);
6   --
7   -- Function:
8   --   Defines the type name attribute for accounts.
9   --
10  --   A type that supports this attribute has a
11  --   printable name. For example, a directory
12  --   listing utility could use this attribute to
13  --   print the types of the objects in a
14  --   directory.
15
16
17  function Type_name(
18    obj: System.untyped_word)
19    return string;
20    -- Name of the "account" object type.
21    --
22    -- Function:
23    --   Returns the type name for account objects.
24
25
26  pragma external;
27
28 end Account_Type_Name_Ex;
```

Note that the instance does not contain a nested Ops package. It corresponds to the attribute's nested Ops package and it will be called whenever one of the general Ops routines is called with a first parameter that is an object to which the attribute applies. Note that pragmas `package_value` and `package_type` occur paired. They can be compared to a *type definition* and a *variable declaration* in BiiN™ Ada.

The `Account_Type_Name` package body simply returns the name "account" when its `Type_name` function is called:

```
1 with System;
2
3 package body Account_Type_Name_Ex is
4
5
6   function Type_name(
7     obj: System.untyped_word)
8     return string
9   is
10  begin
11    return "account";
12  end Type_name;
13
14
15 end Account_Type_Name_Ex;
```

### VII-4.2.3 Initializing the Type's TDO

#### Calls Used:

`Attribute_Mgt.Store_attribute_for_type`  
Stores attribute ID and instance in TDO.

The implementation of the type-name attribute for accounts must be stored in the account TDO to be useful. The following excerpt is from the `Stored_Account_Init_Ex` example package body:

## PRELIMINARY

```
60     type_name_impl: System.untyped_word;
61     -- Implementation of type name attribute
62     -- for accounts.
...
107     type_name_impl := Account_Type_Name_Ex'package_value;
108
109     Attribute_Mgt.Store_attribute_for_type(
110         tdo      => account_TDO,
111         attr_ID   => Type_Name_Attribute_Ex.
112                 Get_type_name_attr_ID,
113         attr_impl => type_name_impl);
```

The 'package\_value BiiN™ Ada attribute (not to be confused with an OS attribute) is used to obtain an AD for the type-specific Account\_Type\_Name\_Ex package, an AD which is then stored in the TDO.

Handling TDOs and attributes that are stored objects is described in Chapter II-3.

### VII-4.2.4 Initializing an Objects Attribute List

#### Calls Used:

`Attribute_Mgt.Retrieve_attribute_list`  
Get's an object's attribute list. If none exists, creates one.

`Attribute_Mgt.Store_attribute_for_object`  
Stores attribute ID and instance in TDO.

Before you can use an object-specific attribute you have to store it in the object's attribute list. To do so, you have to retrieve the attribute list with `Attribute_Mgt.Retrieve_attribute_list`. This returns an AD to the object's attribute list. If none exists, a new attribute list is created. Finally, you can store the attribute using `Attribute_Mgt.Store_attribute_for_object`.

### VII-4.3 Summary

- An *attribute* is a package or data structure that can be defined for multiple objects or types.
- Explicitly type-specific attributes can only be associated with a type, not any object.
- An attribute *instance* is an attribute's value for a particular object or type.
- Attributes are identified by *attribute ID* objects.
- A type manager stores type-specific attribute instances of attributes that it supports in its TDO.
- Anyone with control rights to an object and store rights to an attribute can store that attribute in the object's attribute list.

# MANAGING ACTIVE MEMORY **5**

---

## Contents

A Brief Overview of How Memory Is Allocated .....	VII-5-2
Collecting Garbage Objects -- GCOL .....	VII-5-3
Local GCOL .....	VII-5-3
Global GCOL .....	VII-5-4
Techniques .....	VII-5-5
Trimming the Caller's Stack .....	VII-5-5
Starting Local Garbage Collection .....	VII-5-5
Setting/Changing Local GCOL Parameters .....	VII-5-5
Stopping Local Garbage Collection .....	VII-5-6
Getting Information About a Job's Local Memory .....	VII-5-6
Summary .....	VII-5-6

This chapter points out how you can use certain tools to manage active memory. This chapter does not explain underlying concepts and models of memory management in a BiiN™ system. Refer to Chapter VII-2 for a conceptual explanation of active memory.

For the most part, memory is managed automatically by the OS. You will want to read this chapter if you want to use optional calls to monitor and control your program's memory use.

#### Packages Used:

Object_Mgt	Provides basic calls on objects. Includes a call to shrink the calling process's stack.
SRO_Mgt	Provides calls to get memory information and control local garbage collection.

## VII-5.1 A Brief Overview of How Memory Is Allocated

Virtual address space in active memory is managed on a per-job and per-node basis. Each job has a special type of object associated with it that represents memory and objects local to the job and shared by all its processes. This object is known as a *local storage resource object* (SRO).

A local SRO provides a job with its own local address space, a subset of the node's virtual address space. Objects in the address space can be reclaimed by starting a local garbage collection daemon. The daemon is basically a memory optimization technique used for long-running jobs. It deallocates unreferenced objects (that is, objects with no ADs). See the `SRO_Mgt.Start_GCOL` call.

#### NOTE

Local garbage collection should be started in long-running jobs that need to respond quickly to events, terminal input, or other stimuli. If local garbage collection is not started by the job itself, then local garbage collection is done synchronously whenever the job reaches one of its memory limits. Synchronous local GCOL suspends all other processes in a job until it completes.

#### NOTE

Memory resources can be consumed by system calls other than those that explicitly allocate memory. For example, every time a transaction is started, the transaction counts against the job's "countable object" limit, even after the transaction is committed or aborted. Local GCOL will detect that the job is not using the transaction any longer and will decrement the job's "countable object count" accordingly.

Some more information about the local SRO:

- The local SRO is shared by all processes in the job, and only by the processes in the job.
- All processes in a job have implicit access to their job's local SRO.
- Most object allocation operations require an SRO as a parameter. This parameter defaults to the local SRO of the job to which the calling process belongs.

SROs have a number of properties that indicate how the objects allocated from an SRO are treated by various memory management functions. These properties are:

- relative lifetime* Determines when objects can be *deleted* (that is, deallocation of both the object's representation and its unique object descriptor) and constrains the storing of ADs in objects.
- memory type* Determines whether or not parts of an address space can be relocated.
- memory priority* Determines the frequency with which unused pages are swapped out of active memory; also determines when small segments are compacted onto a single page.
- allocation limits* Determines the amount of virtual storage allowed for all objects allocated.

Each one of these properties is discussed in more detail in Chapter VII-2.

## VII-5.2 Collecting Garbage Objects -- GCOL

Unreferenced objects in active memory (that is, objects with no active ADs) are periodically collected and deleted. This garbage collection (GCOL) is generally done automatically by the system, although it can be configured to clean up local objects for long-running jobs.

### VII-5.2.1 Local GCOL

Local garbage collection is executed by a special daemon process in a particular job. The daemon is *only* present if a process in the running job requests it and can be deleted at times when no garbage collection is needed.

It is useful to configure local GCOL for long-running jobs. When local garbage collection is configured for a job, it can be triggered in one of two ways:

- Automatically, whenever one of the remaining claim values becomes smaller than a percentage of the original claim set by the programmer.
- Manually, by calling `SRO_Mgt.Start_GCOL` with all parameters defaulted.

The effect of a `SRO_Mgt.Start_GCOL` depends on the values of the parameters. Table VII-5-1 summarizes the key parameters. Selected parameter combinations are used to start the daemon manually and then to stop GCOL by deleting the daemon. See "Techniques" in this chapter.

**Table VII-5-1. Key GCOL Parameters**

Parameter	Description
<code>storage_claim_percent</code>	Threshold value at which GCOL daemon wakes up. A percentage of the original number of words of virtual space that the specified SRO is allowed to allocate.
<code>OTP_claim_percent</code>	Threshold value at which GCOL daemon wakes up. A percentage of the original number of object table pages (OTP) assigned for the specified SRO.
<code>minimum_delay</code>	Minimum time between runs of the GCOL daemon.

This can have the effect of starting up the daemon. To prevent the daemon from running too often, a *minimum delay* can be specified as one of the trigger parameters. Garbage collection will not be triggered automatically if the elapsed time since it started its previous run is smaller than the minimum delay. Table VII-5-2 lists the special parameter values and their effect.

**Table VII-5-2. GCOL Parameters to Start and Stop Special GCOL**

Effect	Stop GCOL	Start GCOL
storage_claim_percent	0	100
OTP_claim_percent	0	100
minimum_delay	max_int	null_time

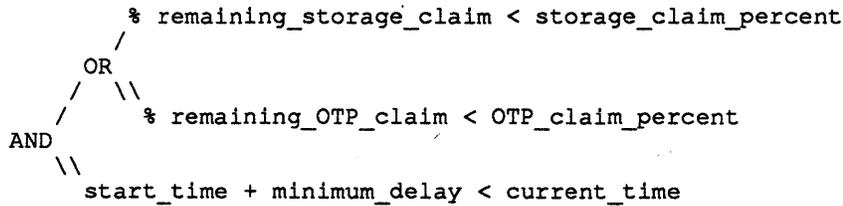
The `max_int` and `null_time` constants are defined in the `Long_Integer_Defs` and `System_Defs` packages under "Support Services."

The garbage collection algorithm has these properties:

- Only objects that are garbage at the time the algorithm starts will be collected.
- Garbage objects are deleted during the final phase of the algorithm.

`SRO_Mgt.Read_SRO_information` returns garbage collection related information.

Figure VII-5-1 shows the algorithm used by the system to determine when global garbage collection is performed:



**Figure VII-5-1. Algorithm That Controls Garbage Collection**

---

`SRO_Mgt.Start_GCOL` parameters specify when the GCOL daemon should begin running. When either of the claims granted to the job's local SRO drops below the trigger values and the minimum delay condition is met, the daemon starts running.

### VII-5.2.2 Global GCOL

Global garbage collection runs periodically and collects garbage objects allocated from both global SROs. Since global ADs may be stored in any object, all objects (local *and* global) on the node are checked. As with local garbage collection, objects and their associated space are only deleted during the final phase of the algorithm. Internally, the system minimizes the need for global garbage collection by minimizing the generation of global garbage.

## VII-5.3 Techniques

After reading this section, you will be able to:

- Trim the caller's stack
- Start local garbage collection
- Stop local garbage collection
- Get information about a job's local memory.

All techniques are taken from the `Memory_ex` example in Appendix X-A.

### VII-5.3.1 Trimming the Caller's Stack

A process can use an event handler to trim its stack in response to the `Event_Mgt.gcol` local event which is signalled to each process in a job whenever a local GCOL daemon is triggered.

**Calls Used:** `Object_Mgt.Trim_stack`  
Shrinks the calling process's stack.

Basically, `Trim_stack` looks at the process's current call stack pointer and then resizes the stack.

```
29 Object_Mgt.Trim_stack;
```

Trimming the stack frees memory and reduces the number of ADs that the local GCOL daemon must scan, thus speeding up garbage collection.

### VII-5.3.2 Starting Local Garbage Collection

To trigger local GCOL to start immediately in the calling job, you can use default parameters.

**Calls Used:**  
`SRO_Mgt.Start_GCOL`  
Controls the local GCOL daemon.

For example:

```
35 SRO_Mgt.Start_GCOL;
```

This will trigger the GCOL daemon to begin reclaiming space allocated from the job's local SRO.

### VII-5.3.3 Setting/Changing Local GCOL Parameters

Local GCOL parameters can be configured to trigger the local GCOL daemon. The daemon is triggered only when the conditions specified in the configuration are met.

**Calls Used:**

`SRO_Mgt.Start_GCOL`  
 Controls the local GCOL daemon.

For example, you might want to configure a local garbage collection daemon to run in the calling job when it has used 50% of its storage claim *or* 50% of its object table page claim, *and* at least 5 minutes has elapsed since a previous local GCOL run in the job.

```

45     SRO_Mgt.Start_GCOL(
46         storage_claim_percent => 50,
47         OTP_claim_percent    => 50,
48         minimum_delay        =>
49             Long_Integer_Defs."*(
50             Long_Integer_Defs.long_integer'(0, 5),
51             System_Defs.stu_per_min));
    
```

**VII-5.3.4 Stopping Local Garbage Collection**

A local GCOL daemon, once started, can be stopped using a `Start_GCOL` call.

**Calls Used:**

`SRO_Mgt.Start_GCOL`  
 Controls local GCOL.

For example:

```

58     SRO_Mgt.Start_GCOL(0, 0, Long_Integer_Defs.max_int);
    
```

This will kill any local garbage collection daemon in the calling job. It does nothing if there is no daemon.

**VII-5.3.5 Getting Information About a Job's Local Memory**

To obtain information about the current status of a job's local memory, call `SRO_Mgt.Read_SRO_information`.

**VII-5.4 Summary**

- Active memory consists of primary memory and swap space.
- A node's active memory contains objects used by executing programs.
- A one-to-one mapping exists between local SROs and jobs.
- Most active objects are allocated from local SROs.
- Global memory is allocated from global SROs.
- There are two types of global SROs: frozen global SROs and normal global SROs that indicate whether reclamation and compaction is allowed in global memory.
- Garbage collection can be configured for objects allocated from local SROs; it has certain trigger values that initiate a daemon process used to reclaim space.

# BUILDING TYPE MANAGERS FOR STORED OBJECTS

# 6

## Contents

Concepts .....	VII-6-2
Storing and Retrieving Objects in Passive Store .....	VII-6-2
Lifetime Requirements .....	VII-6-3
Storing Objects Requires Three Steps .....	VII-6-3
Object Trees in Passive Store .....	VII-6-3
The Type Manager Can Customize Passive Store Operations .....	VII-6-3
Synchronizing Access to Objects -- Transactions and Semaphores .....	VII-6-4
Techniques .....	VII-6-4
Defining the Type's Calls .....	VII-6-5
Implementing the <code>Create_account</code> call .....	VII-6-6
Implementing the <code>Create_stored_account</code> Call .....	VII-6-7
Starting, Committing, and Aborting a Transaction .....	VII-6-8
Storing the Master AD .....	VII-6-9
Updating the Object .....	VII-6-9
Implementing the <code>Change_balance</code> Call .....	VII-6-9
Implementing the <code>Transfer</code> Call .....	VII-6-11
Implementing the <code>Destroy_account</code> Call .....	VII-6-12
Initializing the Type Manager .....	VII-6-13
Protecting the Type Manager .....	VII-6-16
Summary .....	VII-6-18

This chapter describes how to build a type manager for stored objects. The type manager has the following characteristics:

- Objects can be passivated.
- Transactions ensure the consistency of passive versions.
- The multiple activation model is used.
- Objects should not be used by concurrent processes in one job.

The techniques necessary are illustrated by way of an implementation of the `Account_Mgt_Ex` example introduced in Chapter VII-3. The example used in this chapter has an interface identical to the one previously discussed. This is reflected by the fact that the Ada specification is identical for both packages. In addition to the packages described here, there is another implementation of `Account_Mgt_Ex` provided in Appendix X-A. That implementation is slightly simpler and does not provide transaction-oriented calls. The transaction-oriented implementation for stored accounts will be referred to simply as the implementation of `Account_Mgt_Ex`. If any other implementation is referred to, that fact will be explicitly stated. (All example packages used in this chapter can be found in Appendix X-A.)

This chapter is self-contained. It explains all techniques necessary for building a type manager for stored objects. It does not, however, discuss the fundamentals of the type manager model. If you do not know or understand the type manager model of protection, please read Chapters VII-1 and VII-3 before reading this chapter.

## VII-6.1 Concepts

Active memory is the immediate working space of the processors in one node. Active memory is (relatively) small, volatile, and local to a node. Passive store is not limited in size, permanent, and global to a distributed system. Objects that should survive shutdowns or system crashes, or that should pass between node boundaries, have to be passivated. A type manager that stores its objects is distributed by virtue of the distributed nature of passive store.

### VII-6.1.1 Storing and Retrieving Objects in Passive Store

All objects are created as active objects. *Local* active objects disappear when the creating job finishes. *Global* active objects survive as long as the system is up. Objects have to be passivated explicitly. Objects that have been passivated pass transparently between passive store and active memory.

Objects can be labeled *active-only*. Active-only objects cannot be passivated.

A job retrieves a stored object either transparently by supplying an AD or explicitly through a directory pathname. A job can also explicitly request that its current active version be updated from the passive version.

To remove an object that has been passivated, both the active version and the passive version have to be removed. Passive versions have always to be removed explicitly. Deallocating an object's active version has no effect on any existing passive version.

### VII-6.1.1.1 Lifetime Requirements

Objects have a type defined by a *Type Definition Object* (TDO). The TDO acts as a label for the type and it holds information specific to the type. An object may also have an *attribute list*. The lifetimes of TDO and attribute list should be at least as long as the object's own lifetime. For this reason TDO and attribute list have to be passivated before any object is passivated.

An object that has not explicitly been assigned a TDO or whose TDO has been removed is assigned the *generic TDO* by default. This may have certain undesirable consequences. For more details refer to Section VII-6.1.2.

### VII-6.1.1.2 Storing Objects Requires Three Steps

Storing an object for the first time requires three steps:

- TDO and attribute list is stored. If the TDO already exists this step is omitted.
- An AD is stored on the volume set where the object is to be stored. This AD can be stored in a directory or in another object. It will become the stored object's master AD. Master ADs cannot reference across volume sets.
- The object's representation is stored.

Once an object has a passive version, only its representation has to be updated if changes to the active version have been made. Note, that changes to an active version do not become permanent until the passive version has been updated.

### VII-6.1.1.3 Object Trees in Passive Store

Master ADs can be stored inside other objects. Thus hierarchical trees of passive objects can be created where one object holds master ADs for objects one level below. Object trees can be copied, and updated as one unit. Activating the root object of an object tree does not activate all the objects in the tree. Only the root object will be activated and all its ADs converted from passive to active form.

### VII-6.1.2 The Type Manager Can Customize Passive Store Operations

A type manager can supply its own routines for certain passive store operations thus customizing passive store. The mechanism behind this feature is an *attribute call*. For more details on attribute calls, refer to Chapter VII-4.

Passive store provides pairs of calls, *operation* and *Request\_operation* calls. Direct calls, such as *Update*, require representation rights, while *Request\_operation* calls, such as *Request\_Update*, generally require only type rights. One exception are generic objects which require read representation rights for *Request\_operation* calls. (The BiiN™ Operating System acts as a type manager for these objects.)

If upon invoking any *Request\_operation* call you receive the *System\_Exceptions.insufficient\_rep\_rights* exception, this is an indication that something has gone wrong with your TDO. It probably means that either the TDO could not be retrieved because you had insufficient rights to it or that it has been deleted altogether. Remember though that the type manager has total control over what actually happens when *Request\_operation* is called. (The type manager could conceivably require *rep* rights for these operations.)

If a type manager does not explicitly provide an implementation for a `Request_operation` call, the call is mapped by passive store to the direct call. This makes the direct call accessible with only type rights. Therefore, if any particular passive store operation should be disabled, an implementation of the corresponding `Request_operation` operation that refuses the operation, by raising an exception, for example, has to be provided. Otherwise the operation will be available to anyone with type rights.

### VII-6.1.3 Synchronizing Access to Objects -- Transactions and Semaphores

The use of transactions in passive store operations ensures that the stored data is consistent even in the event of system failures. Transactions also coordinate between different jobs accessing an object in passive store. Passive store operations either participate in a caller's default transaction, or a transaction is started for the duration of the call to passive store. Transactions have a built-in blocking protocol that avoids circular blocking of transactions.

Semaphores coordinate access to active objects, typically between processes inside one job. If in the object layout a *locking area* has been provided, passive store transparently creates a semaphore upon activation. A process can also explicitly create a semaphore. This is necessary if the object has never been passivated or is active-only. Semaphore locking is not used in the example described in this chapter. For more details on semaphore locking refer to Chapters VI-1, VI-2, and VIII-1.

It is important to note the conceptual difference between transaction locking and semaphore locking. Transaction locking directly locks an object. While a transaction holds its lock it blocks all others that request access. Semaphore locking relies on voluntary compliance by all participating processes. Semaphore locking is therefore used primarily to coordinate between related processes, for example inside one job.

## VII-6.2 Techniques

**Packages Used:**

<code>Access_Mgt</code>	Interface for checking and changing rights in access descriptors.
<code>Attribute_Mgt</code>	Provides a way to define general-purpose operations supported by multiple object types or objects, with different type-specific or object-specific implementations.
<code>Authority_List_Mgt</code>	Provides Calls to manage authority lists and to evaluate a caller's access rights to objects protected by authority lists.
<code>Directory_Mgt</code>	Manages directories and directory entries.
<code>Identification_Mgt</code>	Provides operations to manage IDs and ID lists.
<code>Object_Mgt</code>	Provides basic calls for object allocation, typing, and storage management. Defines access rights in ADs.
<code>Passive_Store_Mgt</code>	Provides a distributed object filing system.
<code>Transaction_Mgt</code>	Provides <i>transactions</i> used to group a series of related changes to objects so that either all the changes succeed or all are rolled back.
<code>User_Mgt</code>	Provides calls to manage a user's protection set and user profile.

This section describes the techniques necessary for a complete implementation of a type manager. The example described in this chapter and the example described in Chapter VII-3 share the same specification. Therefore, please refer to Chapter VII-3 for the following techniques:

- Defining the public type
- Defining type rights
- Defining exceptions
- Defining the private types
- Binding to a stored TDO.

**VII-6.2.1 Defining the Type's Calls**

The implementation described in this chapter provides the same calls as the one discussed in Chapter VII-3. Some calls work a little differently, though:

<code>Is_account</code>	Checks whether an AD references an account.
<code>Create_account</code>	Creates an account. Caller is responsible for storing the account.
<code>Create_stored_account</code>	Creates and stores an account. Caller supplies a pathname that is not already in use.
<code>Get_balance</code>	Returns an account's current balance.
<code>Change_balance</code>	Adds or subtracts an amount from the account's current balance.

**Transfer** Transfers amounts between accounts. Transfer either completes or fails as a unit.

**Destroy\_account** Removes an account's active and passive versions. May leave a master AD behind.

The implementation of the `Is_type` call will not be discussed here as it is identical to the one discussed in Chapter VII-3. For details, refer to that chapter.

### VII-6.2.2 Implementing the `Create_account` call

The `Create_account` call allocates an object of the right size and type, initializes the representation and returns an AD with no rep rights.

#### Calls Used:

`Object_Mgt.Allocate`  
Allocates an object of specified size and type.

`Object_Mgt.Deallocate`  
Removes an objects active version.

`Access_Mgt.Remove`  
Removes rights on an AD.

The following excerpt from the implementation of `Account_Mgt_Ex` shows all the steps in the `Create_account` call:

```

107 begin
108   -- 1. Check the initial balance:
109   --
110   if starting_balance < Long_Integer_Defs.zero then
111     RAISE insufficient_balance;
112
113   else
114     -- 2. Allocate and initialize the account object:
115     --
116     account_rep_untyped := Object_Mgt.Allocate(
117       size => (account_rep_object'size + 31)/32,
118       tdo => account_TDO);
119     begin
120       -- Inside this block it is guaranteed
121       -- that the object has been allocated.
122       account_rep.all := account_rep_object'(
123         balance => starting_balance);
124
125       -- 3. Remove rep rights for the exported AD:
126       --
127       account_untyped := Access_Mgt.Remove(
128         AD      => account_rep_untyped,
129         rights => Object_Mgt.read_write_rights);
130
131     exception
132       -- 4. If any exception occurs, abort any local
133       -- transaction, deallocate the account,
134       -- and reraise the exception:
135       --
136       when others =>
137         Object_Mgt.Deallocate(account_untyped);
138         RAISE;
139
140     end;
141
142     RETURN account;
143
144   end if;
145 end Create_account;

```

`Object_Mgt.Allocate` is used to allocate an object of the right size and type. This call can be substituted by the Ada new function if the BiiN™ Ada `allocate_with` pragma is specified with the private object type.

As can be seen from the above example, the `Create_object` call does not passivate the new object. It is the caller's responsibility to store the object. Note also, that if an exception occurs during the call after the account has been allocated, it will be deallocated and the exception reraised.

### VII-6.2.3 Implementing the `Create_stored_account` Call

The `Create_stored_account` call allocates an object of the right size and type, stores a master AD under a pathname provided by the caller, updates the passive version, and returns an AD with all type rights and no rep rights. This call illustrates all steps necessary in storing an object. In addition, you will learn how to employ transactions to protect passive store operations.

**Calls Used:**

- Object\_Mgt.Allocate  
    Allocates an object of the right type and size.
- Access\_Mgt.Remove  
    Removes rights.
- Transaction\_Mgt.Get\_default\_transaction  
    Gets the caller's default transaction.
- Transaction\_Mgt.Start\_transaction  
    Starts a local transaction.
- Transaction\_Mgt.Abort\_transaction  
    Aborts a transaction. Rolls back any changes done by transaction oriented calls within the transaction.
- Transaction\_Mgt.Commit\_transaction  
    Commits a transaction. Finalizes changes made within the transaction.
- Directory\_Mgt.Store  
    Stores an AD with a pathname.
- Passive\_Store\_Mgt.Update  
    Updates a passive version.

The Create\_stored\_account call allocates an object and removes rights on the exported AD the same way the Create\_account call does.

**VII-6.2.3.1 Starting, Committing, and Aborting a Transaction**

All passive store operations in this call are enclosed in a transaction, either a caller's default transaction, or a local transaction. The following excerpt from the implementation of Account\_Mgt\_Ex illustrates the use of a local transaction.

```

219      -- 4. Start a local transaction if there is not
220      --      a transaction on the stack:
221      --
222      if Transaction_Mgt.Get_default_transaction =
223         null then
224         Transaction_Mgt.Start_transaction;
225         trans := true;
226      end if;
227      begin
228      .
229      .
241         if trans then
242             Transaction_Mgt.Commit_transaction;
243         end if;
244      exception
245         -- 8. If any exception occurs, abort any local
246         --      transaction, deallocate the account,
247         --      and reraise the exception:
248         --
249         when others =>
250             if trans then
251                 Transaction_Mgt.Abort_transaction;
252             end if;
253             Object_Mgt.Deallocate(account_untyped);
254             RAISE;
255         end;
256

```

This technique avoids starting a local transaction if the caller already supplied a default transaction. Subtransactions should be avoided, unless specifically needed.

The above example also indicates the use of a program block to control the scope of the exception handler. Within this block one can assume that, if `trans` is true, a local transaction has indeed been started.

### VII-6.2.3.2 Storing the Master AD

The next step in storing the object is to store the master AD. The following excerpt from the implementation illustrates the call to `Directory_Mgt`.

```

230         Directory_Mgt.Store (
231             name    => master,
232             object  => account_untyped,
233             aut     => authority);

```

`master` is a text record that contains the pathname to store the account. The pathname must reference an existing directory and not be in use. If the caller did not specify an authority list, `authority` is null, and the target directory's default authority list will be used, if one exists. Otherwise the caller's default authority list will be used. If no default authority list is found, the exception `Directory_Mgt.no_default_authority_list` is raised.

### VII-6.2.3.3 Updating the Object

In the last step the object's representation is stored by calling `Passive_Store_Mgt.Update`:

```

237         Passive_Store_Mgt.Update(account_rep_untyped);

```

Note, that storing the AD does not passivate the object's representation. If you omit this last step, a later attempt to retrieve the object will result in the `System_Exceptions.object_has_no_representation` exception being raised.

### VII-6.2.4 Implementing the `Change_balance` Call

This call is a typical example of a type-specific operation. It illustrates the use of transactions to coordinate access to the passive version of an object between different jobs.

#### Calls Used:

`Access_Mgt.Import`

Checks and amplifies rights on an AD in one step.

`Transaction_Mgt.Get_default_transaction`  
Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`  
Starts a local transaction.

`Transaction_Mgt.Abort_transaction`  
Aborts a transaction.

`Transaction_Mgt.Commit_transaction`  
Commits a transaction.

`Passive_Store_Mgt.Reserve`  
Reserves a passive version of an object on behalf of a transaction.

`Passive_Store_Mgt.Update`  
Updates an object's passive version.

Two steps are necessary before any operations can be performed on the object; the type rights have to be checked on the AD supplied by the caller, and representation rights have to be amplified. The following excerpt from the implementation illustrates the `Access_Mgt.Import` call that performs these two steps together:

```

400     account_untyped := Access_Mgt.Import (
401         AD      => account_untyped,
402         rights => change_rights,
403         tdo    => account_TDO);

```

If the AD's type rights are insufficient, this call will result in the `System_Exceptions.insufficient_type_rights` exception being raised.

Before checking for a sufficient balance in the account, the technique described in the previous section is used to ensure that there is a default transaction. Next, the call reserves the passive version on behalf of the transaction:

```

412     Passive_Store_Mgt.Reserve(account_untyped);

```

The `Passive_Store_Mgt.Reserve` call may have three different outcomes:

- The object is available. The call succeeds and locks the object on behalf of the default transaction.
- The object is locked by another transaction. The blocking protocol permits blocking. The call blocks until the object becomes available.
- The object is locked by another transaction. The blocking protocol does not allow blocking. The call returns with the `System_exceptions.transaction_timestamp_conflict` exception.

You have to be prepared to handle this exception. The technique used here is illustrated by the following excerpt from the implementation:

```

405     loop
406         if Transaction_Mgt.Get_default_transaction =
407             null then
408             Transaction_Mgt.Start_transaction;
409             trans := true;
410         end if;
411     . . .
426     exception
427         when System_Exceptions.
428             transaction_timestamp_conflict =>
429             if trans then
430                 Transaction_Mgt.Abort_transaction;
431             else
432                 RAISE;
433         . . .
440     end;
441     end loop;

```

The `Passive_Store_Mgt.Reserve` operation is enclosed in a program block that has an exception handler for the `transaction_timestamp_conflict` exception. The block in turn is enclosed in a loop that repeats the `Reserve` call until it succeed in either blocking or reserving the object.

You can avoid the `Reserve` call. In that case, if the object had been updated by another job while your call was holding it, passive store would raise the `Passive_Store_Mgt.outdated_object_version` exception. You would handle the exception, request a fresh active version, by calling `Passive_Store_Mgt.Reset_active_version`, redo the changes, and try another up-

date. This technique is not acceptable for our example, since it might result in the decision, whether the balance be changed, being based on an outdated balance.

## VII-6.2.5 Implementing the Transfer Call

The `Transfer` call is similar in nature to other type-specific calls. It is discussed in more detail here, since it gives another example of how transactions can be used to keep data in passive store consistent.

### Calls Used:

`Access_Mgt.Import`

Checks and amplifies rights on an AD in one step.

`Transaction_Mgt.Get_default_transaction`

Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`

Starts a local transaction.

`Transaction_Mgt.Abort_transaction`

Aborts a transaction.

`Transaction_Mgt.Commit_transaction`

Commits a transaction.

`Passive_Store_Mgt.Reserve`

Reserves a passive version of an object on behalf of a transaction.

`Passive_Store_Mgt.Update`

Updates an object's passive version.

You might think that the `Transfer` call is superfluous, since two successive calls to `Change_balance` would achieve the same outcome. This is only partly true, as the `Transfer` call, as described here, enforces atomicity of the transfer. This means, transactions ensure the call cannot charge one account and not credit the other.

First, both ADs, for the source and the destination account, are checked and amplified using the one-step `Access_Mgt.Import` call:

```

494
495     source_untyped := Access_Mgt.Import (
496         AD      => source_untyped,
497         rights => change_rights,
498         tdo     => account_TDO);
499     dest_untyped := Access_Mgt.Import (
500         AD      => dest_untyped,
501         rights => change_rights,
502         tdo     => account_TDO);

```

Next, the call makes sure that there is a default transaction. Note, that if the caller already started a transaction, no further transaction is needed.

The call reserves both objects. Time stamp conflicts are handled the same way as described in the previous section, with a program block with exception handler inside a loop. The following excerpt illustrates the two `Reserve` calls.

```

511         Passive_Store_Mgt.Reserve(source_untyped);
512         Passive_Store_Mgt.Reserve(dest_untyped);

```

Note that if the first Reserve succeeds but the second one fails, Reserve will be called again on both objects. At that point the Reserve call on the first object simply results in no operation.

After both objects have been reserved, the balances are checked. As the following excerpt shows, an insufficient balance in either account will cause the `insufficient_balance` exception to be raised.

```

513         if source_rep.balance - amount < zero
514             or else
515                 dest_rep.balance + amount < zero
516             then
517                 RAISE insufficient_balance;
518
519         else
520             source_rep.balance :=
521                 source_rep.balance - amount;
522             dest_rep.balance :=
523                 dest_rep.balance + amount;
524             Passive_Store_Mgt.Update(source_untyped);
525             Passive_Store_Mgt.Update(dest_untyped);
526             if trans then
527                 Transaction_Mgt.Commit_transaction;
528             end if;
529             RETURN;
530
531         end if;

```

The last step in a successful completion of the call, as shown in the example above, is to update both objects. The new balances do not become permanent until both objects have been successfully updated and the default transaction committed. Note, that even though the variables `source_rep_balance` and `dest_rep_balance` have been assigned the new balances, this has no effect on the passive versions of the objects unless they are updated from the active versions.

## VII-6.2.6 Implementing the Destroy\_account Call

The `Destroy_account` call destroys an account's passive version, and removes the master AD if it is stored with a pathname.

### Calls Used:

```

Access_Mgt.Import
    Checks type rights and amplifies rep rights in one step.

Transaction_Mgt.Get_default_transaction
    Returns the caller's default transaction.

Transaction_Mgt.Start_transaction
    Starts a local transaction.

Transaction_Mgt.Abort_transaction
    Aborts a transaction.

Transaction_Mgt.Commit_transaction
    Commit a transaction.

Directory_Mgt.Get_name
    Returns the pathname of an object's master AD.

Directory_Mgt.Delete
    Deletes a directory entry.

```

`Destroy_account` uses the same techniques described in the previous sections to amplify rights on ADs and keep data in passive store consistent. The following example illustrates that after reserving the object's passive version, then if the balance in the account is zero, it calls `Passive_Store_Mgt.Destroy` to remove the object's passive version. If the object has no passive version, then the `Passive_Store_Mgt.no_master_AD` exception is raised.

```

621      Passive_Store_Mgt.Reserve(account_untyped);
622      if account_rep.balance /=
623         Long_Integer_Defs.zero then
624         RAISE balance_not_zero;
625
626      end if;
627      Passive_Store_Mgt.Destroy(account_untyped);

```

Finally the call attempts to remove the object's master AD. The following excerpt illustrates how:

```

629      loop
630      declare
631         path_text: System_Defs.text(path_length);
632      begin
633         Directory_Mgt.Get_name(
634            obj => account_untyped,
635            name => path_text); -- out.
636         if path_text.length >
637            path_text.max_length then
638            -- Text was lost. Retry:
639            path_length := path_text.length;
640         else
641            Directory_Mgt.Delete(path_text);
642            EXIT;
643
644         end if;
645      exception
646         when Directory_Mgt.no_name =>
647            EXIT;
648
649      end;
650      end loop;

```

If the master AD is (1) not stored in a directory, or (2) is stored in a standalone directory that does not have an associated name mapper, or (3) is stored in a standalone directory whose associated name mapper does not support `Get_name`, the call to `Directory_Mgt.Get_name` may fail and return with the `Directory_Mgt.no_name` exception.

Note that `pathlength` has an initial value of 60. In the event that the pathname is longer than 60 characters, the loop body will be executed again, and this time around the `path_text` text record is declared with the actual length of the pathname.

In the last step the master AD will be deleted by calling `Directory_Mgt.Delete`. A master AD for the object may remain if other directory entries on the same volume set references the object. One of these alias AD will then become a new master AD.

## VII-6.2.7 Initializing the Type Manager

In Section VII-6.1.1.1 we have discussed the need of the TDO to outlive any of its objects. For this reason the TDO has to be created and stored before the first call to this implementation of `Account_Mgt_Ex`. The TDO can be created either by the system administrator using the `configure` utility at node initialization time or by a separate procedure. In this chapter we shall discuss the second alternative. For more details on the first alternative, refer to the *BiiN™ Systems Administrator's Guide*.

**Calls Used:**

`Object_Mgt.Create_TDO`  
Establishes a new type by creating a new *type definition object* (TDO).

`Attribute_Mgt.Store_attribute_for_type`  
Stores an attribute with a TDO.

`Transaction_Mgt.Get_default_transaction`  
Returns the caller's default transaction.

`Transaction_Mgt.Start_transaction`  
Starts a local transaction.

`Transaction_Mgt.Abort_transaction`  
Aborts a transaction.

`Transaction_Mgt.Commit_transaction`  
Commit a transaction.

`Directory_Mgt.Store`  
Stores an AD with a pathname.

`Passive_Store_Mgt.Request_update`  
Requests an update of a passive version. No rep rights required.

The example described in this section is the `Stored_Account_TDO_Init_Ex` procedure. (The complete code of this procedure can be found in Appendix X-A.) This procedure has to be executed before `Account_Mgt_Ex` can be linked. Note also, that a TDO uniquely identifies its type. Calling the initialization procedure creates a new TDO that defines a new distinct type. You have to make sure that at any time there is only one passive version of the TDO on the system and that all instances of `Account_Mgt_Ex` refer to the same TDO, otherwise these instances will not be compatible.

The following excerpt from the `Stored_Account_TDO_Init_Ex` procedure shows how to declare the TDO and an instance of the *passive store attribute*.

```

52  account_TDO: Object_Mgt.TDO_AD;
53  -- TDO for accounts.
54
55  passive_store_impl:
56  Passive_Store_Mgt.PSM_attributes_AD;
57  -- Implementation of passive store attribute
58  -- for accounts.
```

The next step is to create the TDO, to dynamically allocate an instance of the passive store attribute, to initialize the instance, and to store it with the type:

```

93  passive_store_impl := new
94      Passive_Store_Mgt.PSM_attributes_object;
95
96  passive_store_impl.reset :=
97      Refuse_reset_active_version_Ex.
98      Refuse_reset_active_version' subprogram_value;
99
100 passive_store_impl.copy_permitted := false;
101
102 Attribute_Mgt.Store_attribute_for_type(
103     tdo => account_TDO,
104     attr_ID => Passive_Store_Mgt.PSM_attributes_ID,
105     attr_impl => Untyped_from_PSM_attributes(
106         passive_store_impl));
107 type_name_impl := Account_Type_Name_Ex'package_value;
108
109 Attribute_Mgt.Store_attribute_for_type(
110     tdo => account_TDO,
111     attr_ID => Type_Name_Attribute_Ex.
112         Get_type_name_attr_ID,
113     attr_impl => type_name_impl);

```

Note that the `passive_store_impl.reset` variable is initialized with a pointer to a subprogram that executes when

`Passive_Store_Mgt.Request_reset_active_version` is called. The following excerpt from the `Refuse_reset_active_version_Ex` package in Appendix X-A shows this procedure:

```

11  procedure Refuse_reset_active_version(
12      obj: System.untyped_word)
13  is
14      --
15      -- Function:
16      --   Handles requests to reset an account's active
17      --   version by refusing such requests.
18      --
19
20  begin
21
22      RAISE System_Exceptions.operation_not_supported;
23
24  end Refuse_reset_active_version;

```

Note, that this procedure simply raises the `System_Exceptions.operation_not_supported` exception.

In addition, the `copy_permitted` boolean is set to false. This prevents a caller to duplicate accounts. The `Attribute_Mgt.Store_attribute_for_type` links the instance of the passive store attribute to the TDO. This operation does not, however, passivate the attribute instance. The next excerpt from the initialization procedure shows how the TDO and the attribute instance are explicitly stored:

```

122     if Transaction_Mgt.Get_default_transaction =
123         null then
124         Transaction_Mgt.Start_transaction;
125         trans := true;
126     end if;
127
128     begin
129         Directory_Mgt.Store(
130             name => account_text,
131             object => Untyped_from_TDO(account_TDO),
132             aut => authority);
133         Passive_Store_Mgt.Request_update(
134             Untyped_from_TDO(account_TDO));
135         Passive_Store_Mgt.Request_update(
136             Untyped_from_PSM_attributes(
137                 passive_store_impl));
138         Passive_Store_Mgt.Request_update(
139             type_name_impl);
140
141         if trans then
142             Transaction_Mgt.Commit_transaction;
143         end if;
144     exception
145         when Directory_Mgt.entry_exists =>
146             if trans then
147                 Transaction_Mgt.Abort_transaction;
148             end if;
149
150         when others =>
151             if trans then
152                 Transaction_Mgt.Abort_transaction;
153             end if;
154             RAISE;
155
156     end;

```

Note again the use of transactions to ensure consistency of passive store.

## VII-6.2.8 Protecting the Type Manager

Recall for a moment two premises of the type manager model:

- A type manager protects objects of its type.
- A type manager provides black box type functionality.

In order for your type manager to accomplish these requirements you have to properly protect it from other programs. There are two aspects to protecting the type manager, namely

- protecting the type manager inside a running program,
- protecting the type manager's private ADs,

**Calls Used:**

Authority\_List\_Mgt.Create\_authority  
Creates an authority list.

Identification\_Mgt.Get\_user\_ID  
Returns caller's user ID.

Protecting the type manager inside a running program is equivalent to protecting its address space. The BiIN™ Systems Linker provides special support for linking modules so that each

one executes in its own protected address space, called *domain*. Besides creating an executable program, you can also create an *image module* with the linker. Image modules are pre-linked pieces of software that are not linked to a user's program until runtime and that can be shared by several users. An image module always executes in its own domain. For more details on domains and image modules, in particular on how to build domains and image modules with the linker, refer to the *BiiN™ Systems Linker Guide*.

Depending on how your type manager is to be used, you can choose to either link it in the standard way to an interactive interface, or to link it into an image module, thus making it available to be called by user programs. If the type manager consists of small routines that are not going to be called very often, the savings of shared code will not outweigh the overhead of creating an image module. For large programs used frequently, however, using image modules could result in substantial savings.

The second aspect of protecting the type manager is to protect its private ADs. It is necessary for the protection mechanism here that the linking not be left to the user for the following reason: As mentioned above, you need to create and store the TDO before invoking the type manager for the first time. The TDO is created by an initialization routine that stores it with a pathname. This directory entry is protected by an authority list. The following excerpt from `Stored_Account_TDO_Init_Ex` is an example where the authority list includes only the caller.

```

64   owner_only: User_Mgt.protection_set(1);
65   -- Protection set that includes only one ID, namely
66   -- the type manager's owner.
67
68   authority: Authority_List_Mgt.authority_list_AD;
69   -- Authority list that contains only one ID, namely
70   -- the type manager's owner.
.
.
115  owner_only.length := 1;
116  owner_only.entries(1).rights := User_Mgt.access_rights'(
117    true, true, true);
118  owner_only.entries(1).id := Identification_Mgt.Get_user_id;
119
120  authority := Authority_List_Mgt.Create_authority(owner_only);
.
.
129  Directory_Mgt.Store(
130    name => account_text,
131    object => Untyped_from_TDO(account_TDO),
132    aut => authority);

```

The TDO is retrieved at link-time using the Ada pragma `bind`. At that time rights are evaluated against the ID list of the calling process. The following excerpt from the implementation shows this:

```

52  account_TDO: constant Object_Mgt.TDO_AD := null;
53  -- This is a constant AD but not really null; its
54  -- filled in with an AD retrieved by the linker.
55  pragma bind(account_TDO,
56    "account");
57  -- Bind to TDO for accounts.

```

With the TDO thus protected, only people who are included in the TDOs authority list can link the program since noone else has access to the TDO. In the above example this is only you. (You could also create a separate ID just to protect the type manager.)

After the program is linked, it can execute with any ID.

## VII-6.3 Summary

In this chapter you have learned the techniques necessary to build a type manager for stored objects. In particular, you have learned that

- before the first object can be stored, a TDO has to be created and stored together with a list of attributes.
- storing an object requires two steps, namely storing the AD and updating the object's representation.
- the use of transactions keeps passive store consistent even in the event of a system failure.
- transactions can be used to synchronize access to passive objects.
- removing an object that has been passivated requires three steps, namely, deallocating the active version, destroying the passive representation, and deleting the master AD.
- special features of the linker and pragma `bind` can be used to protect the type manager.

### NOTE

Please keep in mind that the example described in this chapter permits processes in different jobs to concurrently use the objects of one type. There is no provision in the example for processes within one job to concurrently access one object. For details on how to achieve that, see Chapter VIII-1.

# UNDERSTANDING SYSTEM CONFIGURATION 7

---

## Contents

Creating a Node's Configuration .....	VII-7-3
Defining a Node's Configuration .....	VII-7-4
Configuration Attribute Calls .....	VII-7-4
Creating Configurable Objects .....	VII-7-5
Attaching Objects to Configurable Objects .....	VII-7-6
Starting Configurable Objects .....	VII-7-7
System SCOs and User SCOs .....	VII-7-8
The configure Utility .....	VII-7-9
Configuring Software Services .....	VII-7-6
Summary .....	VII-7-9

A *configuration* is an arrangement of objects representing the hardware and software resources of a particular BiiN™ node. System administrators routinely manage node configuration using the *configure* utility as described in the *BiiN™ Systems Administrator's Guide*. Two classes of programmers also need to understand system configuration:

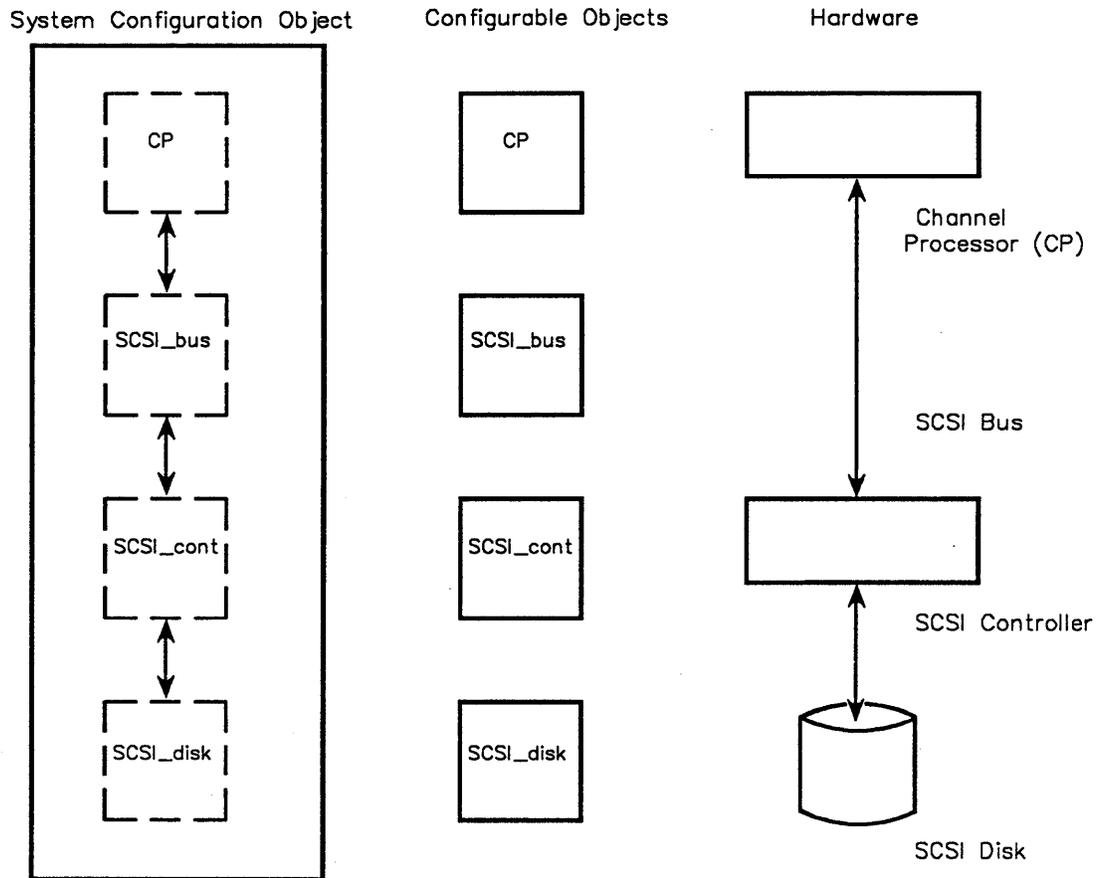
- Programmers adding hardware devices to BiiN™ systems
- Programmers adding software services with unique initialization requirements.

A BiiN™ system provides a variety of predefined system configurations describing systems covering the most common customer characteristics of hardware configuration: number of users, interactive or batch workload, or computational or I/O emphasis. Any of these predefined configurations may be used for generating a tested and balanced BiiN™ Operating System configuration, or may be modified to accommodate site-specific requirements.

**Packages Used:**

**Configuration** Provides operations for creating and modifying a system configuration.

Configuring a system includes creating configurable objects to represent hardware and software system components, then attaching and starting the objects to build a running system.



**Figure VII-7-1. System Configuration**

## VII-7.1 Creating a Node's Configuration

A node's configuration is created when the node is booted (see Figure VII-7-2). Booting a node begins with all hardware connections made, power on, and needed boot images but no software active in the system. Booting ends with a functioning, active system ready to respond to commands. The boot process must search for and initialize hardware and software modules and create the complex network of objects on which a running node depends.

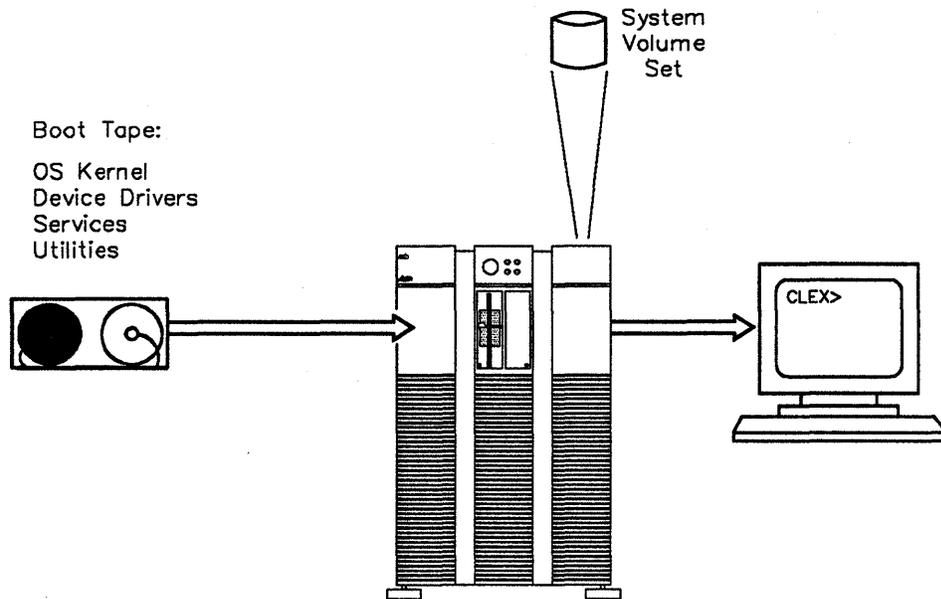


Figure VII-7-2. Booting a Node

Certain information must be available when a node is configured:

- What objects are part of the configuration. For example, there may be objects that represent physical I/O devices, device controllers, logical devices such as volume sets, and software units such as the OS kernel.
- One-time operations to be performed. For example, a hard disk may need to be formatted.
- The sequence in which operations should be performed. For example, a volume set cannot be created on a hard disk until after the disk controller is started and the disk is formatted.

## VII-7.2 Defining a Node's Configuration

A node's configuration is defined by a *System Configuration Object (SCO)*. An SCO provides information needed to create the configuration: the objects involved, the operations involved, and the required sequence of operations.

An SCO is a list of operations to perform, along with parameters for each operation. Only those operations defined by the `Configuration.Ops` attribute package are allowed in an SCO. If an object type needs to actively participate in the configuration process, that type must support the configuration attribute. Such objects are *configurable*.

## VII-7.3 Configuration Attribute Calls

The configuration attribute provides calls for:

- *Attaching* objects to configurable objects
- *Starting* configurable objects.

These calls are normally used within an SCO. Other configuration attribute calls, for *detaching* objects from configurable objects and *stopping* configurable objects, are normally not used within an SCO.

## VII-7.4 Creating Configurable Objects

System configuration is the specification of environmental hardware and software operating parameters of the components to be supported by a BiIN™ Operating System kernel image. *System components* include hardware modules (disk, controller, bus, etc.) and software modules (loadable, non-resident subsystems and optional support services).

A *configurable object (CO)* is a representation of a hardware or software module that must be configured at node initialization, or can be dynamically added to a running node. A *configuration attribute* supports the configuration of objects other than software services, particularly hardware components. A *service configuration attribute* supports the configuration of software services that have configuration and initialization dependencies in common. (An object is configurable only if its TDO contains the configuration or service configuration attribute.)

A configurable object must be created for each system component to be included in a system configuration. After it is created, it is not yet functional, but may be attached to other configurable objects. Attachment binds the configurable objects so they can be started and placed in a usable state.

When the configurable objects are no longer required to provide their function, they can be stopped. When they are no longer needed in the configuration, they are detached from other configurable objects to which they may have been attached.

Figure VII-7-3 illustrates the process of creating a configurable object.

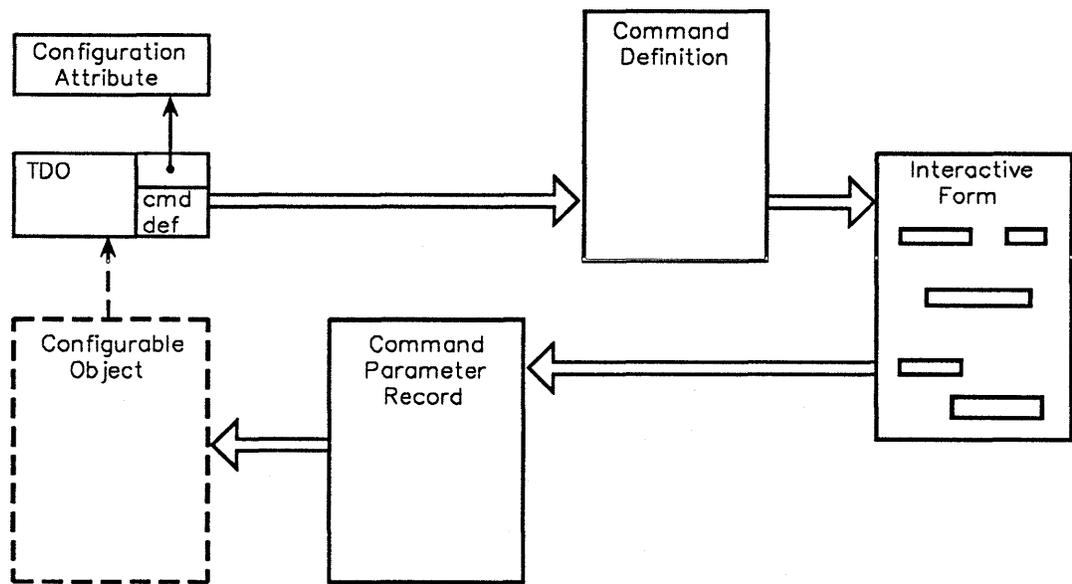


Figure VII-7-3. Creating Configurable Objects

An object to be made configurable must have a TDO which contains a configuration attribute. The TDO contains a command definition that defines the type of information required by a configurable object of the TDO's type. This command definition is displayed in an interactive form through which a user enters parameter data. The data collected by the interactive form is extracted from the command definition format and is used to create a configurable object.

## VII-7.5 Attaching Objects to Configurable Objects

`Attach` and `Detach` operations bind and unbind configurable objects. These configurable objects are considered head or tail objects depending on their relationship in the binding.

A *head object* is the initiating member of a pair of configurable objects associated with each other. A head object is characterized by its ability to function normally without being attached to another configurable object.

A *tail object* is the dependent member of a pair of objects associated with each other. A tail object is characterized by the requirement to be bound to a configurable object before it can become functional. Rights that may be needed on tail objects should be specified by the type manager supporting the `Attach` and `Detach` configuration calls on the tail objects. Tail objects don't have to be configurable when the attachment is unidirectional (tail object attached to head object but head object not attached to the tail object).

An attachment normally indicates that the tail object depends on the head object to function. For example, a volume set must be attached to a disk in order to function. A type manager's implementation of `Attach` normally checks the validity of the attachment by checking the type, rights, and state of the tail object and the rights and state of the head object.

An implementation of `Attach` can be bidirectional, making the attachment in the reverse direction as well. A bidirectional implementation is used when configurable objects are mutually dependent. For example, a CP (channel processor) and a SCSI (Small Computer System Interface) bus must communicate with each other in both directions and therefore require a bidirectional implementation of `Attach`.

## VII-7.6 Configuring Software Services

A configurable object is an object whose TDO contains an instance of a configuration attribute. Kernel, loadable, and application services require an attribute that can deal with the interdependencies inherent between them. For example, the object service uses the distribution service which in turn uses the clearinghouse service. An attribute is provided by configuration that, for example, enables the distribution service to ensure that the object service is started only after the Clearinghouse is started.

The mechanism used to support this binding of services is the *service configuration attribute*. This attribute allows a service to link itself with all the necessary and optional services that it uses. This attribute is extensible in that it allows a service to support the initialization of services that use it, and allows a service's initialization to itself depend on other services. This attribute registers a distribution service-dependent initialization procedure. These procedures are called by the BiiN™ Operating System after the system SCO has been processed when a node is present in a distributed system.

## VII-7.7 Starting Configurable Objects

All configurable objects provide *Start* and *Stop* implementations (which can be null). *Start* places a configurable object into a usable state by performing local initialization. *Start* is called by OS initialization as specified in a System Configuration Object (SCO). *Start* can also be called to start a component in a running system. Starting a configurable object should not start any attached tail objects. However, *Start* may require that tail objects be already started.

When the object to be started is a configurable object (CO) or a software service (SS) that neither is dependent on another software service nor is depended on by another software service, *Start* places it into a usable state by performing local initialization.

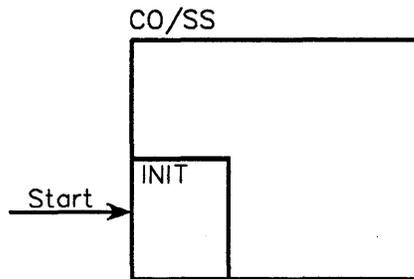


Figure VII-7-4. Simple Attach

When the object to be started is a software service that is dependent on another software service, *Start* performs local node initialization and attaches the first software service to the service on which it is dependent.

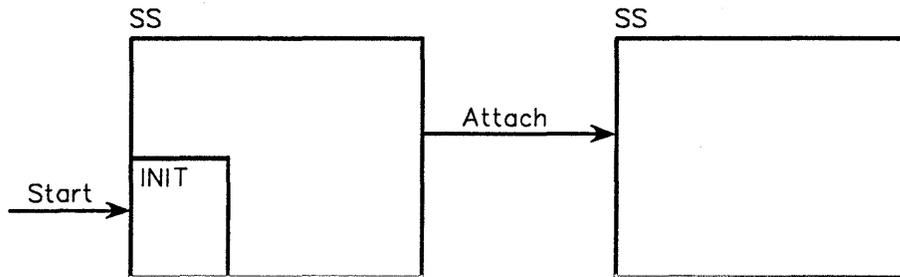


Figure VII-7-5. Attaching to a Dependent Software Service

When the object to be started is a software service that another service depends on, *Start* performs back attaches, that is, attaches the dependent service to the service that it depends on.

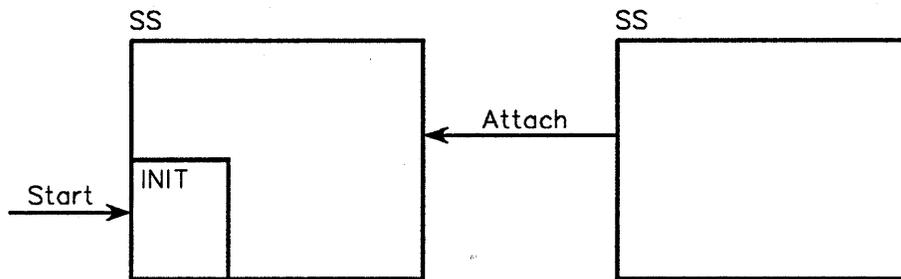


Figure VII-7-6. Back Attachment of a Dependent Software Service

When the object to be started is a software service (A) that is both dependent on another software service (B) and another service (C) depends on it, Start first attaches A to B on which it is dependent, and then performs back attaches from A to C.

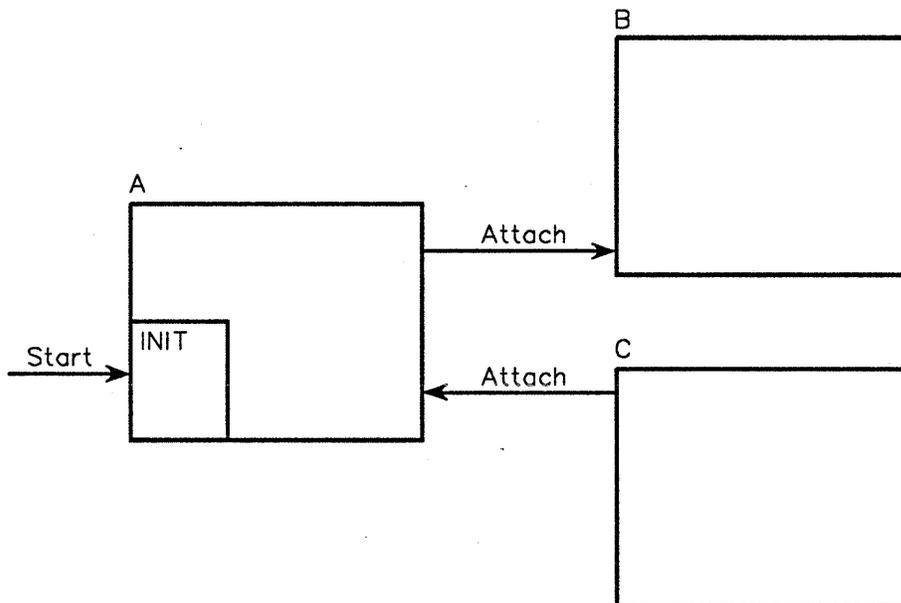


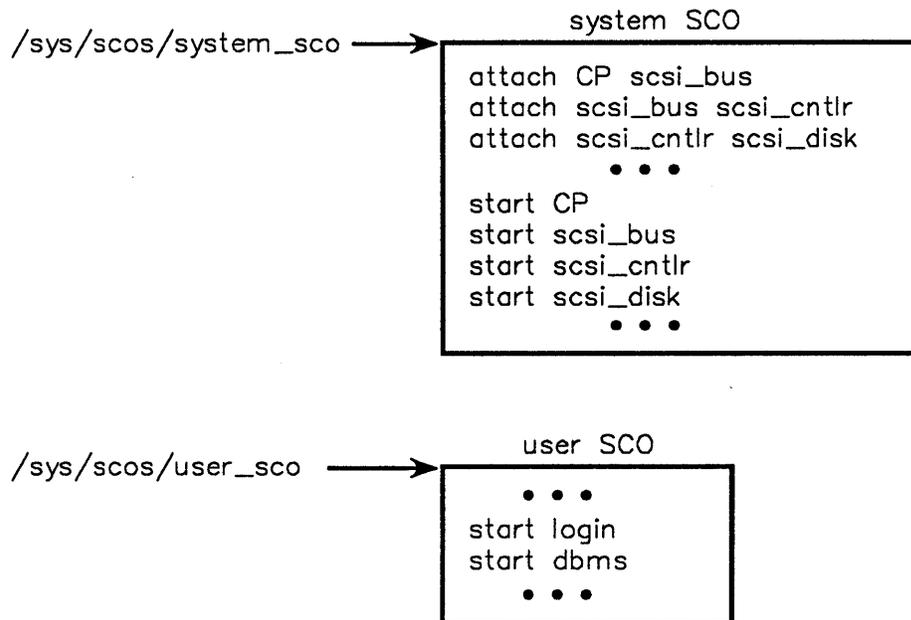
Figure VII-7-7. Compound Attachment

The order of attaches caused by starting a software service is implementation-dependent.

## VII-7.8 System SCOs and User SCOs

A *System Configuration Object* (SCO) is composed of a sequence of commands that attach COs together and start COs. The system administrator specifies a system SCO and a user SCO to use during OS initialization. A *system SCO* references hardware and software components of the configuration that are required to complete the node's initialization of the BiIN™ Operating System. A *user SCO* references components of the configuration that are not required to complete initialization of the OS, such as starting login services, database systems, specific application programs, and other activities that depend on disk write access or distributed system services.

Figure VII-7-8 illustrates system and user SCOs:



**Figure VII-7-8. System Configuration Objects**

---

The order of initialization of configurable objects is defined by the sequence of `Start` calls in the SCOs. The sequence for other configurable objects started after system initialization is determined by their type managers. For example, a set of configurable objects that is part of a CP (Channel Processor) subsystem can be started by starting the configurable object that represents the CP. Conversely, various network services require a separate start for each service specified in the configuration.

All system and user SCOs on a node are contained on the system volume set in the directory `/sys/scos`.

## VII-7.9 The `configure` Utility

Additional system configuration can be performed dynamically when the system is up and running, or at the next boot by updating or creating new SCOs.

The `configure` utility provides runtime commands to dynamically attach, detach, start and stop COs, and to create COs and SCOs for use at a future system initialization. See the *BiiN™ Systems Administrator's Guide* for information about the `configure` utility.

## VII-7.10 Summary

- Hardware components and system software modules are defined to represent a working system.
- A running system can be modified with the `configure` utility to build a site-specific system.

- System configuration is the specification of environmental hardware and software operating parameters of the components to be supported by a BiiN™ Operating System kernel image.
- System configuration is the process which brings a nonfunctional system to the point that it can execute a common application.
- *System components* include hardware modules (disk, controller, bus, etc.), and software modules (loadable, nonresident subsystems, and optional support services).
- A *configurable object (CO)* is a representation of a hardware or software module that must be configured at node initialization or can be dynamically added to a running node.
- A *System Configuration Object (SCO)* is composed of a sequence of commands that attach COs together and starts COs.
- When a system is up and running, additional system configuration can be performed dynamically, or at the next boot by using the `configure` utility.
- A *service configuration attribute* enables a service to link itself with all the necessary and optional services that it uses.

# Part VIII

## Distribution Services

---

This part of the *BiiN™/OS Guide* describes OS support for distributed services.

The chapters in this part are:

**Understanding Distribution**

Explains basic concepts of distribution and distributed services.

**Building a Distributed Type Manager**

Explains how to build a local single-activation distributed type manager, using remote procedure calls.

Distribution Services contains the following services and packages:

*clearinghouse service:*

CH\_Admin  
CH\_Client  
CH\_Support  
Node\_ID\_Mapping

*RPC service:*

RPC\_Admin  
RPC\_Call\_Support  
RPC\_Mgt

*transport service:*

Comm\_Defs  
Datagram\_AM  
DG\_Filter\_Mgt  
Distributed\_Service\_Admin  
Distributed\_Service\_Mgt  
ISO\_Adr\_Defs  
ISO\_Config\_Defs  
ISO\_TM\_Admin  
TM\_Comm\_Defs  
VC\_Filter\_Mgt  
Virtual\_Circuit\_AM



# UNDERSTANDING DISTRIBUTION

# 1

## Contents

Introduction .....	VIII-1-2
What a Distributed System Can Do .....	VIII-1-4
Naming .....	VIII-1-5
The Clearinghouse .....	VIII-1-6
Communications .....	VIII-1-8
Review of the Computational Model .....	VIII-1-10
Processes, Jobs and Sessions .....	VIII-1-10
Active and Passive ADs .....	VIII-1-10
Single and Multiple Activation Model .....	VIII-1-11
Single Activation Distributed Services .....	VIII-1-13
Protection in a Distributed System .....	VIII-1-13
Transparently Distributed Services .....	VIII-1-14
Passive Store .....	VIII-1-14
Directories .....	VIII-1-14
IDs .....	VIII-1-15
Files .....	VIII-1-16
Data Integrity, Synchronization, and Transactions .....	VIII-1-16
Summary .....	VIII-1-16

## VIII-1.1 Introduction

The BiiN™ Operating System supports distributed computing. A distributed system, capable of distributed computing, spans a number of BiiN™ nodes connected by a communication network. The network may contain several subnetworks. In this context a subnetwork is a homogeneous network such as ethernet or HDLC. It is important to note that the network connecting a distributed system need not be homogeneous. Two distributed system may also share a homogeneous subnetwork, such as a LAN (local area network), for example. Distribution is a high level concept independent of the communication media and associated communication protocols. Although distribution is independent of the communication media, it is optimized for high speed LAN applications.

A distributed system may appear as a "single machine" to the casual user. On the other hand a user can use his/her knowledge of the structure of the system, and work with individual or defined collection of components (nodes, I/O devices, and so on).

Figure VIII-1-1 shows an example of a network of BiiN™ nodes.

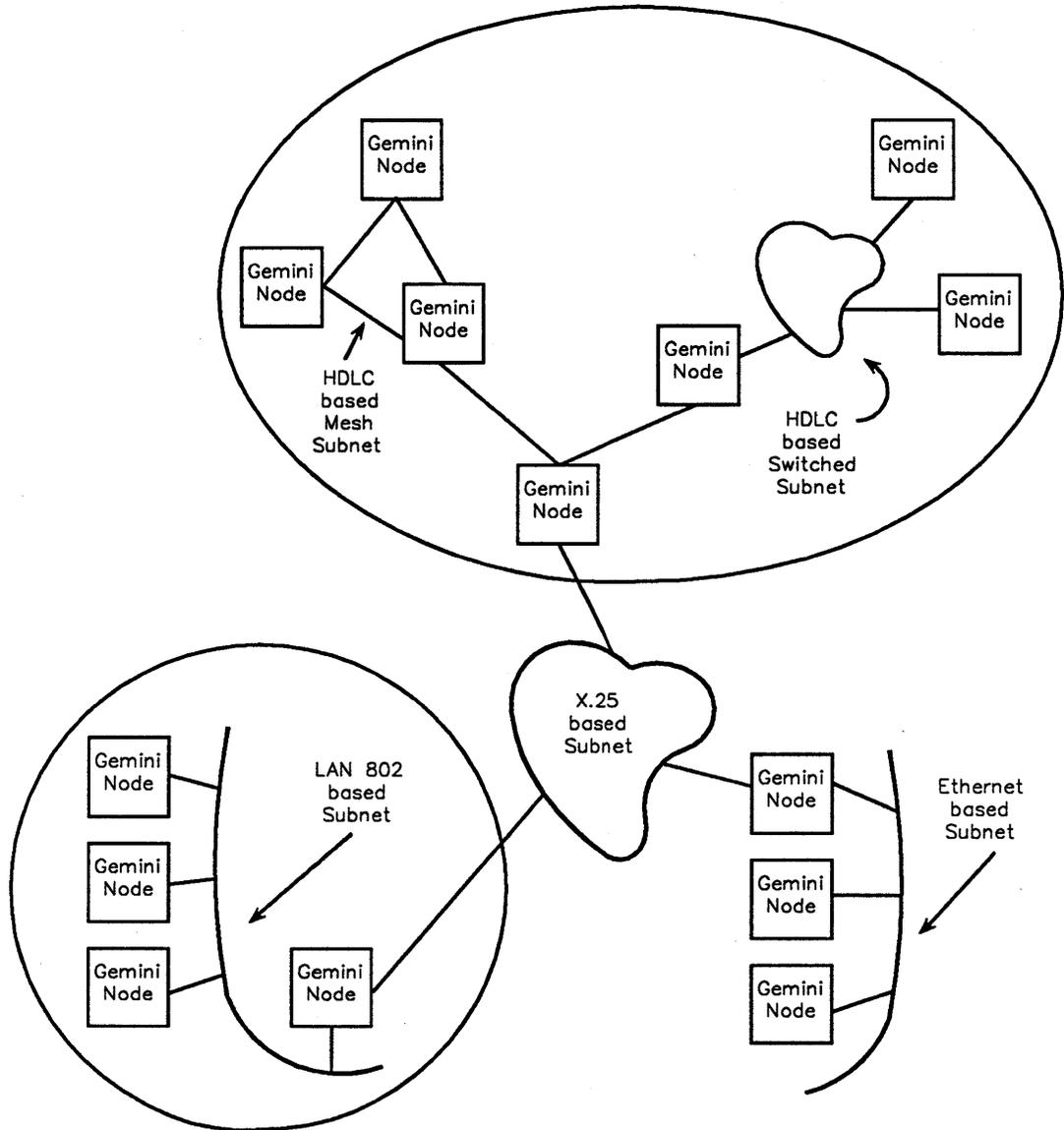


Figure VIII-1-1. A Network of BiiN™ Nodes

This particular network contains two bus-based LANs connected via a public packet switched network. Two additional subnetworks are shown, one based on a set of dedicated point to point communication lines and the second based on a circuit switched network. Circles indicate the boundaries of distributed systems.

Distributed computing lies in between *multiprocessing* and *networking*. Table VIII-1-1 lists important points in which the three concepts differ.

**Table VIII-1-1. Distribution vs. Multiprocessing vs. Networking**

<b>Multiprocessing</b>	<b>Distributed Computing</b>	<b>Networking</b>
Close Cooperation	Cooperation	Mutual Suspicion
Complete Trust	Tempered Trust Access/Resource Controls	No Trust
Single Administrator	Cooperating Administrators	Independent Administrators
Completely Shared Resources	Controlled Sharing of Resource	No Shared Resources
"Single Machine"	Homogeneous	Heterogeneous

On one hand distribution extends the concepts of multiprocessing beyond the limits of one shared memory, and on the other hand distribution takes the ideas of networking one step further.

This chapter explains the concepts of distribution. It does not explain specific techniques or point out the details of implementing a distributed service. This information is contained in chapter VIII-2.

The next section gives examples of what a distributed system can do and what it cannot do. The following sections discuss the most important aspects of distribution in more detail, in particular the following topics:

- Communications
- Naming
- Review of the computational model
- Single activation distributed services
- Protection in a distributed system
- Transparently distributed services.

Communications and naming are the two building blocks of the distributed architecture. For this reason special attention will be given to these two areas.

## VIII-1.2 What a Distributed System Can Do

Distributed computing makes it possible to build computer systems of any size from a single node up to a conglomerate of as many nodes as you choose. (There is no limit to the size of a distributed system.) Even though only a conglomerate of individual machines, the system acts in many ways as if it were one single machine, provided, of course, that the communication media is fast enough.

In most cases the user need not be aware of the physical organization of the distributed system; although nodes are individual machines that can operate by themselves, they appear to the casual user to be one unit. For instance, disks are mounted on individual nodes, but they appear to be mounted on all nodes at once. A user can also choose to run a job on a selected node or to store an object on a particular disk drive of his/her choice.

Jobs are the computational unit in a distributed system. Jobs run on single nodes but they communicate with other jobs, on the same node or on other nodes in the system. The interface for job communication on different nodes and the same node is identical, but there is an efficient implementation of intra-node communications.

By the means of interjob communication, independent jobs may exchange messages or related jobs may be coupled together. A *service*, such as the filing service, may contain jobs that run concurrently on all nodes of the system. The service is thus available on all nodes. All jobs belonging to the service communicate constantly and create a homogeneous environment of file access and usage across the entire system: Any file on the system is uniquely identified and stored in one place; this avoids a considerable amount of duplication. Files are available from any node: Requests to access a file are forwarded to the file's home node and executed there.

The filing service is a *universal service*. Universal services are decentralized; filing requests are serviced on the node where the requested file is stored. Since files can be stored at any node, filing services requests on all nodes of the system. (Diskless nodes are currently not supported.)

Services can also be *regional*. A regional service is centralized; requests can be issued on many nodes but only a few nodes (or even a single node) service requests. Universal services are "symmetric"; on all nodes there is an *agent* that accepts and distributes requests and a *server* that receives requests from an agent and executes them. A regional service is "asymmetric"; there are many agents and only a few servers.

Compare a universal service to the postal service: Every town has its own post office that receives mail from other towns, distributes it to the addressees, and collects and processes outgoing mail. A regional services resembles more an insurance company. Insurance agents sell policies for a company that underwrites the policies. The agent interacts with the clients on the one side and with the insurance company on the other. The insurance agent does not underwrite policies himself.

As an example of a regional service imagine an airline reservation system. All booking information is kept in a few locations. Agents in branch offices make reservations on their local nodes; the requests are transparently forwarded to one of the nodes where booking information is kept.

Distributed systems provide parallel processing. A *session* may span several nodes and contain jobs on all those nodes. If a task can be partitioned, processes in these jobs can work on parts of the task asynchronously.

Currently, load balancing is not implemented. The architecture does not discourage this functionality, however. An application implemented as a distributed service can decide based on the load in the system, how it routes requests to its servers. An example is a distributed batch utility that submits batch jobs to the node with the lowest load in the system.

The following two sections discuss the most important elements in a distributed system, namely how entities are named, and how nodes in the system communicate.

## VIII-1.3 Naming

One of the two building blocks of a distributed architecture is a location-independent naming mechanism. Here is an example of the merit of location-independent naming: A volume set is identified on the machine level by a unique *volume set ID*. The volume set ID reflects where the volume set is currently mounted in the system. The symbolic name of the volume set on the other hand has nothing to do with the location of the volume set. More importantly, the symbolic name does not change when the volume set is moved to another node. You can refer to the volume set without having to know where it is currently located.

Naming extends to stored objects, users, nodes, and volume sets. The map from machine level identifiers to symbolic names is maintained the *clearinghouse*.

The clearinghouse centralizes network information in a few locations. Thus network information can be updated quickly and easily. Volume sets can be moved from one node to another, a node may be added, or a node may be disconnected: Those changes have to be recorded in only a few places, namely where copies of the clearinghouse are kept.

### VIII-1.3.1 The Clearinghouse

The clearinghouse is decentralized and replicated. Instead of one global clearinghouse server there are many local servers each storing a copy of a portion of the global information. Some information in the clearinghouse is cached locally by other services. This allows to bypass the clearinghouse for efficiency and when access to a clearinghouse server is not possible due to a communication failure.

User ids, for example, are available at all nodes. This is necessary in order to allow users to log on to a local node even if that node is disconnected from the rest of the system. The same applies to locally mounted volume sets.

The organization of the clearinghouse is hierarchical. Names of clearinghouse entries consist of four parts representing the four level hierarchy. The names of the four parts are *organization*, *domain*, *environment*, and *local*. Clearinghouse names are specified with single, double and triple slashes between the level names. A full clearinghouse name is always of the following form:

```
///org/dom/env/local
```

Organization and domain together reference a naming domain. A large distributed system is typically split up into multiple naming domains. Thus name evaluation does not become hopelessly slow when the system becomes very large. Every node in the system belongs to exactly one naming domain. The clearinghouse is partitioned on the naming domain level. This means that one clearinghouse server stores all entries of the form

```
///organization/domain/anything/anything
```

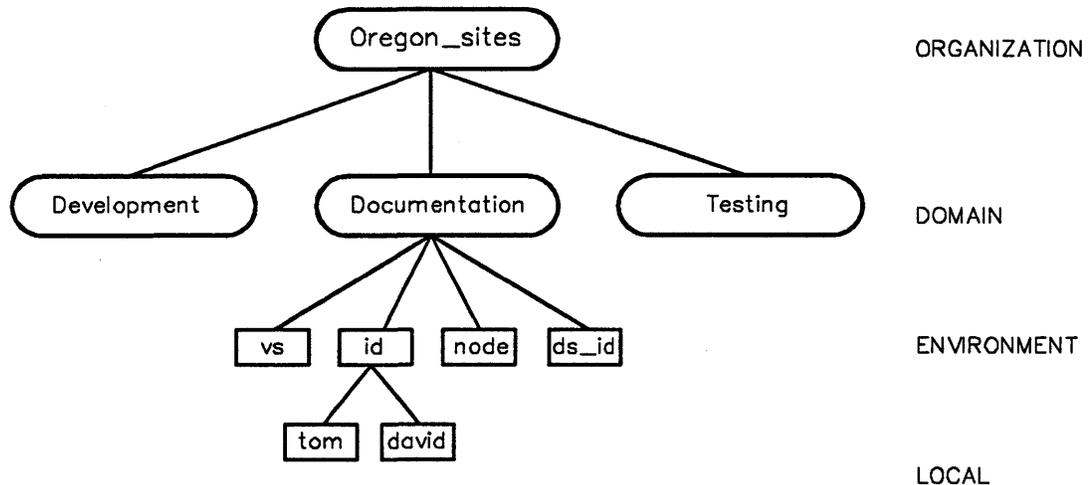
A name starting with two slashes reference an entry in the callers organization:

```
//dom/env/local
```

A clearinghouse name starting with one single slash refers to the local naming domain:

```
/env/local
```

Figure VIII-1-2 illustrates the hierarchical structure of the clearinghouse.



**Figure VIII-1-2. The Hierarchical Structure of the Clearinghouse**

The information in figure VIII-1-2 is shown together in one place. In a real system it is partitioned, replicated, and stored in different locations. The figure is very much simplified and shows entries for only one naming domain. This is done for convenience and ease of understanding.

There is one special naming domain per distributed system, called the *figurehead* naming domain. This domain covers the entire system. More specifically, it references all other entries in the clearing house. In fact, the figurehead naming domain defines the distributed system. It is used whenever the naming domain of an object is not known. This can happen when a passive object is activated: *Passive\_Store\_Mgt* has a *unique identifier* (UID) for the object which contains the ID of the volume set where the object is stored. With the help of the figurehead naming domain, *Passive\_Store\_Mgt* maps the volume set ID to the network address of the node where the volume set is mounted.

The clearinghouse is maintained by the clients, BiiN™ Operating System services or applications that use the clearinghouse. Clients maintain clearinghouse *environments*. In an environment the clients store names and properties associated with those names. The naming service, for example, maintains the *vs* environment. It uses this environment to map volume sets to node addresses, indicating where the volume set is mounted. Another example is the protection service. It maintains the *id* environment that maps user IDs to user profiles (and thus to symbolic user names). This information is used by the logon utility. The distributed OS services use a total of four environments in the clearinghouse, namely *vs*, *id*, *node*, and *ds\_id*. From the point of view of the clearinghouse there is no difference between those environments and other environments. The clearinghouse simply provides the mechanisms for binding symbolic names to properties in one networkwide location. It is entirely up to the client to attribute meaning to the clearinghouse entries.

Most applications will use the clearinghouse indirectly through the OS services. However, if the need arises, an application may use the clearinghouse directly, either through the above mentioned environments or even by setting up its own environment.

A request to the clearinghouse to bind a name to a set of properties may originate anywhere in a distributed system. The request will be directed to a clearinghouse *agent*. The agent knows

the address of at least one clearinghouse *server*. The server will either handle the request directly or, if it does not store the required information, forward the request further to a server that stores the information. This entire process happens invisibly to the client.

In summary the clearinghouse provides the basic tools needed for a high level naming mechanism. But the function of the clearinghouse goes beyond this task. Any type of information may be bound to a name; an internetwork address, in the case of a node, or a telephone number, in the case of a user. Services can use the clearinghouse to whatever purpose they require. The merit of the clearinghouse is that it centralizes all this information and makes it available to everyone. One of the most important uses of the clearinghouse is to provide location independent naming.

## VIII-1.4 Communications

If distribution is compared to a brick wall, then naming corresponds to the bricks and communications to the mortar; either one without the other would be useless. And just as mortar and bricks become invisible once plaster has been applied, so should the details of naming and communications be invisible in a distributed system. However, nobody can build a wall without mortar, and nobody can build a distributed system without communication between nodes. In order to understand distribution, we have to have some understanding of how nodes communicate.

One of the guiding principles in the BiiN™ architecture is that logical structures hide physical structures. This principle also pertains to communications: The system supports a variety of different communication protocols, such as Ethernet, IEEE 802.3, HDLC and X.25. *Transport services* hide the details of these various subnetworks. Through the interfaces provided by transport services a distributed service can use two different high level communication protocols, a connection oriented and a connectionless protocol. We refer to the connection-oriented protocol as a *virtual circuit* and to the connection-less protocol as a *datagram*.

Datagrams are short one-way messages sent from one job to another. They are similar to letters sent through the mail: There is no guarantee that a datagram sent will be received by the addressee or that a number of messages sent will be received in the order that they were sent. Transport services only guarantee that if a message is received, it will be intact. On the positive side datagrams are inexpensive (just as letters), fast, and require little overhead.

Virtual circuits provide a full duplex connection between the connected parties. A virtual circuit is a bidirectional ordered flow of bytes similar to a telephone connection. Receipt of a message is acknowledged and messages sent in a certain order arrive at the addressee in that same order. Setting up, maintaining, and tearing down a virtual circuit presents considerable overhead.

There is a third way for processes to communicate. This method is called a *remote procedure call*. Remote procedure calls are built on top of datagrams and share some of the advantages of datagrams. They provide the following additional services:

### A simple call interface

Making an RPC involves no more than making an ordinary procedure call.

### Authentication and security

Messages are authenticated to insure that they are intended for that server and that they have not been modified in transit.

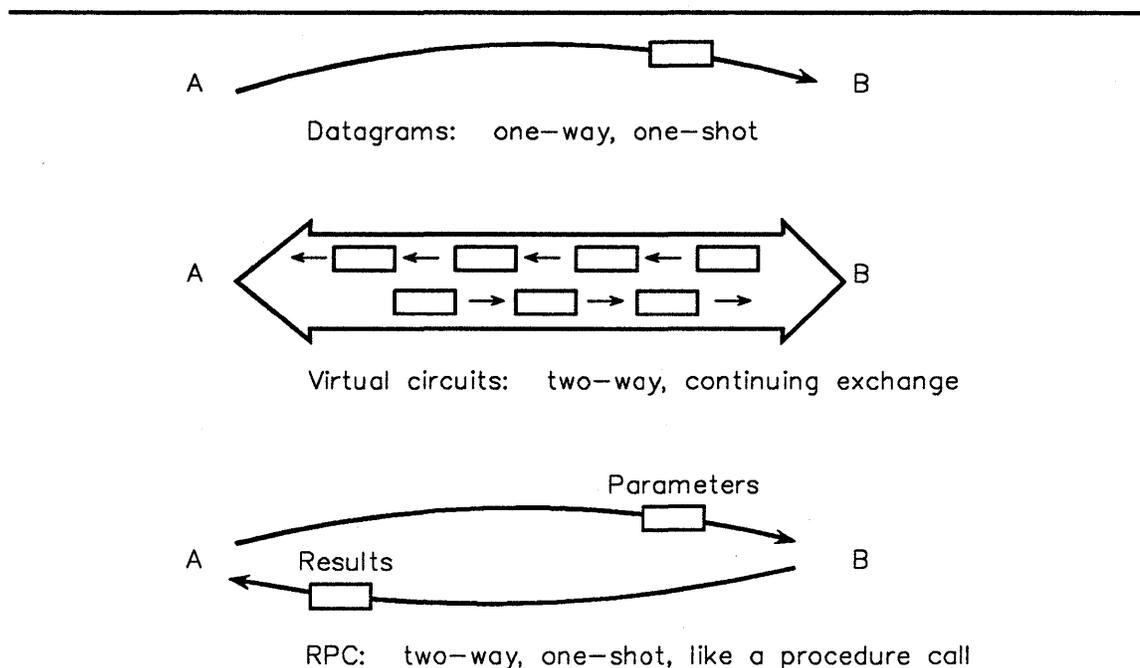
Converting ADs    ADs are converted to their passive form.  
 Locating         Given an AD to the server, RPC locates the server.

RPCs are message/reply pairs. They force the caller to wait until the call has completed. A series of RPCs made by one process is strictly ordered, since the calling process cannot make another RPC before the previous one has completed. RPCs are used within distributed services to communicate between instances of the service. (RPCs made by different processes in a certain order do not necessarily retain that order.)

It is important to note the conceptual difference between RPCs on one side and datagrams and virtual circuits on the other. RPCs use datagrams as means of communication, they provide additional services as mentioned above, and they are not as flexible as datagrams. RPCs are tailored specifically to the needs of distributed services. Datagrams and virtual circuits are basic means of communication and not tailored to any specific application. They provide no locating services, no authentication, and their interface is more complicated than RPCs. In exchange they can be used for any type of communication between jobs, not just between instances of a distributed service.

Whether an application uses RPCs, datagrams or virtual circuits depends on its particular needs. An application set up as a distributed service will find RPCs the easiest to use. For other uses datagrams or virtual circuits provide the necessary flexibility. In particular datagrams are good for sending brief messages, and virtual circuits for reliably transmitting large amounts of data.

Figure VIII-1-3 gives a simplified picture of the differences between datagrams, virtual circuits, and RPCs.



**Figure VIII-1-3. Three Different Communication Methods**

Both datagrams and virtual circuits link two jobs. To be more precise, datagrams are sent from one *transport service access point* (TSAP) to another. A TSAP represents a binding between

the user of a transport service and the transport service itself. A TSAP object represents a TSAP. In the case of datagrams the TSAP object also serves as a repository for information relating to the TSAP that it represents. This includes buffers and state information. TSAPs are specific to either datagrams or virtual circuits.

In the case of a virtual circuit there is an additional, dynamic level of association between communicating processes, the connection. A transport connection point (TCP) represents an endpoint of the connection. In this case the TSAP represents only the static binding between user and transport service and is used to create and destroy TCPs which represent the dynamic binding. Multiple TCPs can be associated with one TSAP (but only one TSAP with any TCP).

TSAPs are bound to a *TSAP address*. A TSAP address uniquely identifies a TSAP over the entire network. A user who wants to send data through his TSAP to another TSAP must know the TSAP address of the destination TSAP. The remote user can receive the data on his TSAP along with the sender's TSAP address.

TSAP addresses are composed of two parts, a network part which uniquely identifies an instance of the transport services, typically associated with one node, and a *transport service endpoint*. The network part is known as an *NSAP*. An NSAP is the point at which an instance of the transport services is bound to the network level services. Inside the realm of an NSAP an endpoint uniquely identifies a TSAP.

It is convenient for some system-wide services to reserve certain fixed values of endpoints. Those endpoints are called *well known endpoints*. Other endpoints are dynamically allocated by the transport services.

Summarizing, the BiiN™ architecture provides high level interfaces for communications between nodes in a distributed system. Depending on the needs of an application communication services can be used at different levels. However, at all those levels an application does not have to be concerned with the details of the communication protocol.

## VIII-1.5 Review of the Computational Model

In the previous two sections we have outlined naming and communications in a distributed system. Those are the building blocks for a distributed architecture. In this section we shall review the BiiN™ computational model briefly and put it in perspective in a distributed system.

### VIII-1.5.1 Processes, Jobs and Sessions

Processes represent linear threads of computation. Multiple processes may be part of one job. Jobs are the unit of program execution in the BiiN™ system. Jobs, and therefore processes, are confined to a single node. A session may contain many jobs on different nodes. The jobs in the session can communicate with each other or with jobs outside their session. In many ways a job acts like a virtual computer.

### VIII-1.5.2 Active and Passive ADs

Active access descriptors (active ADs) are represented by 33bit words where the 33rd bit, the tag bit, is set. Active ADs are valid inside a node's active memory only. Before an AD can cross node boundaries in a distributed system, it has to be converted to its passive version. A passive AD is a much larger entity than an active AD (about 40bytes). A passive AD is a unique reference on all BiiN™ systems at all times. In order for an object to have a passive AD an AD to the object has to have been stored previously.

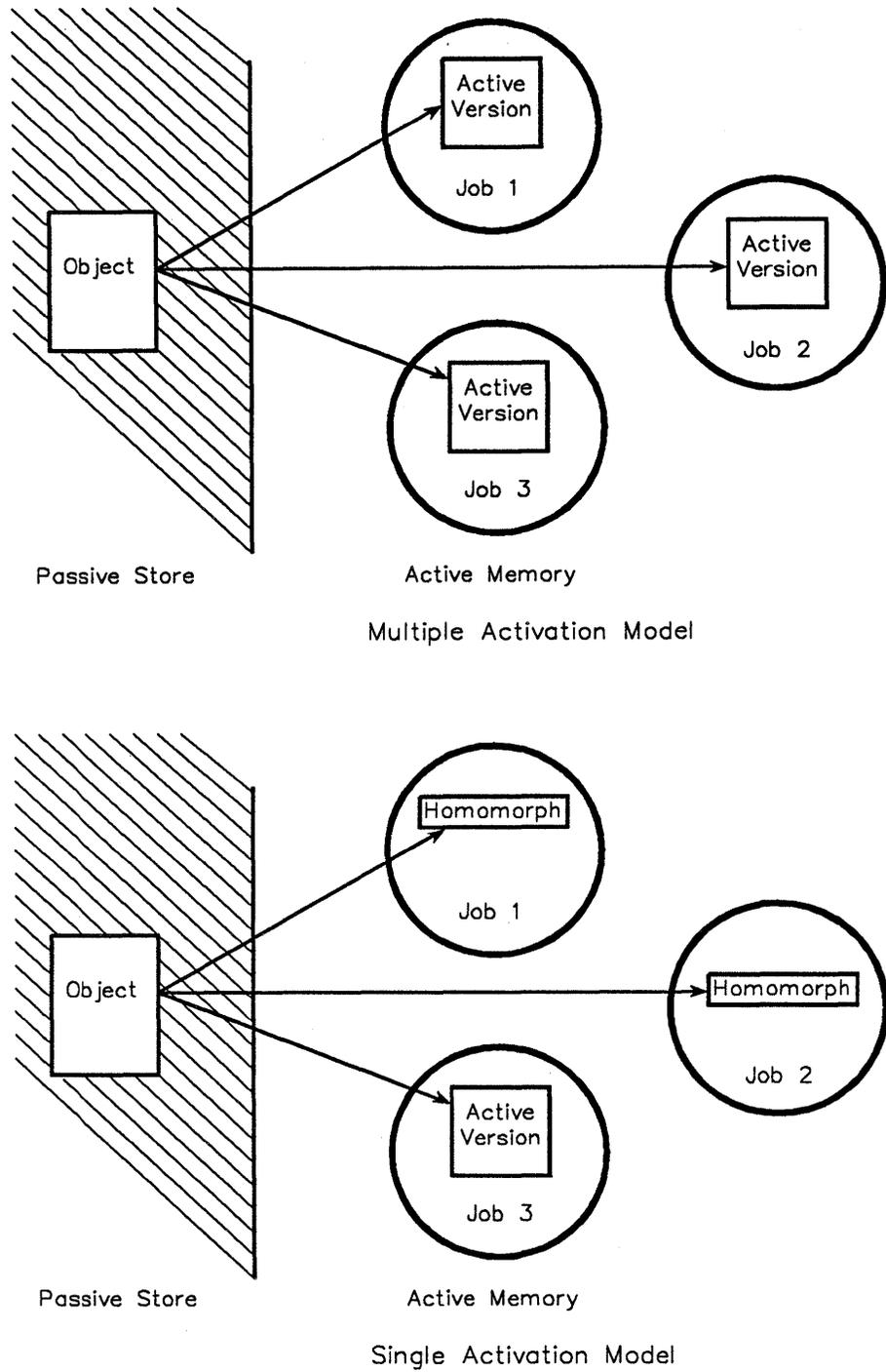
### VIII-1.5.3 Single and Multiple Activation Model

The system supports two different models of activating passive objects (copying passive objects into active memory). In the *multiple activation model* any job activating an object receives an independent active copy of the object. A job can work on its copy and update the passive version from the active version. The multiple activation model is easy to use except for one problem; passive store refuses updates from outdated versions. A job whose update has been refused can handle this situation by requesting a fresh active version, redoing its changes, and attempting another update.

The single activation model avoids the updating problem by allowing only one copy of an object in active memory. One job, the *home job*, receives the active version and all other jobs receive stand-ins, called homomorphs, when activating an object. Those jobs who have homomorphs communicate with the *home job* in order to effect changes on the object. The single activation model is useful for large objects that are used by many jobs simultaneously.

There is an important difference between how global and local objects are treated in both the single and the multiple activation models. Independent of whether in the single or multiple activation model there is always a maximum of one active version per node of a global object. All jobs accessing the global object share this one active version. In the single activation model there is one active version of an object per distributed system, in the multiple activation model there is one active version per node of a global object, and one active version per job of a local object. Independent of the activation model processes within one job always share an active version

Figure VIII-1-4 illustrates the difference between single and multiple activation model. Note that what is shown as active memory in the figure may span several nodes.



**Figure VIII-1-4. Single and Multiple Activation Model**

Distributed services can be built along the lines of either activation model. Very little knowledge of distribution is needed in order to build a multiple activation distributed service. BiiN™ Operating System distributed services take care of the distribution part transparently in this case. Building a distributed service along the lines of the single activation model is more complicated and requires knowledge of the mechanisms of distribution and interjob com-

munication. In the following section we shall present the model of a single activation distributed service.

## VIII-1.6 Single Activation Distributed Services

There are two ways a distributed service can be set up, as a *regional* or as a *universal* distributed service. In both cases the service contains *agents* and *servers*. Requests to the service are directed to an agent. The agent forwards the request to a server which executes it and returns the results to the agent. A universal service has servers and agents on every node of the system. An example of a universal service is the filing service. A regional service has an agent on every node but servers on only a few or even a single node. An example of a regional service is a print service with a printer that is mounted on one particular node, but accepts print jobs on any node.

In a regional service an agent knows the address of at least one server. It does not have to know the address of the server that will actually execute the request; if it directs the request to another server the request will be forwarded until it reaches its destination.

A distributed type manager is also a distributed service. The difference between a type manager and a distributed service in general is that the type manager has representation rights to its objects. It can therefore distinguish between homomorphs and real active versions. This simplifies the model somewhat: There is no need for a strict two level implementation according to the client/server model. In one job the same code can act as the client, in another as the server. The code decides what role it assumes depending on whether it was handed a homomorph or the real active version. If it is handed a homomorph it recognizes that it executes outside the home job. In this case it will act as an agent and forward the request to the server. If it is handed the real active version, that means that it executes inside the home job. In that case it assumes the role of the server and executes the requests directly.

## VIII-1.7 Protection in a Distributed System

Security issues constitute a considerable problem in an open network architecture. In some sense, communications over such a network are similar to radio broadcasts; it is impossible to prevent somebody from broadcasting or from listening to certain broadcasts. If you want to protect broadcasted messages you will have to encrypt them.

The only security mechanisms in effect at the transport level are those that protect TSAPs. Three rights are defined for TSAPs: *Receive*, *Send* and *Control*. Receive rights are necessary to receive messages through a TSAP. Send rights are required to send messages through a TSAP. Control rights are needed to destroy or configure a TSAP.

This protection mechanism does not prevent you from using either datagrams or virtual circuits to send messages to a TSAP on another node or even on your node if you have the TSAP's address. Validation of messages and authentication of the sender is entirely a high level concern. There are two sides to this problem; on one side data in transit should be protected from unauthorized use. On the other side a distributed service's private ADs have to be protected from unauthorized use but at the same time be available to all instances of the service.

Encryption protects data in transit. An application that transmits sensitive data should therefore encrypt that data. There are two solutions to the problem of protecting private ADs.

(Encrypting the data to be transmitted but not protecting private ADs would be like locking the door to one's house but leaving the keys in the lock.) A distributed service can set up its own ID (identical to a user ID). Private ADs can then be stored under well-known pathnames but with an authority list that excludes all IDs but the service's ID. Another solution to the problem is to store the private ADs inside the code of the service, more specifically inside the service's static data object. This simple solution has the disadvantage that all instances of the service have to communicate when one of the private ADs changes.

Remote procedure calls provide authentication and validation services. They also protect data in transit and convert active ADs to their passive version. (An AD still has to be passivated before being transmitted in an RPC -- using an AD on another node if that AD has not been passivated before may have unexpected results.)

When using datagrams or virtual circuits the user has to provide those services himself.

## VIII-1.8 Transparently Distributed Services

The BiiN™ Operating System provides a number of transparently distributed services. With the help of these services a user can take full advantage of a distributed system. They can also be used as tools to build distributed applications. Examples of these services are the filing service, the object service, the concurrent programming service, and the transaction service.

All of the BiiN™ Operating System's distributed services provide transparent access to an entire distributed system's resources. The programmer need not be aware of any of the physical peculiarities of the system.

In the following we shall list some of the most important distributed services:

### VIII-1.8.1 Passive Store

`Passive_Store_Mgt` maintains a system-wide permanent storage. Objects may be stored on volume sets anywhere in the system and can be retrieved from anywhere. Passive store also maintains unique names for all its stored objects. Those names are called *unique identifiers* (UIDs). UIDs are unique not only on one distributed system but on all distributed BiiN™ systems for all times. A volume set may thus be taken from one node in a system to another or even from one distributed system to another. Objects stored on the volume set are always uniquely identified.

### VIII-1.8.2 Directories

`Directory_Mgt` maintains a system-wide directory structure. Directories implement symbolic naming for stored and for active objects. Often `Directory_Mgt` and `Passive_Store_Mgt` will cooperate closely, the former providing the naming mechanism and the latter the actual storing of objects.

However, `Directory_Mgt` may stand on its own: Directory entries can reference any object, active objects as well as passive objects. And while most directories are stored, there are also active-only directories.

The directory structure on each node replicates to a certain extent the entire naming domain the node belongs to. (Certain local aliases may exist on one node, so the directory trees on two nodes are not identical, but their structure is very similar.) The directory structure is not a

simple tree structure: Branches are interconnected and entries may reference backwards in the tree. Thus many different pathnames may reference the same object.

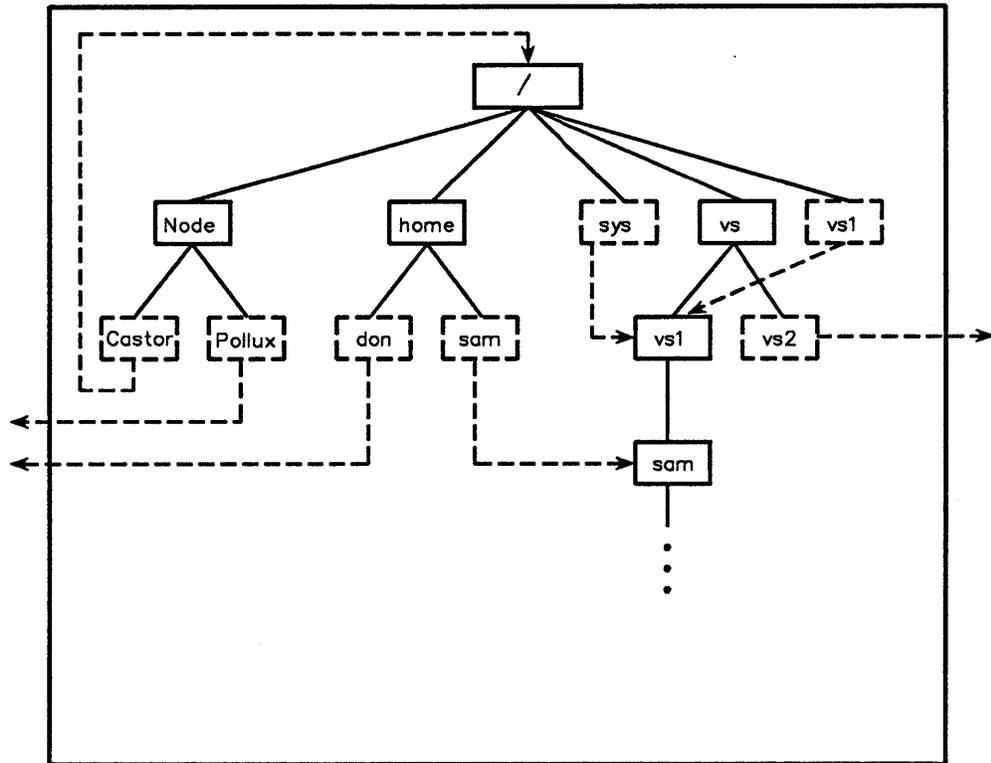


Figure VIII-1-5. Partial View of a Node's Directory Structure

Figure VIII-1-5 shows a partial view of a node's directory structure. (Solid boxes are master entries and dashed boxes represent alias entries.) In particular it illustrates that more than one pathname may reference the same object. For example, `/node/Castor/sys/sam`, `/home/sam`, and `/vs/vs1/sam` all reference Sam's home directory. By the same token `/home/don` references Don's home directory which lives on a different node. This shows that objects with two similar pathnames (`/home/sam` and `/home/don`) do not have to be physically close to each other.

### VIII-1.8.3 IDs

IDs are associated with users. User IDs control access to stored objects and facilitate setting up individualized user environments. A user can be granted access to a distributed system by the system administrator. At that time the system administrator will create a user ID. A user ID grants access to an entire distributed system, not a particular node. Privileges, such as store rights for directories, are granted on a per naming domain basis.

Every process that a user starts and every object that the user stores carries the user ID. IDs are maintained in the clearinghouse's `id` environment.

Very similar to user IDs are *subsystem IDs*. A subsystem ID identifies a subsystem which comprises a collection of domains that share the same stack.

There are other IDs, namely *node IDs*, *volume set IDs*, and *distributed service IDs*. All these IDs play important roles in a distributed system. Node IDs are derived from a hardware module inside a node. They are used in the node environment to map nodes to network addresses.

Volume set IDs uniquely identify volume sets. Together with a time stamp they are incorporated into unique identifiers for objects (passive ADs). Volume set IDs of volume sets mounted locally are cached to allow access to locally stored objects when there is no direct access to the clearing house.

In summary IDs are used whenever certain entities such as users or nodes are to be uniquely identified within a distributed system.

### VIII-1.8.4 Files

Files are among the most important data structures in the BiiN™ architecture. Filing is a distributed service. This means that any file in the system is available anywhere in the system.

Files are global *single activation* objects; files are activated in only one place, namely at their home node. All jobs that use a particular file communicate with the home node when updating the file or reading from the file. Commonly files are large objects. Therefore it makes sense to bring the operation to the data as opposed to bringing the data to the operation.

### VIII-1.8.5 Data Integrity, Synchronization, and Transactions

Data integrity and synchronization across job and node boundaries can be ensured by using transactions. Transactions make operations atomic thus preventing partial completion of operations: Operations included in a transaction either complete successfully or have no effect. Not all operations can be included in a transaction; certain operations are simply irrevocable. Printing is an example: once a page is printed it cannot be made to disappear.

Transactions extend across node boundaries whenever transaction-oriented, distributed BiiN™ Operating System service calls are included in a transaction. Transactions also serve to synchronize access to stored objects; a transaction can reserve an object on its behalf. Then no other transaction can reserve or access the object until the first transaction releases it. Transactions also have a built-in blocking protocol: One transaction can wait for another transaction only if the other transaction is older. (This ordering prevents a circular deadlock situation.)

## VIII-1.9 Summary

Reading this chapter, you have learned that

- distribution makes a collection of BiiN™ nodes connected together, appear as one machine.
- a distributed system is a flexible structure; nodes may be added and removed as the system runs. In particular, distributed services do not depend on the structure of the network that connects the nodes in the system.
- logical organization hides physical organization.
- nodes share a common pool of resources, such as I/O devices, and permanent storage.
- distribution is transparent from the casual user's point of view.

# BUILDING A **2** DISTRIBUTED TYPE MANAGER

---

## Contents

Concepts .....	VIII-2-2
Homomorphs and Active Versions .....	VIII-2-3
The Remote Call .....	VIII-2-3
Synchronizing Access .....	VIII-2-4
Techniques .....	VIII-2-4
Defining The Representation of The Object .....	VIII-2-5
Defining the Homomorph Template .....	VIII-2-6
Setting the Passive Store Attribute .....	VIII-2-6
Defining Buffers for Remote Procedure Calls .....	VIII-2-7
The Is_Call .....	VIII-2-7
The Create_Calls .....	VIII-2-8
Implementing Calls that Require Remote Calls .....	VIII-2-9
Recognizing the Home Job .....	VIII-2-10
Making the Remote Procedure Call .....	VIII-2-10
The Server Stub .....	VIII-2-11
Synchronizing with Transactions and Semaphores .....	VIII-2-12
Initialization .....	VIII-2-12
Private ADs are Hidden in the Static Data Object. ....	VIII-2-13
Creating and Registering the Service .....	VIII-2-14
Creating the Server .....	VIII-2-13
Setting Up the Home Job .....	VIII-2-14
Summary .....	VIII-2-15

This chapter describes how to build a distributed type manager. It focuses on the peculiarities of the *regional service* model. Other features needed for the program, such as transactions, passivating objects, and synchronization are described in chapter VII-6. The basic concepts of the type manager model are treated in chapter VII-3.

Three packages and two initialization procedures are described in this chapter, `Account_Mgt_Ex`, `Distr_Acct_Call_Stub_Ex`, `Distr_Acct_Server_Stub_Ex`, `Distr_acct_init_ex`, and `Distr_acct_home_job_ex`. These packages will be referred to briefly as *core*, *call stub*, *server stub*, *initialization*, and *home job initialization*. All packages and the initialization procedures can be found in Appendix X-A.

## VIII-2.1 Concepts

The type manager described here manages *local* objects on a distributed system that may consist of any number of nodes grouped into any number of naming domains. Active versions of local objects are confined to a single job, and each job activating the object receives its own active version (Some of the active versions may be “ersatz” versions). All processes of one job share the job’s active version. (*Global* objects have only one active version per node shared by all jobs on that node.)

According to the single activation model, the object’s representation is activated in one *home job*. All operations and all synchronization are handled by the home job. Other jobs receive token active versions called *homomorph* and do not operate on the object directly -- they forward all requests to the home job.

As an alternative, a type manager may use the *multiple activation model*: In the multiple activation model every job receives an active version. The multiple activation model is usually simpler to implement, but updating the passive version from multiple active versions has to be carefully coordinated. One can say that the multiple activation model brings the object to the operation, while the single activation model brings the operation to the object: For large objects, such as files for example, the single activation model is more efficient.

The node where the objects are managed is called the *home node*. Any node can be the home node.

The example described manages simple accounts that contain a `long_integer` balance. Accounts can be stored in directories or inside other objects anywhere on the system. When creating an account the application supplies a pathname or an object where the account is to be stored. In order to minimize network traffic it is advisable to store accounts on volume sets mounted at the home node -- the type manager does not enforce this, however. Independently of where accounts are stored they are accessible from any node of the system.

Communications between the home job and any other jobs are implemented by means of remote procedure calls. For more details on the general principle of distribution and RPCs refer to chapter VIII-1.

The type manager provides the following calls:

<code>Is_account</code>	Checks whether an AD references an account.
<code>Create_account</code>	Creates an account and stores it inside an object supplied by the caller.

**Create\_stored\_account**      Creates an account and stores it with a pathname supplied by the caller.  
**Get\_balance**      Returns an account's balance.  
**Change\_balance**      Changes an accounts balance and returns the new balance.  
**Transfer**      Transfers an amount between accounts.  
**Destroy\_account.**      Destroys an account.

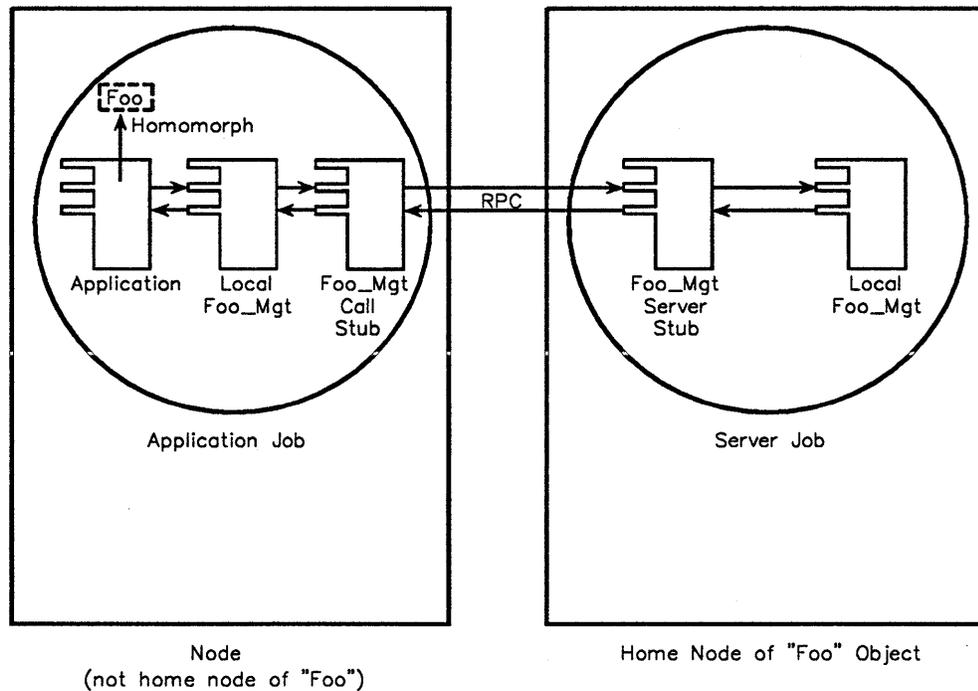
### VIII-2.1.1 Homomorphs and Active Versions

The type manager creates a template that is activated in place of the active version in all jobs but the home job. The template does not have to have the same type as the object it will stand for. The template merely represents a bit pattern that is copied into active memory and become the homomorph. Only the type manager using the representation rights can distinguish between homomorph and active version. The type manager can use the homomorph to store information related to a calling job. Such information can be statistical, for example frequency of calls, or use of resources.

### VIII-2.1.2 The Remote Call

A call to the type manager involves two jobs, the calling job and the type manager's *server job*. The server job is also the home job. The two jobs may live on a single node or on two separate nodes.

Figure VIII-2-1 illustrates the general model of a distributed service implemented with RPCs.



**Figure VIII-2-1. General Model of Communication Using RPCs**

A user program in the calling job holds an AD to the object called FOO. The calling job is not the home job of FOO objects and the AD points to a homomorph. The user program calls the local instance of FOO\_Mgt, the type manager for FOO objects. FOO\_Mgt recognizes from the homomorph that the job is not the home job and forwards the call to its call stub. The call stub packs the parameters into a message buffer and issues an RPC to the server. The initial program in the server is FOO\_Mgt's server stub which calls the local instance of FOO\_Mgt. FOO\_Mgt performs the requested operation and the result is returned.

This is how the general model maps to the special case described here: Account\_Mgt\_Ex acts as the type manager's front end. It corresponds to Local\_Foo\_Mgt in the picture. Applications that want to use the type manager call this package. Thus the distributed implementation looks identical from the outside to the other implementations of the account manager described in Chapters VII-3 and VII-6. All communication between different instances of the type manager on different nodes happens behind the scenes, namely in the call stub, Distr\_Acct\_Call\_Stub\_Ex, and the server stub, Distr\_Acct\_Server\_Stub\_Ex.

The actual work of the type manager is done by Account\_Mgt\_Ex in the home job. This package distinguishes between objects and their homomorphs. When it encounters a real object its operations are identical to the ones of the package described in Chapter VII-6 except for the semaphore synchronization mechanism. (This happens in the home job.) When it encounters a homomorph it hands off the call to the call stub that takes care of the remote calling mechanism. (This happens in an application job.) Thus the remote calling syntax is not part of the type manager's core and can be easily changed without affecting the type manager.

### VIII-2.1.3 Synchronizing Access

The single activation model centralizes synchronization in the home job. Multiple simultaneous requests may be serviced by concurrent processes inside the home job. Processes in the home job share the active version of an account. Access to the active version is synchronized by *semaphores*. Semaphore locking relies on voluntary compliance of all processes. Processes that operate on an object have to call P before touching the object. This will block the calling process if another process has locked the semaphore previously. However, nothing prevents a process from circumventing the semaphore mechanism altogether.

No provisions are made to synchronize access to passive versions since according to the model of this distributed service there is never more than one active version from which the passive version can be updated.

As with all locking mechanisms there is a problem of circular waiting. Transaction come with a built-in blocking protocol that avoids this. For semaphores the problem can be solved by enclosing all semaphores within transactions to use the transaction timeout to break any circular waiting pattern.

## VIII-2.2 Techniques

## Packages Used:

<code>Access_Mgt</code>	Interface for checking and changing rights in access descriptors.
<code>Attribute_Mgt</code>	Provides a way to define general-purpose operations supported by multiple object types or objects, with different type-specific or object-specific implementations.
<code>Authority_List_Mgt</code>	Provides Calls to manage authority lists and to evaluate a caller's access rights to objects protected by authority lists.
<code>Directory_Mgt</code>	Manages directories and directory entries.
<code>Identification_Mgt</code>	Provides operations to manage IDs and ID lists.
<code>Object_Mgt</code>	Provides basic calls for object allocation, typing, and storage management. Defines access rights in ADs.
<code>Passive_Store_Mgt</code>	Provides a distributed object filing system.
<code>RPC_Call_Support.Remote_call</code>	Calls a service that may be at another node.
<code>Semaphore_Mgt.P</code>	Enters / locks / waits at a semaphore.
<code>Semaphore_Mgt.V</code>	Unlocks / leaves / signals a semaphore.
<code>Transaction_Mgt</code>	Provides <i>transactions</i> used to group a series of related changes to objects so that either all the changes succeed or all are rolled back.
<code>User_Mgt</code>	Provides calls to manage a user's protection set and user profile.

## VIII-2.2.1 Defining The Representation of The Object

In addition to other contents the type manager's objects hold two fields: A locking area and an `is_homomorph` boolean. The locking area is needed for semaphore locking and the `is_homomorph` field allows the type manager to distinguish homomorphs from active versions. The example from the core shows the account layout which contains the `long_integer` `balance` plus those two fields:

```
96     type account_rep_object is
97         -- Representation of an account.
98         record
99             lock: Semaphore_Mgt.semaphore_AD;
100             -- Locking area
101             is_homomorph: boolean;
102             -- If false identifies the object
103             -- as the active version; if true
104             -- as a homomorph.
105             balance: Long_Integer_Defs.long_integer;
106             -- Starting balance.
107         end record;
```

The locking area is null in the passive version but is filled in with an AD to a semaphore when the object is activated.

The object layout is specified with an address clause. This is necessary since the type manager relies on the layout of the object in memory: Record layout in memory may vary from compiler version to compiler version.

```

108     FOR account_rep_object USE
109     record AT mod 32;
110         lock      at 0  range 0 .. 31;
111         is_homomorph at 4  range 0 .. 7;
112         balance    at 8  range 0 .. 63;
113     end record;

```

### VIII-2.2.2 Defining the Homomorph Template

The homomorph template acts as a bit pattern that is copied into active memory in place of an active version. In the simplest case the template is defined with `is_homomorph` set to true while in the active version `is_homomorph` is false. Other information can be stored in the template. In particular, the type manager can use the template to store resource or statistical information pertaining to the calling job. The following example is from the initialization procedure `Distr_acct_init_ex`. (This procedure can be found in its entirety in Appendix X-A. In our example only the `is_homomorph` field is used. The other fields are initialized to null.

```

90     type template is
91     record
92         dummy_word0:  System.untyped_word;
93         is_homomorph: boolean;
94         dummy_word1:  System.untyped_word;
95         dummy_word2:  System.untyped_word;
96     end record;
97
98     FOR template USE
99     record AT mod 32;
100         dummy_word0 at 0  range 0 .. 31;
101         is_homomorph at 4  range 0 .. 7;
102         dummy_word1 at 8  range 0 .. 31;
103         dummy_word2 at 12 range 0 .. 31;
104     end record;
105
106     type homomorph_AD is access template;
107     pragma access_kind(homomorph_AD, AD);
108
109     homomorph: homomorph_AD;
.
.
.
149     -- 2. Allocate and initialize homomorph template:
150     --
151     homomorph := new template'(
152         dummy_word0 => System.null_word,
153         is_homomorph => true,
154         dummy_word1 => System.null_word,
155         dummy_word2 => System.null_word);

```

Note that template does not even have the same type as the object proper.

### VIII-2.2.3 Setting the Passive Store Attribute

In order for `Passive_Store_Mgt` to transparently substitute a homomorph for active versions in all jobs but the home job, the `homomorph` field in the `PSM_attributes_object` has to be non-null. If the field is not null `Passive_Store_Mgt` uses the AD contained in that field as a reference to a template to substitute for the object. The following excerpt from the initialization shows how the passive store attribute defined and how it is initialized:

```

73     passive_store_impl:
74         Passive_Store_Mgt.PSM_attributes_AD;
75         -- Implementation of passive store attribute
76         -- for accounts.
. . .
145     -- 1. Allocate new passive store attribute implementation:
146     --
147     passive_store_impl := new
148         Passive_Store_Mgt.PSM_attributes_object;
. . .
156
157     -- 3. Initialize passive store attribute implementation:
158     --
159     passive_store_impl.homomorph := Untyped_from_homomorph(homomorph);
160
161     passive_store_impl.reset :=
162         Refuse_reset_active_version_Ex.
163         Refuse_reset_active_version'subprogram_value;
164
165     passive_store_impl.copy_permitted := false;
166
167     passive_store_impl.locking_area_start := 0;
168     passive_store_impl.locking_area_end := 0;
169     -- Area in account where semaphore AD will be
170     -- stored when account is activated.

```

The PSM\_attributes\_object also specifies where the locking area is and that accounts cannot be copied.

## VIII-2.2.4 Defining Buffers for Remote Procedure Calls

Buffers are necessary for both parameters and results in remote procedure calls. The following example from the server stub defines one buffer type for both parameters and results.

```

14     type buffer is
15         -- Buffer used for remote calls.
16         record
17             first_word: System.untyped_word;
18             second_word: System.untyped_word;
19             amount: Long_Integer_Defs.long_integer;
20         end record;

```

The buffer has room for two ADs and one long\_integer. This is the maximum transmitted in one single call. (Transfer). Note again that an address clause is used to fix the layout of the buffer in memory:

```

23     FOR buffer USE
24         record AT mod 32;
25             first_word at 0 range 0 .. 31;
26             second_word at 4 range 0 .. 31;
27             amount at 8 range 0 .. 63;
28         end record;

```

## VIII-2.2.5 The Is\_Call

### Calls Used:

Object\_Mgt.Retrieve\_TDO  
Returns an object's type.

No inter-job communication is necessary for the Is call: The object itself is not involved in the call at all: The type manager only retrieves a TDO and compares it to its own TDO. For this reason the the core does the work directly as can be seen in the following example:

```

139     return obj /= System.null_word
140         and then
141         Object_Mgt.Retrieve_TDO(obj) = account_TDO;

```

## VIII-2.2.6 The The Create\_ Calls

### Calls Used:

```

Transaction_Mgt.Get_default_transaction
    Returns the transaction on top of the transaction stack.

Transaction_Mgt.Start_transaction
    Starts a transaction and pushes is it on the stack.

Transaction_Mgt.Commit_transaction
    Commits a transaction.

Transaction_Mgt.Abort_transaction
    Aborts a transaction.

```

The type manager uses the `is_homomorph` field to distinguish between the home job and any other job. This method fails with the `Create_` calls since there is neither a homomorph nor an active version to check before the object has been created. (Remember that `is_homomorph` is false in the home job and true in all other jobs.)

For this reason any job can create objects. This means that in both `Create_` calls the core does the operation directly. In order to prevent multiple active versions the new object is deallocated as soon as it has been created and passivated. The three steps, *Allocate*, *Passivate* and *Deallocate* are enclosed in a transaction. Thus the `Create_` calls cannot succede partially leaving unwanted active versions.

The following excerpt from the core shows these essential part of the `Create_account` call:

```

341     if Transaction_Mgt.Get_default_transaction =
342         null then
343         Transaction_Mgt.Start_transaction;
344         trans := true;
345     end if;
346
347
348     begin
349         -- This block controls the scope of
350         -- the exception handler.
351
352         -- 5. Create the master AD:
353         --
354         Directory_Mgt.Store(
355             name => master,
356             object => account_untyped,
357             aut => authority);
358
359         -- 6. Passivate the representation of the account:
360         --
361         Passive_Store_Mgt.Update(account_rep_untyped);
362
363         -- 7. Deallocate the active version of the
364         -- account:
365         --
366         Object_Mgt.Deallocate(account_rep_untyped);
367
368         -- 8. Commit any local transaction.
369         --
370         if trans then
371             Transaction_Mgt.Commit_transaction;
372         end if;
373
374     exception
375
376         -- 9. If an exception occurs, abort any local
377         -- transaction, deallocate the account and
378         -- reraise the exception:
379         --
380         when others =>
381             if trans then
382                 Transaction_Mgt.Abort_transaction;
383             end if;
384             Object_Mgt.Deallocate(account_rep_untyped);
385             RAISE;
386
387     end;

```

The type manager provides a second `Create_call` named `Create_stored_account`. While the `Create_account` call simply allocates a new account, the `Create_stored_account` also stores the account with a pathname supplied by the caller. The calling mechanism is identical to the `Create_account` call and the operation proper in the core is identical to the one described in Chapter VII-6.

## VIII-2.2.7 Implementing Calls that Require Remote Calls

Except for the three calls discussed in the previous sections, namely `Is_account`, `Create_account`, and `Create_stored_account`, all calls of the type manager require remote calls. The remote call has the same calling syntax for jobs on one node and for jobs on different nodes. When a remote call is needed the core hands it off to the call stub that takes care of it.

### VIII-2.2.7.1 Recognizing the Home Job

The `is_homomorph` field is used to recognize the home job. In the home job the type manager will see `is_homomorph` as false, in any other job as true:

```

458     if account_rep.is_homomorph then
459
460         -- 2. We have a homomorph:
461         --
.
.
.
468     else
469
470         -- 3. We are in the home job for accounts:
471         --
.
.
.
530     end if;
```

When `is_homomorph` is true a remote procedure call has to be made and the core hands the call off to the call stub. When `is_homomorph` is false the operation can be done directly.

### VIII-2.2.7.2 Making the Remote Procedure Call

#### Calls Used:

`RPC_Call_Support.Remote_Call`  
 Makes an RPC to an RPC service.

A remote procedure call is a means of communication between two jobs. All information passed between the jobs is contained in buffers.

Both the caller and the callee have to agree on the format of the buffers. Once transmitted to another job a buffer is no more than a pattern of bits that has to be interpreted correctly. Two buffers are required, one for parameters and one for results. This is shown in the following example from the call stub:

```

72     parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
```

For the type declaration of `buffer` refer to section VIII-2.2.4. Before the call the calling job packs parameters into the buffer and after the call results are unpacked from the results buffer:

```

82     parameters := Distr_Acct_Server_Stub_Ex.buffer' (
83         first_word => account_untyped,
84         second_word => System.null_word,
85         -- irrelevant
86         amount     => Long_Integer_Defs.zero);
87         -- irrelevant
.
.
.
101     current_balance := results.amount;
```

The layout of the buffer is designed for maximum required size. Not all slots are needed in all calls.

When making a remote call the calling job specifies the service to be called. This directs the call to a server job where the service is currently registered. Optionally a node ID can be specified in the call. This will direct the call to the server on the specified node. This option can be used when multiple servers exist and one in particular is to be chosen.

The calling job also specifies an ordinal value called `target_proc`. The main package's calls are assigned an ordinal value and depending on the value of `target_proc` in the call the associated procedure or function in the main package is called.

In the case of our example the assignments are as follows:

- 0            Used to initialize the server job.
- 1            Get\_balance.
- 2            Change\_balance.
- 3            Transfer.
- 4            Destroy\_account.

Note that `Is_account`, `Create_account`, and `Create_named_account` are not assigned an ordinal value. These functions are always performed locally and do not require a remote call.

The addresses and sizes of the buffers are also specified, and a boolean parameter is used to indicate that ADs are being transmitted. ADs have to be converted in a remote call. Indicating that no ADs are present speeds up the call.

The following example shows the syntax of the remote call:

```

91        length := RPC_Call_Support.Remote_call(
92            service        => service,
93            target_proc   => 1,
94            param_buf     => parameters' address,
95            param_length  => parameters' size,
96            ADs_present   => true,
97            results_buf   => results' address,
98            results_length => results' size);

```

As you can see from the above assignments this remote call will result in `Get_balance` being called by the server. The variable `length` contains the actual length of the results buffer. This is useful when the result buffer's length varies. The variable is not used here since the results buffer in this example has a fixed length. In order to see where `service` comes from refer to section VIII-2.2.9.3.

### VIII-2.2.7.3 The Server Stub

#### Calls Used:

`RPC_Mgt.Server_stub`

Template for a stub procedure to be called by the server.

When the server is called it executes an initial procedure called the *server stub*. The procedure declaration of the server stub matches a template, namely `RPC_Mgt.Server_stub`. The type manager provides the implementation of the template. The declaration looks like this:

```

21        procedure server_stub(
22            target_proc:            System.short_ordinal;
23            version:                System.ordinal;
24            param_buf:             System.address;
25            param_length:          System.ordinal;
26            results_buf:           System.address;
27            results_length:        in out System.ordinal;
28            ADs_returned;          out boolean)

```

Depending on the value of `target_proc` the server stub interprets the parameter buffer and makes the requested call. In the example the server stub is coded with a case statement:

```

59     case target_proc is
77         when 2 => account_one_untyped := parameters.first_word;
78             amount :=
79                 Account_Mgt_Ex.Change_balance(
80                     account =>
81                         account_one,
82                     amount =>
83                         parameters.amount);
84             results := buffer'(
85                 first_word => System.null_word,
86                 -- Irrelevant.
87                 second_word => System.null_word,
88                 -- irrelevant.
89                 amount => amount);
90             ADs_returned := false;
117        when others =>
118            RAISE System_Exceptions.operation_not_supported;
119    end case;

```

Note that the server stub calls the core. This does not result in an infinite loop by triggering another remote call since this call takes place inside the home job. The core performs the requested operation and returns the result.

### VIII-2.2.8 Synchronizing with Transactions and Semaphores

Access to account objects is centrally synchronized in the home job. In the home job multiple concurrent processes may access an account. Concurrent processes in the home job use *semaphore locking* to reserve the active version of an account. More details on synchronization and semaphore locking can be found in Chapters VI-1 and VI-2.

- Access to the passive version of an account is not synchronized since no more than one active version of an account exists. Here lies one of the advantages of the single activation model.
- Transactions are used to prevent semaphore deadlock and to protect passive versions from incomplete updates. Please note that the transaction timeout period is set when the system is configured.
- Outside the home job no synchronization is required since object representations are never touched outside the home job.

### VIII-2.2.9 Initialization

This type manager is a distributed service and spans at least two jobs. Two procedures are needed to initialize the type manager, `Distr_acct_init_ex`, and `Distr_acct_home_job_ex`. Both procedures can be found in Appendix X-A.

The following three points should be considered when the service is initialized:

- Depending on how the service is set up it may or may not create a lot of network traffic. The worst possible situation arises when the type manager's image module is stored on one node, the stub on another, the home node is still another node, and accounts are stored all over the network. Objects should be stored close to the home node, ideally on the home node itself.
- The type manager model of protection can only be fully realized if the code is linked into its own separate domain. In particular, the type manager's private ADs are hidden in the static data object with the help of the BiIN™ Ada pragma `bind` at link-time. Therefore the static data object should not be accessible to any other module but the type manager.

- As part of the initialization the server is created and installed. When installing the server the caller can specify an SSO from which the server is scheduled and a cpu time limit. If those parameters are not explicitly specified (as in our example) the server is allocated from the caller's SSO and inherits the caller's time limit. For this reason the type manager should be installed from a privileged ID. Otherwise the server may experience resource exhaustion at some unexpected time.

### VIII-2.2.9.1 Private ADs are Hidden in the Static Data Object.

The ADs for the TDO and the service are stored in the type manager's module, more precisely the static data object. This is necessary since these objects are created by the `Distr_acct_init_ex` procedure and stored with an authority list that includes only the developer thus making them inaccessible to the user of the type manager. They are retrieved when then type manager is linked. For this reason linking has to be done with the developer's ID. A third AD, the one for the homomorph, is stored by the `Distr_acct_init_ex` procedure in the passive store attribute.

The objects referenced by these ADs should only be created once. For example: One type is identified by exactly one TDO. There cannot be two TDOs referencing the same type. By definition two objects referencing different TDOs have different type. (If a TDO is destroyed it can of course be replaced by a new one.) By the same token there is only one distributed service, and one homomorph template. For this reason `Distr_acct_init_ex` should only be executed once on a distributed system, prior to linking the type manager. Then, after the type manager has been linked, `Distr_acct_home_job_ex` should be executed to initialize the server.

After these steps have been executed the main package can be called by an application. The following sections explain the steps in the initialization:

### VIII-2.2.9.2 Creating the Server

#### Calls Used:

`RPC_Mgt.Create_RPC_server`  
Creates an RPC server.

`RPC_Mgt.Install_server`  
Installs an RPC server and returns an AD to the server job.

The following call creates a server on the local node:

```

61  server: constant RPC_Mgt.RPC_server_AD :=
62      RPC_Mgt.Create_RPC_server;
63      -- Server for accounts.
64
65  server_job: Job_Types.job_AD;
66      -- Installed server job.
. . .
193  -- 7. Install server:
194  --
195  server_job := RPC_Mgt.Install_RPC_server(
196      server => server);

```

Four optional parameters can be specified with the call (default values are given in parentheses): A maximum (2) and a minimum (2) number of processes for the server, a maximum number of services (1) that can be registered with the server, and a naming domain

with which the server will associate. (naming domain of the creating node). Note that two steps have to be taken to create the server, first it has to be created, second it has to be installed. Installing the server creates the server job. This example package should first be called by a job with unlimited resources, or an unlimited SSO should be specified in this call.

### VIII-2.2.9.3 Creating and Registering the Service

#### Calls Used:

`RPC_Mgt.Create_RPC_service`

Creates an RPC service and returns an AD to the service.

`RPC_Mgt.Register_RPC_service`

Registers a service with a server. More than one service can be registered with one server.

An RPC service is transparently accessible. That means that the caller does not have to know the physical address of the server, but can specify the service and the call will be routed transparently. The service is not automatically associated with a server. In order to bind a service to a server the service has to be registered with the server. Multiple services can be registered with one service. Exactly how many is determined by the `max_services` parameter in the `RPC_Mgt.Create_RPC_Server` call. The following excerpt from the initialization shows these two calls:

```
198 -- 8. Create the service:
199 --
200 service := RPC_Mgt.Create_RPC_service(
201     server => server);
202
```

When registering a service the caller specifies a stub procedure. That stub procedure matches the `RPC_Mgt.Server_stub` template. The server executes the stub procedure registered with one service when it receives a remote call from that service.

### VIII-2.2.9.4 Setting Up the Home Job

#### Calls Used:

`Passive_Store_Mgt.Set_home_job`

Establishes the calling job as home job for objects of one type. Undoes the effect of any previous call by another job.

Before the service can be called the server has to become the home job for account objects. This is achieved by executing the `Distr_acct_home_job_ex` procedure. The following excerpt shows this procedure in its entirety:

```

27 begin
28   -- Set up server as home job
29   --   by calling procedure ``0``:
30   --
31   parameters := Distr_Acct_Server_Stub_Ex.buffer'(
32     first_word => account_TDO_untyped,
33     -- account TDO
34     second_word => System.null_word, -- Irrelevant.
35     amount      => Long_Integer_Defs.zero);
36   -- Irrelevant.
37
38   length := RPC_Call_Support.Remote_call(
39     service      => service,
40     target_proc  => 0,
41     -- Server will call Passive_Store_Mgt.Set_home_job.
42     param_buf    => parameters'address,
43     param_length => parameters'size,
44     ADs_present  => true,
45     results_buf  => results'address,
46     results_length => results'size);
47
48   end Distr_Acct_Home_Job_Ex;

```

This procedure makes a remote call specifying 0 as the target procedure. In turn, the server stub which is running in the server job calls `Passive_Store_Mgt.Set_home_job` when 0 is specified as the target procedure:

```

59   case target_proc is
60     when 0 => account_TDO_untyped := parameters.first_word;
61             Passive_Store_Mgt.Set_home_job(
62               tdo => account_TDO);
63             ADs_returned := false;
64
65   end case;
119

```

Note that the `Passive_Store_Mgt.Set_home_job` procedure has to call and cannot call `Set_home_job` directly since only the server executes exclusively in the server job.

## VIII-2.3 Summary

From this chapter you should have learned how to build a distributed type manager. The example described has the following properties.

- The type manager acts as a distributed service.
- Objects are managed in one home job.
- Local instances of the service communicate with the home job by remote procedure calls.

More specifically you should have learned how to

- set up the object's representation including a locking area and an `is_homomorph` field.
- initialize the passive store attribute to implement the single activation model.
- define a template that is activated instead of the object's active version in all jobs but the home job.
- define buffers for remote calls.
- create and install the server.
- create and register the service.

- **define the call stub.**
- **recognize a homomorph.**
- **pack and unpack buffers.**
- **make remote calls.**

# Part IX

## Device Services

---

This part of the *BiiN™/OS Guide* provides information about device drivers and device managers. This part contains one chapter:

### Understanding Device Managers and Device Drivers

Describes the low-level I/O model and general architecture of device managers and drivers.

Device Services contains the following services and packages:

*Device driver service:*

- CP\_IO\_Defs
- CP\_Mgt
- CP\_Resources
- DD\_Support
- Handling\_Support
- Interrupt\_Handling\_Support
- IO\_Messages\_Defs
- IO\_Messages\_Ops
- Region\_3\_Support
- SCSI\_Bus\_Dependent\_Defs
- SCSI\_Record\_Defs

*shared queue service:*

- Cluster\_Service
- IO\_Shared\_Queues

*asynchronous communication service:*

- Async\_Defs

*mass storage service:*

- Bus\_Independent\_Disk\_Defs
- Bus\_Independent\_Streamer\_Defs
- Bus\_Independent\_Tape\_Defs
- Mass\_Store\_Reply\_Codes
- MS\_Configuration\_Defs

*SCSI service:*

- CP\_SCSI\_Defs
- CP\_SCSI\_Mgt
- SCSI\_Bus\_Dependent\_Defs

*subnet service:*

- Carrier\_Mgt
- Subnet\_CL\_AM
- Subnet\_CO\_AM
- Subnet\_Defs
- Trace\_Defs
- Trace\_Support

*HDLC service:*

- HDLC\_Mgt

*LAN service:*

- CSMA\_CD\_Defs
- Ethernet\_LAN\_Mgt
- IEEE8023\_LAN\_Mgt



# UNDERSTANDING DEVICE MANAGERS AND DEVICE DRIVERS

# 1

## Contents

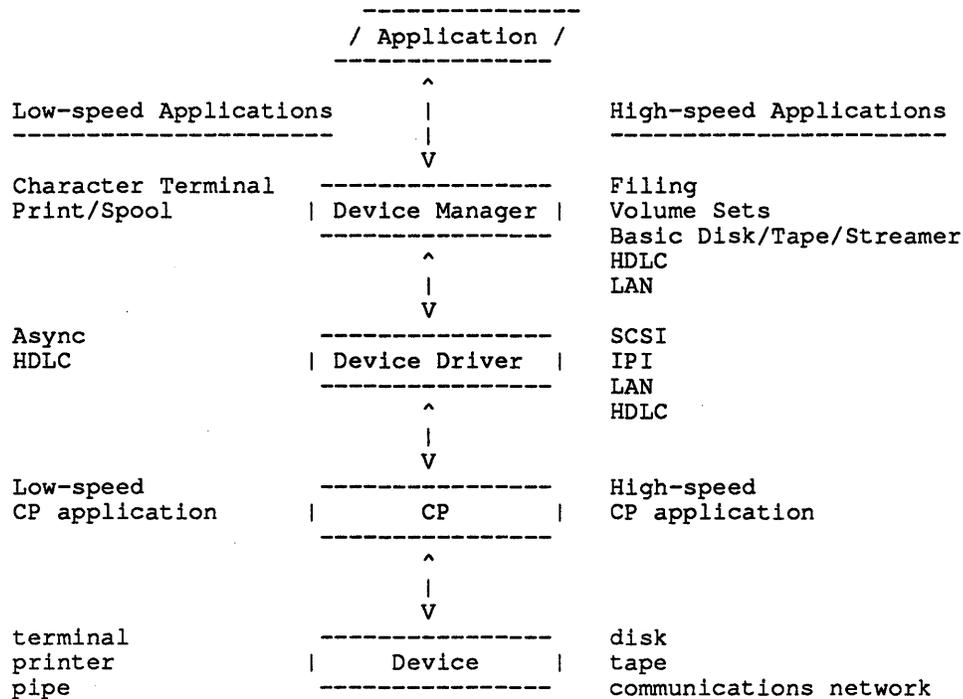
Concepts .....	IX-1-3
I/O Model .....	IX-1-3
Access Methods .....	IX-1-4
Device Managers .....	IX-1-4
Device Drivers .....	IX-1-4
Device Classes .....	IX-1-4
I/O Mechanisms .....	IX-1-4
The I/O Messages Mechanism .....	IX-1-5
Data Transfer Via the I/O Messages Mechanism .....	IX-1-6
I/O Recovery Agent .....	IX-1-8
Data Transfer Via the Shared Queues Mechanism .....	IX-1-8
Clusters and Cluster Servers .....	IX-1-9
Administrative Interface .....	IX-1-9
Device Driver Example .....	IX-1-9
Summary .....	IX-1-13

This chapter describes device manager and device driver architectures.

**Packages Used:**

- IO\_Messages\_Defs  
Defines the I/O messages mechanism interface.
- IO\_Messages\_Ops  
Provides driver-independent I/O message calls for device drivers.
- Cluster\_Service  
Manages cluster servers.
- IO\_Shared\_Queues  
Defines the shared queues I/O mechanism.
- Port\_Mgt  
Provides fast interprocess communication within a job.
- CP\_Mgt  
This package defines the types used in communicating with a Channel Processor (CP). This includes the format of various data structures used by a Channel Processor. Furthermore, the Send\_to\_CP operation is defined here. It forwards an I/O message to a Channel Processor for service.
- DD\_Support  
Supports device drivers.
- Interrupt\_Handling\_Support  
Manages interrupt handlers.
- Handling\_Support  
Provides calls to save and restore global registers.
- Region\_3\_Support  
Provides a call for installing macrocode in Region 3.
- Unsafe\_Object\_Mgt  
Provides special object allocation and deallocation calls.
- Countable\_Object\_Mgt  
Supports type managers of countable global objects.

The relationship between an application, a device manager, device driver and a device is shown in Figure IX-1-1.



**Figure IX-1-1. Device Environment**

## IX-1.1 Concepts

This section introduces methods, concepts and terminology necessary for understanding the role of device managers and device drivers in communicating with devices.

A typical I/O process involves the following actions:

- A device object is opened by an application using an `Open` access method call prior to sending data to a device.
- An I/O data transfer mechanism combined with a device class forms an I/O interface through which the device manager can communicate with a device driver, a CP (Channel Processor), and ultimately a device.

This chapter describes two I/O data transfer mechanisms which may be used to form an I/O interface, and describes the roles of device managers and device drivers.

## IX-1.2 I/O Model

The primary elements of the I/O model are device objects, device managers and opened device objects. A *device object* is a typed object that represents a device. A single device object is associated with each device in the system. A *device manager* is a type manager that controls access to a device. Devices include files, magnetic tapes, terminals, and pipes. An *opened device object* is a typed object that represents an input/output connection between a device manager and a device. Zero, one or more opened device objects may exist for the same device. Opened device objects are analogous to I/O channels on other systems.

### IX-1.2.1 Access Methods

Applications interact with device managers via access methods. An *access method* is a collection of procedures which provide a device-independent interface to perform I/O. A device object has associated with it the implementations of the access methods supported by that device. An access method is a type attribute of device objects and opened device objects.

To perform device operations, an application selects an access method and passes a device object to its `Open` operation. `Open` returns an opened device object representing an opened device channel. The opened device object is passed as a parameter when making access method calls.

A device can be simultaneously accessed by more than one access method. This is convenient, for example, when a call is made to a library function that internally uses a different access method.

### IX-1.2.2 Device Managers

A *device manager* is a type manager of a specific type of device which provides a high-level interface through which an application can communicate with a device.

### IX-1.2.3 Device Drivers

A *device driver* provides a device manager with access to a physical device. In the BiiN™ Series 60/80, a device driver is connected to its device through a CP. Device drivers are simplified by being connected to a CP since drivers do not need to provide such functions as handling interrupts and issuing device commands.

### IX-1.2.4 Device Classes

A *device class* is a specification which defines the device-specific details necessary to access a class of device using an I/O mechanism. Device classes are used by device managers and implemented by device drivers. Device class specifications provide opening parameters (initial values for the `IO_Shared_Queues.device_state_rep`), command codes used in the Common Part of the I/O message (`IO_Messages_Defs.IO_message.command_code`), and reply codes used in the Common Part of the I/O message (`IO_Messages_Defs.IO_message.reply_record`). A device class specification used with an I/O mechanism forms a device-specific I/O interface through which device managers and device drivers may communicate on behalf of devices of the device class.

### IX-1.2.5 I/O Mechanisms

The BiiN™ Operating System defines two I/O mechanisms available to device drivers:

- I/O messages
- Shared queues.

I/O messages supports high-speed, block-oriented data transfer. shared queues supports low-speed, character-oriented data transfer. These design characteristics make the I/O messages mechanism more suitable for disk I/O and network communications, and the shared queues mechanism more suitable for I/O to terminals.

Although these mechanisms are designed to provide communications between device managers and device drivers, they may also be used for device managers to communicate with other components such as other device managers. For example, a terminal might be connected to a system via a terminal concentrator on a network. The terminal device manager could use the shared queues mechanism to talk to a software component that converts the shared queues protocol to subnet message-based requests.

These mechanisms provide data transfer. The I/O messages mechanism is also used in an administrative interface.

## IX-1.2.6 The I/O Messages Mechanism

The I/O messages mechanism consists of operations that device managers can call to support data transfer, including administrative functions, with high-speed, block-oriented devices such as disks, tapes and high-speed communications.

### The I/O Message

An *I/O message* is an object consisting of four parts:

- Common part
- Device Driver part
- Device Manager part
- Buffer Description part.

The *Common part* of the I/O message has fields at fixed offsets that are visible to device managers, device drivers and CPs. It contains information about an I/O request including the type of request, the device involved and the number of buffers associated with the message.

The Common part contains pointers, offsets and IDs for locating the reply mechanism, the physical device, the CP, the beginning of the buffer description array and the Common part itself. Other fields identify the type of reply mechanism used, usage information about the buffer descriptions, request and reply priorities, error ID, command code and any device-specific parameters.

The *Device Driver* part follows the Common part, is variable in size depending on the device class, and is reserved for use by device drivers and CPs.

The *Device Manager* part follows the Device Driver part, is variable in size depending on the device class and is reserved for use by the Device Manager.

The *Buffer Description* part contains an array of buffer descriptions. Each buffer description contains the size and address of its buffer and use indicators. Since this array does not begin at a fixed location within the message, the Common part contains an offset field with which device drivers and device managers can locate the beginning of the array of buffer descriptions.

I/O messages may have several buffers. The buffers must be allocated in frozen memory. A device manager must not modify the buffers between the time a request is issued and the time the I/O message is returned to the device manager.

The contents of a buffer depend on the type of request and the device class associated with the I/O message. (The semantics assigned to each request are described in the device class specification/package.) Some I/O messages might not reference any buffers at all, such as a

device-specific *reset* request. Other requests such as a Read normally require at least one buffer.

### Reply Mechanism

The device manager decides the reply mechanism, interrupt reply procedure or reply port from which it will receive its returned I/O messages. The selected mechanism is specified by the values in `reply_port_or_proc` and `type_of_reply`.

The *interrupt reply procedure* is called by an interrupt handler, and performs post-processing of the serviced I/O message such as setting `error_id` and `total_returned_length`. A template for this procedure is provided via `IO_Messages_Defs.Process_IO_message`. The *reply port* mechanism is an inter-process communications mechanism on which I/O messages can be enqueued.

The interrupt reply procedure has the advantage of not causing a context switch, but does execute an interrupt handler. Thus the implementation of an interrupt reply procedure must comply with all constraints placed on interrupt handlers (see `Interrupt_Handling_Support` for a list of interrupt handler constraints). Most BiiN™ Operating System device managers use the I/O reply port mechanism.

## IX-1.3 Data Transfer Via the I/O Messages Mechanism

Most systems will employ CP-connected devices because I/O via CPs is available and efficient for the more common protocols (see *BiiN™/OS Reference Manual* for a list of supported devices). Using a CP also greatly simplifies the tasks which must be performed by a device driver.

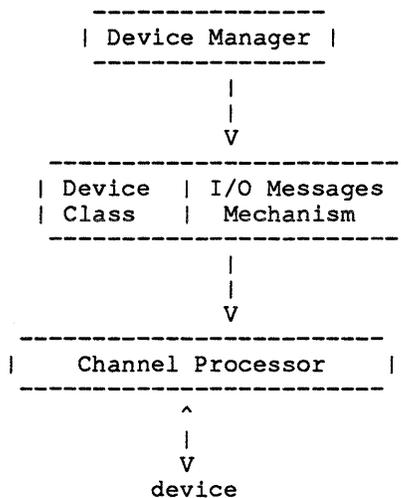


Figure IX-1-2. Device Driver using the I/O messages Mechanism

Data transfer to a CP-connected device using the I/O messages mechanism can be done via the following steps:

1. The application calls an access method `Open` to create an opened device.

2. The device manager allocates the data buffers and buffer descriptions (optionally using `DD_Support.Set_buffer_description`), and fills in the following fields:

- `queuing_space`
- `reply_port_or_proc`
- `total_request_length`
- `type_of_reply`
- `reply_priority`
- `io_msg`
- `used_buffers`, optional
- `max_buffers`
- `command_code`
- `buffer_descr_offset`
- `device_specific_params`

The device manager may optionally allocate a pool of I/O messages by repeatedly creating I/O messages and calling `DD_Support.Register_IO_message`. A pool of I/O messages may be shared by several devices.

3. The device manager calls `IO_Messages_Ops.Ops.Issue_request` to forward the I/O message to a device for service.
4. Any time after the I/O message has been sent to the device (Step 2), the device manager calls `Port_Mgt.Receive` or `Port_Mgt.Conditional_receive` to receive the message from the reply port, if a reply port was selected as the reply mechanism. If the selected reply mechanism is an interrupt reply procedure the message receipt method is be defined by the procedure.
5. The device driver gets access to the I/O message, and fills in the following fields of the Common part of the I/O message:

- `phys_dev`
- `request_priority`, optional
- `cp_id`
- `device_id`

The device driver also fills in the following fields defined in the Device Driver part of the I/O message required by the CP:

- `interrupt_q_addr`
- `phys_buf_desc_addr`

`interrupt_q_addr` is the physical address of an interrupt queue head. It identifies the return path from a CP to a CPU after the message has been serviced.

`phys_buf_desc_addr` is the physical address of the buffer description array.

The device driver can call an access method's `Get_device_info` call to acquire information for some of these fields. It can also place other information in the undefined section of the Device Driver part for its own use.

The device driver must set these fields because a device manager will generally use one pool of I/O messages to issue requests for all the devices it manages. Since a device manager may manage some devices that are connected to the system by CPs and others that are directly connected, several different device drivers may service a single device manager's I/O requests. They may use the Device Driver part of the I/O messages differently. Therefore, a device driver must set all the fields in an I/O message that specify device information.

6. The device driver issues an I/O request to the CP by calling `CP_Mgt.Send_to_CP`.
7. After the CP has finished servicing the I/O request, it writes the following results in the I/O message:
  - `error_id`, if an error occurred.
  - `total_returned_length`
  - `reply_record`
8. The CP sends the I/O message to the interrupt queue specified by `interrupt_q_addr` and generates an interrupt.
9. The CPU interrupt handler which processes CP-generated interrupts, returns the I/O message to the reply mechanism specified in the I/O message (`Port_Mgt.Send` for a reply port).
10. The device manager may continue issuing requests for service, calling receive operations and logging any errors.
11. When the device manager completes and needs no further access to the device, it waits for pending I/O requests to complete (or cancels them and calls an access method's `Close` to close the opened device).
12. After the device manager has received the I/O messages from the reply mechanism (Step 3), and closed all the devices that it manages, it may optionally deregister the pool of I/O messages with the recovery agent via `DD_Support.Deregister_IO_message`.

### IX-1.3.1 I/O Recovery Agent

A *recovery agent* is provided on each node by the BiiN™ Operating System. This agent detects I/O processor failures and maintains a table of existing I/O messages. Device managers keep this list current by calling `DD_Support.Register_IO_message` each time they create an I/O message, and by calling `DD_Support.Deregister_IO_message` before they deallocate an I/O message.

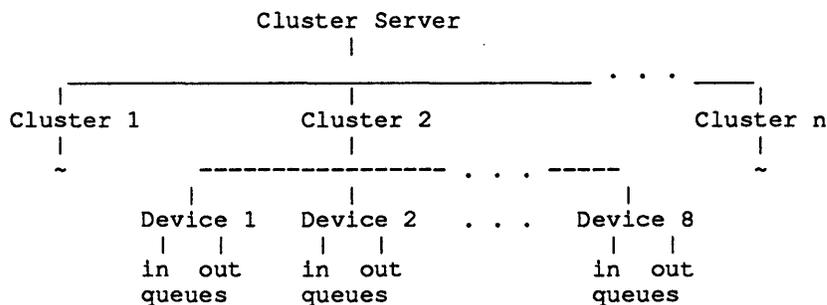
## IX-1.4 Data Transfer Via the Shared Queues Mechanism

The shared queues I/O mechanism is designed to handle low-speed, character-oriented communications for such devices as terminals and printers. This design minimizes context switches and interrupts while maintaining satisfactory response time.

The shared queues mechanism is comprised of a cluster servers which services one or more clusters which contain up to eight pairs of input and output queues (circular buffers). This mechanism employs an input and output queue for each device. These queues are grouped into clusters. A *cluster* is a group of queues that are serviced together. A cluster represents a group of devices, typically those serviced by the same channel processor (CP) task. See Figure IX-1-3.

## IX-1.5 Clusters and Cluster Servers

Clusters are configurable objects (CO) and are typically created and attached to devices during system initialization. A cluster may contain shared queues for up to eight devices. Cluster servers may service any number of clusters.



**Figure IX-1-3. Cluster Server, Clusters and shared queues**

---

The devices of each cluster must be of the same device class.

### IX-1.5.1 Administrative Interface

The shared queues I/O mechanism is a data transfer mechanism. Each device class that uses this mechanism must also specify an administrative interface. An administrative interface contains operations which initialize queues, set device parameters, etc.

When the I/O messages mechanism is used as an administrative interface, for example, the device class specification defines device-specific command codes and reply records and is used to initialize the clusters.

### IX-1.5.2 Device Driver Example

Figures IX-1-4 and IX-1-5 show how shared queues work with CPs and their relationship with an administrative interface.

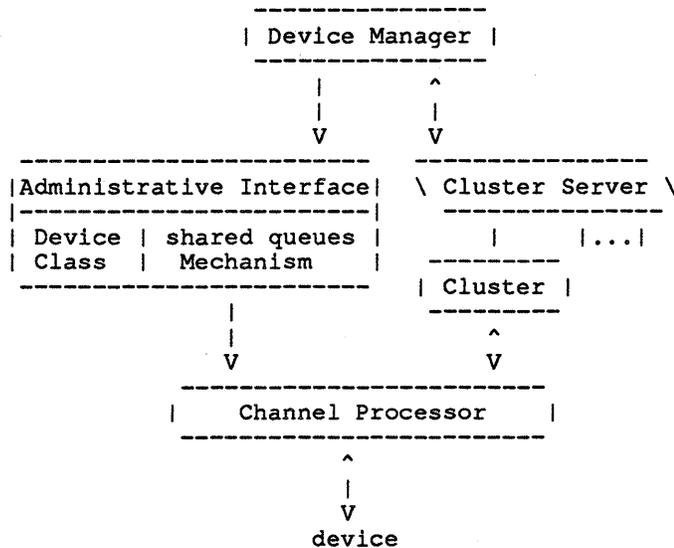


Figure IX-1-4. Device Driver with the Shared Queues Mechanism

### IX-1.5.3 I/O Shared Queues Data Transfer Mechanism

An input and an output queue are used to support data transfer between a device manager and a low-speed device via a CP/device driver. Each queue has a read pointer and a write pointer which indicate where the next character will be read or written, flags to indicate queues needing service and semaphores to block writers when queues are full. The data transfer process consists of four distinct activities:

- **Data Transfer From the Device Manager to the Output Queue**

The device manager writes data to the output queue.

- **Data Transfer From the Output Queue to the Device**

The CP/device driver polls its devices' output queues, and transfers any characters to those devices.

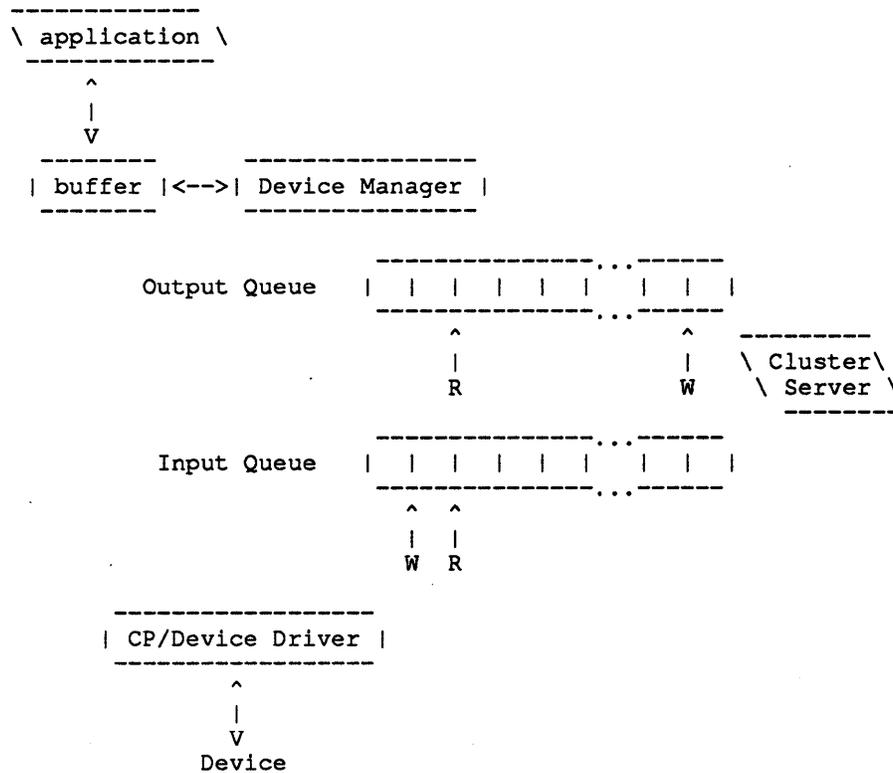
- **Data Transfer From the Device to the Input Queue**

The device interrupts the CP/device driver when it has characters to be returned to the device manager. The CP/device driver transfers the data to the input queue.

- **Data Transfer From the Input Queue to the Device Manager**

The cluster server polls its clusters and calls an input handler for any input queue containing characters.

These activities are described in more detail following Figure IX-1-5.



**Figure IX-1-5. I/O shared queues Data Transfer Mechanism**

1. A device manager transfers characters from an application's buffer to the output shared queue associated with the device.
2. When each write completes, the `cluster_object.new_output_flags` flag corresponding to the output queue associated with the device is set to show that this output queue is active (contains characters to be transferred to the device).
3. If the output queue fills before the device manager completes a write, `cluster_object.new_input_flags` is still set to active, and the writer blocks on `device_state_rep.block_user`. The device manager sets the boolean `device_state_rep.writer_blocked` to true.
4. The cluster server periodically checks the state of the output queue, and unblocks the writer when the contents of the output queue reach a low enough number of characters that more characters can be accepted.
5. When the number of characters remaining in the output queue is less than a `low_water_mark` (`device_state_rep.low_water_mark`), the cluster server unblocks the writer (calls `Semaphore_Mgt.V`), sets `device_state_rep.block_user` to false and calls `device_state_rep.input_handler`. This optimization technique prevents excessive blocking and context switching.

`device_state_rep.output_write_ptr` and `device_state_rep.output_read_ptr` are pointers for the output queue that indicate where to write and where to begin reading the next character. The device manager writes

characters beginning at the location indicated by the write pointer, and increments the pointer by the number of characters written. Likewise, the device manager reads characters beginning at the location indicated by the read pointer and increments the pointer by the number of characters read.

The queue is empty when the read pointer is equal to the write pointer. The queue is full when the read pointer is one more than the write pointer mod the queue size.

#### Data Transfer From the Output Queue to the Device

1. A CP/device driver periodically reads `cluster_object.new_output_flags` to determine if any of its device's output queues needs to be serviced.
2. For each active device, it sets the device's output flag in `cluster_object.new_output_flags` to false and sends a character to the device starting an interrupt-driven transfer loop.

The interrupt-driven loop is initiated by the CP/device driver when it polls the output queue and finds the new output flag set. The interrupt routine sets the new output flag to false and sends a character from the output queue to the device. (The flag must be reset before the character is sent.) When the device interrupts the CP/device driver to acknowledge receipt of the character, the loop checks the output queue again for another character to be sent. This loop continues until there are no more characters in the output queue.

#### NOTE

Occasionally, an output queue is marked active for which the interrupt-driven output transfer loop is in progress. The CP can detect this situation because it maintains an internal flag for each device that indicates whether or not a send is in progress. If a send is in progress, the CP marks the queue as inactive and moves on to the next active output queue.

#### Data Transfer From the Device to the Input Queue

1. The device sends an interrupt to the CP/device driver when it has a character to send. The CP/device driver calls an interrupt handler which places the character in the input queue, and sets the new input flag to true. (The character must be sent before the flag is reset.)
2. If the CP/device driver is unable to put a character in an input queue because the queue is full, it discards the character and sets the queue's overflow boolean, `input_lost`.

The use of the pointers in the input queue is similar to the use with the output queues except that the CP/device driver writes the characters using the write pointer and the device manager reads the character using the read pointer. A CP/device driver updates the read pointer of the output queue when removing characters. A CP/device driver reads the characters at the read pointer and increments the read pointer.

#### Data Transfer From the Input Queue to the Device Manager

1. The cluster server periodically checks the new input flags. If an input flag is set, the cluster server calls the input handler for the device (`device_state_rep.input_handler`).

## IX-1.6 Summary

- A device object is a typed object that represents a device.
- A device manager is a type manager that controls access to a device.
- An opened device object is a typed object that represents an input/output connection between a device manager and a device.
- A device class is a specification that defines the device-specific details necessary to access a member of a class of devices using an I/O mechanism.
- An access method is a collection of procedures that provide a device-independent interface to perform I/O.
- The I/O messages data transfer mechanism supports high-speed, block-oriented data transfer.
- The shared queues data transfer mechanism supports low-speed, character-oriented data transfer.
- An I/O message is an object consisting of four parts: common part, device driver part, device manager part and buffer description part.
- A recovery agent detects I/O processor failures and maintains a table of existing I/O messages.



# Part X

## Appendixes

---

The appendixes are:

- |                     |  |
|---------------------|--|
| <b>Ada Examples</b> | Contains complete listings of all examples used in this guide. |
| <b>Glossary</b>     | Defines terms used in this guide.                              |



# ADA EXAMPLES **A**

## Contents

Support Services .....	X-A-4
Example_Messages Package Specification .....	X-A-5
Long_Integer_Ex Package Specification .....	X-A-7
Long_Integer_Ex Package Body .....	X-A-8
Make_menu_group_DDef_ex Procedure .....	X-A-12
Manage_application_environment_ex Procedure .....	X-A-20
String_list_ex Procedure .....	X-A-23
Directory Services .....	X-A-23
Create_directory_cmd_ex Procedure .....	X-A-24
Create_name_space_cmd_ex Procedure .....	X-A-26
List_current_directory_cmd_ex Procedure .....	X-A-31
Make_object_public_ex Procedure .....	X-A-33
Show_current_directory_cmd_ex Procedure .....	X-A-35
I/O Services .....	X-A-36
DBMS_Support_Ex Package Specification .....	X-A-37
DBMS_Support_Ex Package Body .....	X-A-38
Employee_Filing_Ex Package Specification .....	X-A-42
Employee_Filing_Ex Package Body .....	X-A-46
Hello_ada_ex Procedure .....	X-A-54
Hello_OS_ex Procedure .....	X-A-55
Join_File_Ex Package Specification .....	X-A-56
Join_File_Ex Package Body .....	X-A-57
Record_Locking_Ex Package Specification .....	X-A-61
Record_Locking_Ex Package Body .....	X-A-62
Output_bytes_ex Procedure .....	X-A-64
Output_records_ex Procedure .....	X-A-65
Print_cmd_ex Procedure .....	X-A-67
Print_Cmd_Messages Package .....	X-A-70
Record_AM_Ex Package Specification .....	X-A-71
Record_AM_Ex Package Body .....	X-A-75
Simple_editor_cmd_ex Procedure .....	X-A-84
Simple_Editor_Ex Package Specification .....	X-A-85
Simple_Editor_Ex Package Body .....	X-A-89
Human Interface Services .....	X-A-104
Inventory_main Procedure .....	X-A-105
Inventory_Files Package Specification .....	X-A-108
Inventory_Files Package Body .....	X-A-115
Inventory_Forms Package Specification .....	X-A-121
Inventory_Forms Package Body .....	X-A-126
Inventory_Menus Package Specification .....	X-A-137
Inventory_Menus Package Body .....	X-A-140

**PRELIMINARY**

Inventory_Reports Package Specification .....	X-A-144
Inventory_Reports Package Body .....	X-A-146
Inventory_Windows Package Specification .....	X-A-152
Inventory_Windows Package Body .....	X-A-154
Inventory_Messages Package Specification .....	X-A-156
Program Services .....	X-A-156
At_cmd_ex Procedure .....	X-A-157
At_support_ex Package Specification .....	X-A-160
At_support_ex Package Body .....	X-A-162
Compiler_ex Package Specification .....	X-A-168
Compiler_ex Package Body .....	X-A-169
Conversion_support_ex Package Specification .....	X-A-172
Memory_ex Procedure .....	X-A-176
Process_globals_support_ex Package Specification .....	X-A-177
Process_globals_support_ex Package Body .....	X-A-182
Stream_file_ex Procedure .....	X-A-103
Symbol_table_ex Package Specification .....	X-A-191
Symbol_table_ex Package Body .....	X-A-193
Word_processor_ex Package Specification .....	X-A-197
Word_processor_ex Package Body .....	X-A-198
View_device_main Procedure .....	X-A-203
VD_Defs Package Specification .....	X-A-206
VD_Commands Package Specification .....	X-A-208
VD_Commands Package Body .....	X-A-209
VD_Devices Package Specification .....	X-A-213
VD_Devices Package Body .....	X-A-215
Type Manager Services .....	X-A-218
Acct_main_ex Procedure .....	X-A-219
Acct_visual Package Specification .....	X-A-236
Acct_visual Package Body .....	X-A-238
Account_Manager Command File .....	X-A-244
Account_Types_ex Package Specification .....	X-A-250
Account_Mgt_ex Package Specification .....	X-A-251
Account_Mgt_ex (Active Only) Package Body .....	X-A-256
Account_Mgt_ex (Stored, Non-transaction-oriented) Package Body .....	X-A-261
Account_Mgt_ex (Stored, Transaction-oriented) Package Body .....	X-A-267
Stored_Account_TDO_Init_ex Procedure .....	X-A-276
Account_Type_Name_ex Package Specification .....	X-A-279
Account_Type_Name_ex Package Body .....	X-A-280
Type_Name_Attr_ex Package Specification .....	X-A-281
Type_Name_Attr_ex Package Body .....	X-A-282
Type_Name_Attribute_Init_ex Procedure .....	X-A-283
Refuse_Reset_Active_Version_ex Package Specification .....	X-A-284
Refuse_Reset_Active_Version_ex Package Body .....	X-A-285
Account_Mgt_ex (Distributed) Package Body .....	X-A-286
Distr_Acct_Call_Stub_ex Package Specification .....	X-A-298
Distr_Acct_Call_Stub_ex Package Body .....	X-A-300
Distr_Acct_Server_Stub_ex Package Specification .....	X-A-304
Distr_Acct_Server_Stub_ex Package Body .....	X-A-306
Distr_Acct_Init Procedure .....	X-A-308
Distr_Acct_Home_Job_ex Procedure .....	X-A-312
Makefile .....	X-A-313

-----

Named_copy_ex Procedure .....	X-A-315
Older_than_ex Function .....	X-A-317

## X-A.1 Introduction

This appendix contains full listings of all the examples in the *BiiN™/OS Guide* grouped by service area.

All examples were compiled using Version V1.00.02 of the BiiN™ Ada compiler, and all compiled successfully (except where noted). Most examples are not yet tested, however.

## X-A.2 Support Services

## X-A.2.1 Example\_Messages Package Specification

```
1  with Incident_Defs,
2      System,
3      System_Defs;
4
5  package Example_Messages is
6  --
7  -- Function:
8  --   Define messages used by example programs.
9  --
10 --   A single message file is used. All messages
11 --   defined use a module ID of 0.
12
13 msg_file_pathname: constant System_Defs.text_AD :=
14   new System_Defs.text'(
15     30,30,"/examples/msg/example_messages");
16   -- AD to pathname of message file, bound to
17   -- "msg_obj", following.
18   --
19   -- *This will go away when "pragma bind" changes.*
20
21 msg_obj: constant System.untyped_word :=
22   System.null_word;
23   pragma bind(msg_obj,
24     "example_messages.msg_file_pathname");
25   -- Message object for incident codes in
26   -- example programs, bound to above "message_file_pathname".
27   --
28   -- *When the resident compiler and linker are*
29   -- *ready, this pragma will become:*
30   -- |   pragma bind(msg_obj,
31   -- |     "/examples/msg/example_messages");
32
33
34 not_directory_code:
35   constant Incident_Defs.incident_code :=
36     (0, 1, Incident_Defs.error, msg_obj);
37   --
38   --*M* store :module=0 :number=1 \
39   --*M*       :msg_name=not_directory_code \
40   --*M*       :short = \
41   --*M*       "$pl<pathname> is not a directory."
42
43 not_exist_or_no_access_code:
44   constant Incident_Defs.incident_code :=
45     (0, 2, Incident_Defs.error, msg_obj);
46   --
47   --*M* store :module=0 :number=2 \
48   --*M*       :msg_name=not_exist_or_no_access_code \
49   --*M*       :short = \
50   --*M*       "$pl<pathname> does not exist or does\
51   --*M* not allow you access."
52
53 no_access_code:
54   constant Incident_Defs.incident_code :=
55     (0, 3, Incident_Defs.error, msg_obj);
56   --
57   --*M* store :module=0 :number=3 \
58   --*M*       :msg_name=no_access_code \
59   --*M*       :short = \
60   --*M*       "$pl<pathname> does not allow\
61   --*M* you access."
62
63 overwrite_query_code:
64   constant Incident_Defs.incident_code :=
65     (0, 4, Incident_Defs.information, msg_obj);
66   --
67   --*M* store :module=0 :number=4 \
68   --*M*       :msg_name=overwrite_query_code \
69   --*M*       :short = \
70   --*M*       "$pl<pathname> exists. Overwrite it?"
71 not_overwritten_code:
72   constant Incident_Defs.incident_code :=
73     (0, 5, Incident_Defs.error, msg_obj);
74   --
```

## PRELIMINARY

```
75  --*M* store :module=0 :number=5 \  
76  --*M*      :msg_name=not_overwritten_code \  
77  --*M*      :short = \  
78  --*M*      "$pl<pathname> not overwritten."  
79  
80  create_name_space_aborted_code:  
81      constant Incident_Defs.incident_code :=  
82      (0, 6, Incident_Defs.information, msg_obj);  
83  --  
84  --*M* store :module=0 :number=6 \  
85  --*M*      :msg_name= \  
86  --*M*      create_name_space_aborted_code \  
87  --*M*      :short = "Operation aborted.\  
88  --*M*      No name space was created."  
89  
90  name_space_created_code:  
91      constant Incident_Defs.incident_code :=  
92      (0, 7, Incident_Defs.information, msg_obj);  
93  --  
94  --*M* store :module=0 :number=7 \  
95  --*M*      :msg_name=name_space_created_code \  
96  --*M*      :short = \  
97  --*M*      "Name space $pl<pathname> created."  
98  
99  end Example_Messages;
```

## X-A.2.2 Long\_Integer\_Ex Package Specification

```
1 with Long_Integer_Defs;
2
3 package Long_Integer_Ex is
4   --
5   -- Function:
6   -- Provide examples of using long integers.
7   -- See the package body for detailed comments.
8
9
10  function Long_integer_value(
11    image: string)
12    return Long_Integer_Defs.long_integer;
13
14
15  function Get_long_integer
16    return Long_Integer_Defs.long_integer;
17
18
19  function Multiply_divide(
20    a: integer;
21    b: integer;
22    c: integer)
23    return integer;
24
25
26  procedure Use_it;
27
28
29    pragma external;
30
31  end Long_Integer_Ex;
```

## X-A.2.3 Long\_Integer\_Ex Package Body

```

1  with Byte_Stream_AM,
2      Device_Defs,
3      Long_Integer_Defs,
4      Process_Mgt,
5      Process_Mgt_Types,
6      System,
7      System_Exceptions;
8
9  package body Long_Integer_Ex is
10     --
11     -- Function:
12     --   Provide examples of using long integers.
13     --
14     -- History:
15     --   12-02-87 Martin L. Buchanan Initial version.
16
17
18     function Long_integer_value(
19         image: string)
20         return Long_Integer_Defs.long_integer
21     --
22     -- Function:
23     --   Converts a string image to a long integer.
24     --
25     --   The image must have the following syntax:
26     --   |
27     --   | image ::= {space} [sign] digit { [_] digit }
28     --   |                               {space}
29     --   | space ::= ' '
30     --   | sign ::= +/-
31     --   | digit ::= 0|1|2|3|4|5|6|7|8|9
32     --
33     --   After leading and trailing spaces are stripped
34     --   off, the remaining part of the image cannot
35     --   be longer than 31 characters.
36     --
37     -- Notes:
38     --   Unlike "Long_Integer_Defs.Long_integer_value",
39     --   this function handles strings of varying length
40     --   and strings that contain trailing spaces.
41     --
42     -- Exceptions:
43     --   System_Exceptions.bad_parameter -
44     --   "image" has incorrect syntax, contains a
45     --   number longer than 31 characters, or contains
46     --   a number that cannot be represented as a long
47     --   integer.
48     is
49     li_string: Long_Integer_Defs.string_integer;
50     -- Fixed-length string required by
51     -- "Long_Integer_Defs.long_integer_value"
52     -- when converting to a long integer.
53     i: integer;
54     -- Will be index of right-most non-space character
55     -- in "image".
56     j: integer;
57     -- Will be index of left-most non-space character
58     -- in "image".
59     k: integer;
60     -- Will be index of left-most character in
61     -- "li_string" that is copied from "image(j..i)".
62     li: Long_Integer_Defs.long_integer;
63     -- The resulting long integer to return.
64     begin
65     -- Make "i" the index of the right-most
66     -- non-space character in "image":
67     --
68     i := image'last;
69     loop
70     if i < image'first then
71
72     -- "image" contains all spaces, or is a
73     -- null string:
74     --

```

```

75     RAISE System_Exceptions.bad_parameter;
76
77     else
78         EXIT when image(i) /= ' ';
79         i := i - 1;
80     end if;
81 end loop;
82
83 -- Make "j" the index of the left-most
84 -- non-space character in "image". No check
85 -- is needed for "image" being null or all
86 -- spaces, as those conditions are checked
87 -- above.
88 --
89 j := image'first;
90 loop
91     exit when image(j) /= ' ';
92     j := j + 1;
93 end loop;
94
95 if (i - j + 1) > li_string'length then
96
97     -- The number is longer than 31 characters
98     -- after stripping off spaces:
99     --
100    RAISE System_Exceptions.bad_parameter;
101
102 else
103
104     -- "k" is the index within "li_string" of the
105     -- leftmost character copied from "image". "k" is
106     -- computed to satisfy the following predicate:
107     -- | i - j = li_string'last - k
108     -- This predicate simply specifies that the number
109     -- of source characters copied equals the number
110     -- of destination characters.
111     --
112     k := li_string'last + j - i;
113
114     -- Copy the significant characters from "image" to
115     -- be right-justified within "li_string":
116     --
117     li_string(k .. li_string'last) :=
118         image(j .. i);
119
120     -- Fill any remaining left-hand characters in
121     -- "li_string" with spaces:
122     --
123     for m in li_string'first .. k-1 loop
124         li_string(m) := ' ';
125     end loop;
126
127     -- Compute and return the long integer value:
128     --
129     Long_Integer_Defs.Long_integer_value(
130         image => li_string,
131         number => li);    -- out.
132     RETURN li;
133
134 end if;
135 end Long_integer_value;
136
137
138 function Get_long_integer
139     return Long_Integer_Defs.long_integer
140 --
141 -- Function:
142 --     Gets a long integer on a single line
143 --     from the calling process's standard input.
144 --
145 -- Notes:
146 --     See "Long_integer_value" in this package
147 --     for a description of the required long
148 --     integer syntax and of what happens if
149 --     the syntax is violated.
150 --
151 --     There is no check for a line that's too long.

```

## PRELIMINARY

```

152 is
153   LINE_SIZE: constant integer := 80;
154   -- A line read from the standard input must
155   -- be <= 80 characters.
156   line: string(1 .. LINE_SIZE);
157   -- Line buffer.
158   length: integer;
159   -- Number of characters actually read.
160 begin
161   -- Read the line:
162   --
163   length := integer(Byte_Stream_AM.Ops.Read(
164     Device_Defs.opened_device(
165       Process_Mgt.Get_process_globals_entry(
166         Process_Mgt_Types.standard_input)),
167     line'address,
168     System.ordinal(LINE_SIZE)));
169
170   -- Strip any linefeed at the end:
171   --
172   if line(length) = ASCII.LF then
173     length := length - 1;
174   end if;
175
176   -- Convert to a long integer and return:
177   --
178   return Long_integer_value(line(1..length));
179 end Get_long_integer;
180
181 function Multiply_divide(
182   a: integer;
183   b: integer;
184   c: integer)
185   return integer
186   -- (a * b) / c
187   --
188   -- Function:
189   -- Computes and returns the product of two
190   -- integers divided by a third integer, using
191   -- a long integer for the intermediate result
192   -- to avoid overflow.
193   --
194   -- This function is useful for scaling and
195   -- unit conversions, to avoid overflow within
196   -- the calculation when the result after the
197   -- division step can still be represented as
198   -- an integer.
199   --
200   --
201   -- Exceptions:
202   -- System_Exceptions.bad_parameter -
203   -- Overflow or division by zero.
204 is
205   -- Convert all parameters to long integers:
206   --
207   a_long: Long_Integer_Defs.long_integer :=
208     Long_Integer_Defs.Convert_to_long_integer(a);
209   b_long: Long_Integer_Defs.long_integer :=
210     Long_Integer_Defs.Convert_to_long_integer(b);
211   c_long: Long_Integer_Defs.long_integer :=
212     Long_Integer_Defs.Convert_to_long_integer(c);
213
214   -- Import long integer operators:
215   --
216   use Long_Integer_Defs;
217
218 begin
219   return Convert_to_integer( (a_long * b_long) / c_long );
220 end Multiply_divide;
221
222 procedure Use_it
223   --
224   -- Function:
225   -- Show some computations with long integers.
226   --
227   --
228   -- Notes:

```

```

229  -- This procedure is not yet testable as it
230  -- is not a command and its variables are not
231  -- yet displayed.
232  is
233  -- Import long integer operators and the
234  -- "long_integer" type:
235  --
236  use Long_Integer_Defs;
237
238  -- Some variables to play with:
239  --
240  a: long_integer;
241  b: long_integer;
242  i: integer;
243
244  -- Declaring a negative long integer constant,
245  -- the easy way and the hard way:
246  --
247  negative_twenty: constant long_integer :=
248  - long_integer'(0, 20);
249
250  another_negative_twenty: constant long_integer :=
251  (16#ffff_ffff#, 16#ffff_ffec#);
252  -- Use the hard way when you want a declaration
253  -- elaborated at compile-time instead of
254  -- at run-time.
255  begin
256  -- Add one to a long integer:
257  --
258  a := a + Long_Integer_Defs.one;
259
260  -- Add a positive integer "i" to a long integer:
261  --
262  b := b + long_integer'(0, System.ordinal(i));
263  end Use_it;
264
265
266  end Long_Integer_Ex;

```

## X-A.2.4 Make\_menu\_group\_DDef\_ex Procedure

```

1  with Data_Definition_Mgt,
2      Directory_Mgt,
3      Passive_Store_Mgt,
4      System,
5      System_Defs,
6      Text_Mgt;
7
8  procedure Make_menu_group_DDef_ex
9  --
10 -- Function:
11 --   Creates and stores a menu group DDef,
12 --   containing two menus and five menu items:
13 --
14 --|           -----           -----
15 --|           Menu 1             Menu 2
16 --|           -----           -----
17 --|           Menu Item 1       Menu Item 1
18 --|           Menu Item 2       Menu Item 2
19 --|           -----           -----
20 --|
21 --
22
23 is
24
25   use Data_Definition_Mgt;    -- to import enumeration types
26
27   ddf:          Data_Definition_Mgt.DDef_AD;
28   untyped_ddf: System.untyped_word;
29   FOR untyped_ddf USE AT ddf'address;
30
31   group_node:   Data_Definition_Mgt.node_reference;
32   menu_list_node: Data_Definition_Mgt.node_reference;
33   menu_node:    Data_Definition_Mgt.node_reference;
34   item_list_node: Data_Definition_Mgt.node_reference;
35   item_node:    Data_Definition_Mgt.node_reference;
36   dont_care_node: Data_Definition_Mgt.node_reference;
37   name:         System_Defs.text(100);
38   prop_value:   Data_Definition_Mgt.property_value(100);
39
40 begin
41
42   ddf := Data_Definition_Mgt.Create_DDef;
43
44   -- Create menu group
45
46   Text_Mgt.Set(name, "group_node");
47   group_node := Data_Definition_Mgt.Create_node(
48     DDef => ddf,
49     node_name => name,
50     root => private_root_node);
51
52   prop_value.simple_pv := (pv_boolean, true);
53   Data_Definition_Mgt.Add_property_value(
54     node_ref => group_node,
55     property => pi_derive_all,
56     value => prop_value);
57
58   prop_value.simple_pv := (pv_boolean, true);
59   Data_Definition_Mgt.Add_property_value(
60     node_ref => group_node,
61     property => pi_import,
62     value => prop_value);
63
64   prop_value.simple_pv := (pv_type => pv_string);
65   Text_Mgt.Set(prop_value.text_value, "menu_group_t");
66   Data_Definition_Mgt.Add_property_value(
67     node_ref => group_node,
68     property => pi_DDef_name,
69     value => prop_value);
70
71   Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
72   Data_Definition_Mgt.Add_property_value(
73     node_ref => group_node,
74     property => pi_DDef_name,

```

```

75     value    => prop_value);
76
77 Text_Mgt.Set(name, "menu_list");
78 menu_list_node := Data_Definition_Mgt.Create_field(
79     record_node => group_node,
80     node_name   => name,
81     property    => pi_has_value,
82     value       => (pv_node_reference, menu_node));
83
84
85 -- Create the first menu ("Menu 1"):
86 --
87 Text_Mgt.Set(name, "menu_node");
88 menu_node := Data_Definition_Mgt.Create_node(
89     DDef      => ddf,
90     node_name => name,
91     root      => private_root_node);
92
93 prop_value.simple_pv := (pv_boolean, true);
94 Data_Definition_Mgt.Add_property_value(
95     node_ref => menu_node,
96     property => pi_derive_all,
97     value    => prop_value);
98
99 prop_value.simple_pv := (pv_boolean, true);
100 Data_Definition_Mgt.Add_property_value(
101     node_ref => menu_node,
102     property => pi_import,
103     value    => prop_value);
104
105 prop_value.simple_pv := (pv_type => pv_string);
106 Text_Mgt.Set(prop_value.text_value, "menu_t");
107 Data_Definition_Mgt.Add_property_value(
108     node_ref => menu_node,
109     property => pi_DDef_name,
110     value    => prop_value);
111
112 Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
113 Data_Definition_Mgt.Add_property_value(
114     node_ref => menu_node,
115     property => pi_DDef_name,
116     value    => prop_value);
117
118 Text_Mgt.Set(name, "menu_id");
119 dont_care_node := Data_Definition_Mgt.Create_field(
120     record_node => menu_node,
121     node_name   => name,
122     property    => pi_has_value,
123     value       => (pv_int4, 1));
124
125 prop_value.simple_pv := (pv_type => pv_string);
126 Text_Mgt.Set(prop_value.text_value, "Menu 1");
127 Text_Mgt.Set(name, "menu_title");
128 dont_care_node := Data_Definition_Mgt.Create_field(
129     record_node => menu_node,
130     node_name   => name,
131     property    => pi_has_value,
132     value       => prop_value.simple_pv);
133
134 Text_Mgt.Set(name, "item_list");
135 item_list_node := Data_Definition_Mgt.Create_field(
136     record_node => menu_node,
137     node_name   => name);
138
139
140 -- Now create the menu items for menu 1:
141 --
142
143 -- Create menu item 1:
144 --
145 Text_Mgt.Set(name, "item_node");
146 item_node := Data_Definition_Mgt.Create_node(
147     DDef      => ddf,
148     node_name => name,
149     root      => private_root_node);
150
151 prop_value.simple_pv := (pv_boolean, true);

```

PRELIMINARY

```

152 Data_Definition_Mgt.Add_property_value(
153     node_ref => item_node,
154     property => pi_derive_all,
155     value    => prop_value);
156
157 prop_value.simple_pv := (pv_boolean, true);
158 Data_Definition_Mgt.Add_property_value(
159     node_ref => item_node,
160     property => pi_import,
161     value    => prop_value);
162
163 prop_value.simple_pv := (pv_type => pv_string);
164 Text_Mgt.Set(prop_value.text_value, "menu_item_t");
165 Data_Definition_Mgt.Add_property_value(
166     node_ref => item_node,
167     property => pi_DDef_name,
168     value    => prop_value);
169
170 Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
171 Data_Definition_Mgt.Add_property_value(
172     node_ref => item_node,
173     property => pi_DDef_name,
174     value    => prop_value);
175
176 Text_Mgt.Set(name, "item_id");
177 dont_care_node := Data_Definition_Mgt.Create_field(
178     record_node => item_node,
179     node_name   => name,
180     property    => pi_has_value,
181     value       => (pv_int4, 1));
182
183 Text_Mgt.Set(name, "checked");
184 dont_care_node := Data_Definition_Mgt.Create_field(
185     record_node => item_node,
186     node_name   => name,
187     property    => pi_has_value,
188     value       => (pv_boolean, true));
189
190 Text_Mgt.Set(name, "enabled");
191 dont_care_node := Data_Definition_Mgt.Create_field(
192     record_node => item_node,
193     node_name   => name,
194     property    => pi_has_value,
195     value       => (pv_boolean, true));
196
197 prop_value.simple_pv := (pv_type => pv_string);
198 Text_Mgt.Set(prop_value.text_value, "Menu Item 1");
199 Text_Mgt.Set(name, "text");
200 dont_care_node := Data_Definition_Mgt.Create_field(
201     record_node => item_node,
202     node_name   => name,
203     property    => pi_has_value,
204     value       => prop_value.simple_pv);
205
206 -- Add menu item 1 to menu 1:
207 --
208 prop_value.simple_pv := (pv_node_reference, item_node);
209 Data_Definition_Mgt.Add_property_value(
210     node_ref => item_list_node,
211     property => pi_has_value,
212     value    => prop_value);
213
214
215 -- Create menu item 2 for menu 1:
216 --
217 Text_Mgt.Set(name, "item_node");
218 item_node := Data_Definition_Mgt.Create_node(
219     DDef    => ddf,
220     node_name => name,
221     root    => private_root_node);
222
223 prop_value.simple_pv := (pv_boolean, true);
224 Data_Definition_Mgt.Add_property_value(
225     node_ref => item_node,
226     property => pi_derive_all,
227     value    => prop_value);
228

```

```

229 prop_value.simple_pv := (pv_boolean, true);
230 Data_Definition_Mgt.Add_property_value(
231     node_ref => item_node,
232     property => pi_import,
233     value    => prop_value);
234
235 prop_value.simple_pv := (pv_type => pv_string);
236 Text_Mgt.Set(prop_value.text_value, "menu_item_t");
237 Data_Definition_Mgt.Add_property_value(
238     node_ref => item_node,
239     property => pi_DDef_name,
240     value    => prop_value);
241
242 Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
243 Data_Definition_Mgt.Add_property_value(
244     node_ref => item_node,
245     property => pi_DDef_name,
246     value    => prop_value);
247
248 Text_Mgt.Set(name, "item_id");
249 dont_care_node := Data_Definition_Mgt.Create_field(
250     record_node => item_node,
251     node_name   => name,
252     property    => pi_has_value,
253     value       => (pv_int4, 2));
254
255 Text_Mgt.Set(name, "checked");
256 dont_care_node := Data_Definition_Mgt.Create_field(
257     record_node => item_node,
258     node_name   => name,
259     property    => pi_has_value,
260     value       => (pv_boolean, false));
261
262 Text_Mgt.Set(name, "enabled");
263 dont_care_node := Data_Definition_Mgt.Create_field(
264     record_node => item_node,
265     node_name   => name,
266     property    => pi_has_value,
267     value       => (pv_boolean, false));
268
269 prop_value.simple_pv := (pv_type => pv_string);
270 Text_Mgt.Set(prop_value.text_value, "Menu Item 2");
271 Text_Mgt.Set(name, "text");
272 dont_care_node := Data_Definition_Mgt.Create_field(
273     record_node => item_node,
274     node_name   => name,
275     property    => pi_has_value,
276     value       => prop_value.simple_pv);
277
278
279 -- Add menu item 2 to menu 1:
280 --
281 prop_value.simple_pv := (pv_node_reference, item_node);
282 Data_Definition_Mgt.Add_property_value(
283     node_ref => item_list_node,
284     property => pi_has_value,
285     value    => prop_value);
286
287
288 -- Add menu 1 to the menu group:
289 --
290 prop_value.simple_pv := (pv_node_reference, menu_node);
291 Data_Definition_Mgt.Add_property_value(
292     node_ref => menu_list_node,
293     property => pi_has_value,
294     value    => prop_value);
295
296 -- Create menu 2:
297 --
298 Text_Mgt.Set(name, "menu_node");
299 menu_node := Data_Definition_Mgt.Create_node(
300     DDef    => ddf,
301     node_name => name,
302     root     => private_root_node);
303
304 prop_value.simple_pv := (pv_boolean, true);
305 Data_Definition_Mgt.Add_property_value(

```

## PRELIMINARY

```

306     node_ref => menu_node,
307     property => pi_derive_all,
308     value    => prop_value);
309
310     prop_value.simple_pv := (pv_boolean, true);
311     Data_Definition_Mgt.Add_property_value(
312     node_ref => menu_node,
313     property => pi_import,
314     value    => prop_value);
315
316     prop_value.simple_pv := (pv_type => pv_string);
317     Text_Mgt.Set(prop_value.text_value, "menu_t");
318     Data_Definition_Mgt.Add_property_value(
319     node_ref => menu_node,
320     property => pi_DDef_name,
321     value    => prop_value);
322
323     Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
324     Data_Definition_Mgt.Add_property_value(
325     node_ref => menu_node,
326     property => pi_DDef_name,
327     value    => prop_value);
328
329     Text_Mgt.Set(name, "menu_id");
330     dont_care_node := Data_Definition_Mgt.Create_field(
331     record_node => menu_node,
332     node_name   => name,
333     property    => pi_has_value,
334     value       => (pv_int4, 2));
335
336     prop_value.simple_pv := (pv_type => pv_string);
337     Text_Mgt.Set(prop_value.text_value, "Menu 2");
338     Text_Mgt.Set(name, "menu_title");
339     dont_care_node := Data_Definition_Mgt.Create_field(
340     record_node => menu_node,
341     node_name   => name,
342     property    => pi_has_value,
343     value       => prop_value.simple_pv);
344
345     Text_Mgt.Set(name, "item_list");
346     item_list_node := Data_Definition_Mgt.Create_field(
347     record_node => menu_node,
348     node_name   => name);
349
350     -- Now create menu items for menu 2:
351
352     -- Create menu item 1 for menu 2:
353     --
354     Text_Mgt.Set(name, "item_node");
355     item_node := Data_Definition_Mgt.Create_node(
356     DDef      => ddf,
357     node_name => name,
358     root      => private_root_node);
359
360     prop_value.simple_pv := (pv_boolean, true);
361     Data_Definition_Mgt.Add_property_value(
362     node_ref => item_node,
363     property => pi_derive_all,
364     value    => prop_value);
365
366     prop_value.simple_pv := (pv_boolean, true);
367     Data_Definition_Mgt.Add_property_value(
368     node_ref => item_node,
369     property => pi_import,
370     value    => prop_value);
371
372     prop_value.simple_pv := (pv_type => pv_string);
373     Text_Mgt.Set(prop_value.text_value, "menu_item_t");
374     Data_Definition_Mgt.Add_property_value(
375     node_ref => item_node,
376     property => pi_DDef_name,
377     value    => prop_value);
378
379     Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
380     Data_Definition_Mgt.Add_property_value(
381     node_ref => item_node,
382     property => pi_DDef_name,

```

PRELIMINARY

```

383     value    => prop_value);
384
385 Text_Mgt.Set(name, "item_id");
386 dont_care_node := Data_Definition_Mgt.Create_field(
387     record_node => item_node,
388     node_name   => name,
389     property    => pi_has_value,
390     value       => (pv_int4, 1));
391
392 Text_Mgt.Set(name, "checked");
393 dont_care_node := Data_Definition_Mgt.Create_field(
394     record_node => item_node,
395     node_name   => name,
396     property    => pi_has_value,
397     value       => (pv_boolean, true));
398
399 Text_Mgt.Set(name, "enabled");
400 dont_care_node := Data_Definition_Mgt.Create_field(
401     record_node => item_node,
402     node_name   => name,
403     property    => pi_has_value,
404     value       => (pv_boolean, true));
405
406 prop_value.simple_pv := (pv_type => pv_string);
407 Text_Mgt.Set(prop_value.text_value, "Menu Item 1");
408 Text_Mgt.Set(name, "text");
409 dont_care_node := Data_Definition_Mgt.Create_field(
410     record_node => item_node,
411     node_name   => name,
412     property    => pi_has_value,
413     value       => prop_value.simple_pv);
414
415
416 -- Add menu item 1 to menu 2:
417 --
418 prop_value.simple_pv := (pv_node_reference, item_node);
419 Data_Definition_Mgt.Add_property_value(
420     node_ref => item_list_node,
421     property => pi_has_value,
422     value    => prop_value);
423
424
425 -- Create menu item 2 for menu 2:
426 --
427 Text_Mgt.Set(name, "item_node");
428 item_node := Data_Definition_Mgt.Create_node(
429     DDef    => ddf,
430     node_name => name,
431     root     => private_root_node);
432
433 prop_value.simple_pv := (pv_boolean, true);
434 Data_Definition_Mgt.Add_property_value(
435     node_ref => item_node,
436     property => pi_derive_all,
437     value    => prop_value);
438
439 prop_value.simple_pv := (pv_boolean, true);
440 Data_Definition_Mgt.Add_property_value(
441     node_ref => item_node,
442     property => pi_import,
443     value    => prop_value);
444
445 prop_value.simple_pv := (pv_type => pv_string);
446 Text_Mgt.Set(prop_value.text_value, "menu_item_t");
447 Data_Definition_Mgt.Add_property_value(
448     node_ref => item_node,
449     property => pi_DDef_name,
450     value    => prop_value);
451
452 Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
453 Data_Definition_Mgt.Add_property_value(
454     node_ref => item_node,
455     property => pi_DDef_name,
456     value    => prop_value);
457
458 Text_Mgt.Set(name, "item_id");
459 dont_care_node := Data_Definition_Mgt.Create_field(

```

## PRELIMINARY

```

460     record_node => item_node,
461     node_name   => name,
462     property    => pi_has_value,
463     value       => (pv_int4, 2));
464
465 Text_Mgt.Set(name, "checked");
466 dont_care_node := Data_Definition_Mgt.Create_field(
467     record_node => item_node,
468     node_name   => name,
469     property    => pi_has_value,
470     value       => (pv_boolean, true));
471
472 Text_Mgt.Set(name, "enabled");
473 dont_care_node := Data_Definition_Mgt.Create_field(
474     record_node => item_node,
475     node_name   => name,
476     property    => pi_has_value,
477     value       => (pv_boolean, true));
478
479 prop_value.simple_pv := (pv_type => pv_string);
480 Text_Mgt.Set(prop_value.text_value, "Menu Item 2");
481 Text_Mgt.Set(name, "text");
482 dont_care_node := Data_Definition_Mgt.Create_field(
483     record_node => item_node,
484     node_name   => name,
485     property    => pi_has_value,
486     value       => prop_value.simple_pv);
487
488
489 -- Add menu item 2 to menu 2:
490 --
491 prop_value.simple_pv := (pv_node_reference, item_node);
492 Data_Definition_Mgt.Add_property_value(
493     node_ref => item_list_node,
494     property => pi_has_value,
495     value    => prop_value);
496
497
498 -- Create menu item 3 for menu 2:
499 --
500 Text_Mgt.Set(name, "item_node");
501 item_node := Data_Definition_Mgt.Create_node(
502     DDef    => ddf,
503     node_name => name,
504     root    => private_root_node);
505
506 prop_value.simple_pv := (pv_boolean, true);
507 Data_Definition_Mgt.Add_property_value(
508     node_ref => item_node,
509     property => pi_derive_all,
510     value    => prop_value);
511
512 prop_value.simple_pv := (pv_boolean, true);
513 Data_Definition_Mgt.Add_property_value(
514     node_ref => item_node,
515     property => pi_import,
516     value    => prop_value);
517
518 prop_value.simple_pv := (pv_type => pv_string);
519 Text_Mgt.Set(prop_value.text_value, "menu_item_t");
520 Data_Definition_Mgt.Add_property_value(
521     node_ref => item_node,
522     property => pi_DDef_name,
523     value    => prop_value);
524
525 Text_Mgt.Set(prop_value.text_value, "/ddefs/menu_DDef");
526 Data_Definition_Mgt.Add_property_value(
527     node_ref => item_node,
528     property => pi_DDef_name,
529     value    => prop_value);
530
531 Text_Mgt.Set(name, "item_id");
532 dont_care_node := Data_Definition_Mgt.Create_field(
533     record_node => item_node,
534     node_name   => name,
535     property    => pi_has_value,
536     value       => (pv_int4, 3));

```

**PRELIMINARY**

```

537
538 Text_Mgt.Set(name, "checked");
539 dont_care_node := Data_Definition_Mgt.Create_field(
540   record_node => item_node,
541   node_name   => name,
542   property    => pi_has_value,
543   value       => (pv_boolean, true));
544
545 Text_Mgt.Set(name, "enabled");
546 dont_care_node := Data_Definition_Mgt.Create_field(
547   record_node => item_node,
548   node_name   => name,
549   property    => pi_has_value,
550   value       => (pv_boolean, false));
551
552 prop_value.simple_pv := (pv_type => pv_string);
553 Text_Mgt.Set(prop_value.text_value, "Menu Item 3");
554 Text_Mgt.Set(name, "text");
555 dont_care_node := Data_Definition_Mgt.Create_field(
556   record_node => item_node,
557   node_name   => name,
558   property    => pi_has_value,
559   value       => prop_value.simple_pv);
560
561
562 -- Add menu item 3 to menu 2:
563 --
564 prop_value.simple_pv := (pv_node_reference, item_node);
565 Data_Definition_Mgt.Add_property_value(
566   node_ref => item_list_node,
567   property => pi_has_value,
568   value    => prop_value);
569
570
571 -- Add menu 2 to the menu group:
572 --
573 prop_value.simple_pv := (pv_node_reference, menu_node);
574 Data_Definition_Mgt.Add_property_value(
575   node_ref => menu_list_node,
576   property => pi_has_value,
577   value    => prop_value);
578
579
580 -- Complete and close the menu group:
581 --
582 prop_value.simple_pv := (pv_type => pv_string);
583 Text_Mgt.Set(prop_value.text_value, "/tdo/menu_group_tdo");
584 Data_Definition_Mgt.Add_property_value(
585   node_ref => group_node,
586   property => pi_kind,
587   value    => prop_value);
588
589
590 -- Close the definition (DDef):
591 --
592 Data_Definition_Mgt.Close(
593   DDef => ddf);
594
595
596 -- Store the DDef:
597 --
598 Text_Mgt.Set(name, "///pathname/menu_group_DDef");
599 Directory_Mgt.Store(name, untyped_ddf);
600
601 -- Request update of stored DDef:
602 --
603 Passive_Store_Mgt.Request_update(
604   obj => untyped_ddf);
605
606 end Make_menu_group_DDef_ex;

```

## X-A.2.5 Manage\_application\_environment\_ex Procedure

```

1  with CL_Defs,
2      Environment_Mgt,
3      String_List_Mgt,
4      System,
5      System_Defs,
6      Text_IO,
7      Text_Mgt;
8
9  procedure Manage_Application_Environment_Ex
10 --
11 -- Function:
12 --   Example program showing use of environment
13 --   variables.
14 --
15 -- History:
16 --   06-26-87, William Anton Rohm:  Written.
17 --   12-02-87, WAR:                 Revised.
18 --
19 is
20
21 package Int_IO is new Text_IO.Integer_IO(integer);
22
23 -- Variables:
24 --
25 variable_name: System_Defs.text(
26     CL_Defs.max_name_sz);
27 variable_type: CL_Defs.var_type;
28 variable_mode: CL_Defs.var_mode;
29
30 variable_name_list: System_Defs.string_list(1000);
31
32 integer_value: integer;
33 ASCII_value:   System_Defs.text(1000);
34 answer:       character;
35
36 use CL_Defs; -- to import "=" for CL_Defs.var_mode
37 use System;  -- to import "+" for System.ordinal
38
39 begin
40
41 -- Create a new local integer variable named
42 -- "new_integer":
43 --
44 Text_Mgt.Set(
45     dest => variable_name,
46     source => "new_integer");
47
48 Environment_Mgt.Set_integer(
49     var_name => variable_name,
50     value    => 0,
51     mode     => CL_Defs.read_write,
52     global   => false);
53
54
55 -- Display all local variable names:
56 --
57 Environment_Mgt.Get_all_names(
58     group_name => System_Defs.null_text,
59     list       => variable_name_list,
60     global     => false);
61
62 Text_IO.Put_line("List of local variables:");
63
64 for i in 1 .. variable_name_list.count loop
65
66     String_List_Mgt.Get_element(
67         from    => variable_name_list,
68         el_pos  => i,
69         element => variable_name);
70
71     Text_IO.Put_line(variable_name.value);
72
73 end loop;
74

```

## PRELIMINARY

```

75
76 -- Read type and mode of given variable:
77 --   If integer and read-write, add one to variable;
78 --   otherwise, read and display ASCII
79 --   representation of value:
80 --
81 Text_IO.Put("Enter a variable name:");
82
83 Text_IO.Get(variable_name.value);
84
85 variable_type := Environment_Mgt.Get_var_type(
86   var_name => variable_name);
87
88 variable_mode := Environment_Mgt.Get_var_mode(
89   var_name => variable_name);
90
91 if variable_type = CL_Defs.integer_type then
92
93   integer_value := Environment_Mgt.Get_integer(
94     var_name => variable_name);
95
96   Text_IO.Put("Original value of ");
97   Text_IO.Put(variable_name.value);
98   Text_IO.Put(" integer variable is:");
99   Int_IO.Put(integer_value);
100  Text_IO.Put_line("");
101
102  if variable_mode = CL_Defs.read_write then
103    integer_value := integer_value + 1;
104
105    Environment_Mgt.Set_integer(
106      var_name => variable_name,
107      value    => integer_value);
108
109    Text_IO.Put("New value of ");
110    Text_IO.Put(variable_name.value);
111    Text_IO.Put(" integer variable is:");
112    Int_IO.Put(integer_value);
113    Text_IO.Put_line("");
114
115  else
116    Text_IO.Put("Mode of ");
117    Text_IO.Put(variable_name.value);
118    Text_IO.Put_line(" integer variable is 'read-only'.");
119
120  end if;          -- if "read_write"
121
122 else              -- not "integer_type"
123
124   Environment_Mgt.Convert_and_get(
125     var_name => variable_name,
126     value    => ASCII_value);
127
128   Text_IO.Put("Value of ");
129   Text_IO.Put(variable_name.value);
130   Text_IO.Put(" variable is:");
131   Text_IO.Put_line(ASCII_value.value);
132
133   if variable_mode = CL_Defs.read_write then
134
135     Text_IO.Put("Change value?");
136
137     Text_IO.Get(answer);
138
139     if answer = 'y' or
140        answer = 'Y' then
141
142       Text_IO.Put("Enter new value:");
143       Text_IO.Get(ASCII_value.value);
144
145       Environment_Mgt.Convert_and_set(
146         var_name => variable_name,
147         value    => ASCII_value,
148         var_type => variable_type);
149
150     end if;          -- if answer = 'y'
151

```

## PRELIMINARY

```
152     else
153         Text_IO.Put("Mode of ");
154         Text_IO.Put(variable_name.value);
155         Text_IO.Put_line(" variable is 'read-only'.");
156
157     end if;                                -- if mode = read_write
158
159 end if;                                    -- if "integer_type"
160
161
162 -- Remove new variable:
163 --
164 Text_Mgt.Set(
165     dest => variable_name,
166     source => "new_integer");
167
168 Environment_Mgt.Remove(
169     var_name => variable_name,
170     quiet => true,
171     global => false);
172
173 end Manage_Application_Environment_Ex;
174
```

### X-A.2.6 String\_list\_ex Procedure

```
1  with String_List_Mgt,  
2      System_Defs;  
3  
4  procedure String_list_ex  
5      --  
6      -- Function:  
7      --   Create string list with following entries:  
8      --   1. "ux_group"  
9      --   2. "world"  
10 is  
11   string_list: System_Defs.string_list(255);  
12   begin  
13  
14     -- 1) "ux_group"  
15     String_List_Mgt.Set(string_list,  
16       System_Defs.text'(8, 8, "ux_group"));  
17  
18     -- 2) "world"  
19     String_List_Mgt.Append(string_list,  
20       System_Defs.text'(5, 5, "world"));  
21  
22   end String_list_ex;
```

### X-A.3 Directory Services

## X-A.3.1 Create\_directory\_cmd\_ex Procedure

```

1  with Command_Handler,
2     Device_Defs,
3     Directory_Mgt,
4     System_Defs;
5
6  procedure Create_directory_cmd_ex
7     --
8     -- Function:
9     --   Creates a named subdirectory in the
10    --   caller's current directory.
11    --
12    -- Command Definition:
13    --   The command has the form:
14    --   create.directory :name=<string>
15    --
16    -- Create the command definition by entering:
17    --| clex -> manage.program :tagged_source=create.dir.sb
18    --
19    --*D* set.program create.directory
20    --*D*
21    --*D* manage.commands
22    --*D*
23    --*D*   create.invocation_command
24    --*D*   define.argument_name :type = string
25    --*D*   set.lexical_class symbolic_name
26    --*D*   set.maximum_length 252
27    --*D*   set.mandatory
28    --*D*   set.description :text = "
29    --*D*     -- Name of directory to be created.
30    --*D*     "
31    --*D*   end
32    --*D*   set.description :text = "
33    --*D*     -- Creates a directory in the
34    --*D*     -- current directory.
35    --*D*     "
36    --*D*   end
37    --*D*   exit          -- manage.commands
38    --*D*   exit          -- manage.program
39    --
40  is
41
42     opened_command: Device_Defs.opened_device;
43     -- Opened invocation command input device.
44
45     dir_name: System_Defs.text(252);
46     -- Name of the directory to be created.
47
48     dir_AD: Directory_Mgt.directory_AD;
49     -- Newly created directory's AD; returned
50     -- but not used by "create.directory".
51  begin
52
53     -- Open invocation command input device:
54     --
55     opened_command := Command_Handler.
56       Open_invocation_command_processing;
57
58     -- Get ":name" parameter:
59     --
60     Command_Handler.Get_string(
61       cmd_odo => opened_command,
62       arg_number => 1,
63       arg_value => dir_name);
64
65     -- Close invocation command input device:
66     --
67     Command_Handler.Close(opened_command);
68
69
70     -- Create new named directory:
71     --
72     dir_AD := Directory_Mgt.Create_directory(
73       name => dir_name);
74

```

## PRELIMINARY

```
75 end Create_directory_cmd_ex;  
76
```

## X-A.3.2 Create\_name\_space\_cmd\_ex Procedure

```

1  with CL_Defs,
2      Command_Handler,
3      Device_Defs,
4      Directory_Mgt,
5      Environment_Mgt,
6      Example_Messages, -- Example package.
7      Incident_Defs,
8      Message_Services,
9      Name_Space_Mgt,
10     Passive_Store_Mgt,
11     String_List_Mgt,
12     System,
13     System_Defs,
14     System_Exceptions,
15     Transaction_Mgt;
16
17 procedure Create_name_space_cmd_ex
18     --
19     -- Function:
20     -- Defines a command to create a name space,
21     -- along with the code that executes the command.
22     --
23     -- Command Definition:
24     -- The command has the form:
25     --
26     --     create.name_space
27     --         :name=<string>
28     --         :directory_list=<string_list>
29     --         [ :force=<boolean>:=false]
30     --
31     -- Pathnames in the directory list must name
32     -- directories.
33     --
34     -- If "force" is omitted or false then the "name"
35     -- pathname must not be in use. If "force" is
36     -- true and the "name" pathname is in use, then
37     -- the environment variable "user.confirm" is
38     -- consulted. If "user.confirm" is true (or does
39     -- not exist), then the user is queried before
40     -- deleting the existing use of the pathname.
41     --
42     --*C* set.message_file \
43     --*C*     :file = /examples/msg/example_messages
44     --*C*
45     --*C* create.command \
46     --*C*     :cmd_def = create.n_s.inv_cmd \
47     --*C*     :cmd_name = create.name_space
48     --*C*
49     --*C* define.argument name \
50     --*C*     :type = string
51     --*C*     set.lexical_class symbolic_name
52     --*C*     set.maximum_length 252
53     --*C*     set.mandatory
54     --*C* end
55     --*C*
56     --*C* define.argument directory_list \
57     --*C*     :type = string_list
58     --*C*     set.lexical_class symbolic_name
59     --*C*     set.maximum_length 508
60     --*C* end
61     --*C*
62     --*C* define.argument force \
63     --*C*     :type = boolean
64     --*C*     set.value_default false
65     --*C* end
66     --*C* end
67     --*C*
68     --*C* run "store.command_definitions \\  

69     --*C*     :exec_unit = create.n_s \\  

70     --*C*     :invocation_cmd = create.n_s.inv_cmd"
71     --*C*
72     --*C* run "store.default_message_file \\  

73     --*C*     create.n_s \\  

74     --*C*     /examples/msg/example_messages

```

## PRELIMINARY

```

75  --
76
77
78  is
79
80  opened_cmd: Device_Defs.opened_device;
81  -- Opened command input device.
82
83  name: System_Defs.text(Incident_Defs.txt_length);
84  -- Pathname of new name space.
85
86  directory_list: System_Defs.string_list(508);
87  -- String list containing pathnames of the
88  -- directories in the new name space.
89
90  force: boolean;
91  -- Whether the new name space's pathname should
92  -- overwrite an existing entry.
93
94  i: natural;
95  -- Index into "directory_list".
96
97  directory_path: System_Defs.text(Incident_Defs.txt_length);
98  -- Text containing each successive pathname from
99  -- "directory_list".
100
101  valid: boolean := true;
102  -- True if "directory_list" is valid. Assigned
103  -- false if it is invalid.
104
105  name_space: Name_Space_Mgt.name_space_AD;
106  -- The new name space.
107
108  name_space_untyped: System.untyped_word;
109  FOR name_space_untyped USE AT name_space'address;
110  -- The new name space as an untyped word.
111
112  user_confirm_name: constant System_Defs.text(
113    12) := (12, 12, "user.confirm");
114  -- Text record of an environment variable's name.
115
116  user_confirm_var_exists: boolean;
117  -- Whether a user variable named
118  -- "user.confirm" exists.
119
120  user_confirm_var: boolean;
121  -- Value of "user.confirm" variable, if it exists
122  -- ("user_confirm_var_exists" is true).
123
124  overwrite: boolean;
125  -- Whether the created name space can overwrite an
126  -- existing entry with the same pathname.
127
128
129  begin
130
131  -- Get command arguments:
132  --
133  opened_cmd := Command_Handler.
134    Open_invocation_command_processing;
135
136  -- Get first argument (name of new name space):
137  --
138  Command_Handler.Get_string(opened_cmd, 1,
139    arg_value => name);
140
141  -- Get second argument (list of directories):
142  --
143  Command_Handler.Get_string_list(opened_cmd, 2,
144    arg_value => directory_list);
145
146  -- Get third argument (force overwrite):
147  --
148  force := Command_Handler.Get_boolean(opened_cmd, 3);
149
150
151  Command_Handler.Close(opened_cmd);

```

PRELIMINARY

```

152
153
154 -- Check each pathname in the directory list:
155 --
156 i := 1;
157
158 loop
159
160   String_List_Mgt.Get_element_by_index(
161     from      => directory_list,
162     list_index => i,
163     element   => directory_path);
164
165   -- Exit after last string:
166   --
167   EXIT when i = 0;
168
169   -- Check if pathname exists, and is a directory:
170   --
171   begin
172     if not Directory_Mgt.Is_directory(
173       Directory_Mgt.Retrieve(directory_path)) then
174
175       valid := false;
176
177       Message_Services.Write_msg(
178         Example_Messages.not_directory_code,
179         Incident_Defs.message_parameter(
180           typ => Incident_Defs.txt,
181           len => directory_path.length)'(
182           typ      => Incident_Defs.txt,
183           len      => directory_path.length,
184           txt_val => directory_path));
185     end if;
186
187   exception
188     when Directory_Mgt.no_access =>
189
190     valid := false;
191
192     Message_Services.Write_msg(
193       Example_Messages.no_access_code,
194       Incident_Defs.message_parameter(
195         typ => Incident_Defs.txt,
196         len => directory_path.length)'(
197         typ      => Incident_Defs.txt,
198         len      => directory_path.length,
199         txt_val => directory_path));
200
201   end;
202
203 end loop;
204
205 if not valid then
206   Message_Services.Write_msg(
207     Example_Messages.
208     create_name_space_aborted_code);
209 else
210   name_space := Name_Space_Mgt.Create_name_space(
211     directory_list);
212
213   -- Store new name space as a directory entry:
214   --
215   loop
216     begin
217
218       -- Start a transaction to store new name space:
219       --
220       Transaction_Mgt.Start_transaction;
221       Directory_Mgt.Store(name, name_space_untyped);
222
223       -- Exit if no exception raised:
224       --
225       EXIT;
226
227     exception
228

```

## PRELIMINARY

```

229 when System_Exceptions.
230     transaction_timestamp_conflict =>
231
232     Transaction_Mgt.Abort_transaction;
233
234
235 when Directory_Mgt.entry_exists =>
236
237     Transaction_Mgt.Abort_transaction;
238
239     if force then
240
241         begin
242             user_confirm_var := Environment_Mgt.Get_boolean(
243                 user_confirm_name);
244
245             user_confirm_var_exists := true;
246
247         exception
248             when CL_Defs.non_existent |
249                  CL_Defs.invalid_type |
250                  CL_Defs.no_value =>
251                 user_confirm_var_exists := false;
252         end;
253
254         if user_confirm_var_exists and then
255             (not user_confirm_var) then
256             -- No confirmation necessary:
257             --
258             overwrite := true;
259
260         else
261             -- Confirm overwrite:
262             --
263             overwrite :=
264                 Message_Services.Acknowledge_msg(
265                     Example_Messages.
266                         overwrite_query_code,
267                     Incident_Defs.
268                         message_parameter(
269                             typ => Incident_Defs.txt,
270                             len => name.max_length)'(
271                         typ
272                             =>
273                             Incident_Defs.txt,
274                             len
275                                 =>
276                                 name.max_length,
277                                 txt_val => name));
278         end if;
279
280     else
281         -- "force" false:
282         --
283         overwrite := false;
284     end if;
285
286     if overwrite then
287         begin
288             Directory_Mgt.Delete(name);
289
290         exception
291             when Directory_Mgt.no_access =>
292                 null;
293         end;
294
295     else
296         Message_Services.Write_msg(
297             Example_Messages.not_overwritten_code,
298             Incident_Defs.message_parameter(
299                 typ => Incident_Defs.txt,
300                 len => name.max_length)'(
301                 typ
302                     => Incident_Defs.txt,
303                 len
304                     => name.max_length,
305                 txt_val => name));
306
307         Message_Services.Write_msg(
308             Example_Messages.
309                 create_name_space_aborted_code);

```

PRELIMINARY

```

306         end if;
307
308     when Directory_Mgt.no_access =>
309
310         Transaction_Mgt.Abort_transaction;
311
312         Message_Services.Write_msg(
313             Example_Messages.no_access_code,
314             Incident_Defs.message_parameter(
315                 typ => Incident_Defs.txt,
316                 len => name.max_length)'(
317                 typ      => Incident_Defs.txt,
318                 len      => name.max_length,
319                 txt_val => name));
320
321         Message_Services.Write_msg(
322             Example_Messages.
323             create_name_space_aborted_code);
324
325     when others =>
326
327         Transaction_Mgt.Abort_transaction;
328
329         RAISE;
330
331     end;
332
333 end loop;
334
335 -- Update passive version:
336 --
337 Passive_Store_Mgt.Request_update(
338     name_space_untyped);
339
340 -- Commit the "store new name space" transaction:
341 --
342 Transaction_Mgt.Commit_transaction;
343
344 -- Inform user of succesful creation of new name
345 -- space:
346 --
347 Message_Services.Write_msg(
348     Example_Messages.name_space_created_code,
349     Incident_Defs.message_parameter(
350         typ => Incident_Defs.txt,
351         len => name.length)'(
352         typ      => Incident_Defs.txt,
353         len      => name.length,
354         txt_val => name));
355 end if;      -- if all directories in path are
356              -- valid
357
358 end Create_name_space_cmd_ex;
359

```

## X-A.3.3 List\_current\_directory\_cmd\_ex Procedure

```

1  with Byte_Stream_AM,
2      Command_Handler,
3      Device_Defs,
4      Directory_Mgt,
5      Process_Mgt,
6      Process_Mgt_Types,
7      System,
8      System_Defs,
9      Unchecked_Conversion;
10
11 procedure List_current_directory_cmd_ex
12 --
13 -- Function:
14 --   Lists names of entries in user's current
15 --   directory.
16 --
17 --   Each entry name is written to the user's
18 --   standard output, on a separate line.
19 --
20 -- Command Definition:
21 --   The command has the form:
22 --   list.current_directory [:pattern=<string>]
23 --
24 --*D*   manage.commands
25 --*D*   create.invocation_command
26 --*D*
27 --*D*   define.argument pattern \
28 --*D*       :type = string
29 --*D*       set.lexical_class symbolic_name
30 --*D*       set.maximum_length 252
31 --*D*       set.value_default ""
32 --*D*   end
33 --*D* end
34 --*D*
35 --
36 --
37 is
38
39 -- Generic function:
40 --
41 function Directory_AD_from_untyped_word is
42     new Unchecked_conversion(
43         source => System.untyped_word,
44         target => Directory_Mgt.directory_AD);
45
46
47 -- Variables:
48 --
49 odo: Device_Defs.opened_device :=
50     Command_Handler.
51     Open_invocation_command_processing;
52 -- Opened invocation command input device.
53
54 pattern: System_Defs.text(252) := (252, 252, (others => ' '));
55 -- Optional ":pattern" used to select entries
56 -- matching the pattern, such as "abc?" or
57 -- "m*device". Default is "!.*", meaning all
58 -- entries NOT beginning with a "." (period).
59
60 opened_dir: Device_Defs.opened_device;
61 -- Opened device for reading stream of names
62 -- from user's current directory.
63
64 standard_output: Device_Defs.opened_device :=
65     Device_Defs.opened_device(
66         Process_Mgt.Get_process_globals_entry(
67             Process_Mgt_Types.standard_output));
68 -- User's standard output.
69
70 name_buffer: array(1 .. 250) of character;
71 -- Each entry name is read into this buffer
72 -- and then written from it.
73
74 length: System.ordinal;

```

## PRELIMINARY

```
75     -- Length in bytes (characters) of last
76     -- entry name read.
77 use System;      -- for " 'size/8 " arithmetic
78
79 begin
80
81     -- Get ":pattern", if any:
82
83     Command_Handler.Get_string(
84         cmd_odo => odo,
85         arg_number => 1,
86         arg_value => pattern);
87
88     -- Close invocation command input device:
89     --
90     Command_Handler.Close(odo);
91
92     -- Open directory for reading, filtered by
93     -- ":pattern":
94     --
95     opened_dir := Directory_Mgt.Open_directory(
96         dir      => Directory_AD_from_untyped_word(
97             Process_Mgt.Get_process_globals_entry(
98                 Process_Mgt.Types.current_dir)),
99         pattern => pattern);
100
101
102     -- Get and write each entry name:
103     --
104     loop
105
106         length := Byte_Stream_AM.Ops.Read(
107             opened_dev => opened_dir,
108             buffer_VA  => name_buffer'address,
109             length     => name_buffer'size/8);
110
111         Byte_Stream_AM.Ops.Write(
112             opened_dev => standard_output,
113             buffer_VA  => name_buffer'address,
114             length     => length);
115
116     end loop;
117
118 exception
119
120     when Device_Defs.end_of_file =>
121
122         Byte_Stream_AM.Ops.Close(opened_dir);
123
124         RETURN;
125
126 end List_current_directory_cmd_ex;
127
```

## X-A.3.4 Make\_object\_public\_ex Procedure

```

1  with Authority_List_Mgt,
2      Directory_Mgt,
3      Identification_Mgt,
4      Passive_Store_Mgt,
5      System,
6      System_Defs,
7      Transaction_Mgt,
8      User_Mgt;
9
10 procedure Make_object_public_ex(
11     obj:      System.untyped_word;
12     -- Object to be made public.
13     aut_list_path: System_Defs.text)
14     -- Pathname under which to store the new
15     -- authority list.
16     --
17     -- Function:
18     --   Makes an object "public" by giving it an
19     --   authority list that grants all type rights
20     --   to the "world" ID.
21     --
22     -- Logic:
23     --   1. Get an AD to the world ID.
24     --   2. Define a protection set that grants all
25     --   type rights to the world ID.
26     --   3. Create an authority list with that
27     --   protection set.
28     --   4. Enclose steps (5) and (6) in a transaction.
29     --   5. Store the authority list under the pathname
30     --   given as the "aut_list_path" parameter.
31     --   6. Passivate the authority list, so that it
32     --   will endure in passive store along with
33     --   the object that it protects.
34     --   7. Assign the authority list as the object's
35     --   authority list.
36     --
37     -- Exceptions:
38     --   Authority_List_Mgt.set_authority_refused -
39     --   The object's master AD was stored with
40     --   no authority list protecting the object,
41     --   and an authority list cannot now be assigned.
42 is
43     -- Get the world ID AD
44     world_name: constant System_Defs.text(9) :=
45         (9, 9, "/id/world");
46     world_untyped: constant System.untyped_word :=
47         Directory_Mgt.Retrieve(world_name);
48     world_id: Identification_Mgt.ID_AD;
49     FOR world_id USE AT world_untyped'address;
50
51     -- Define the protection set
52     entries: constant User_Mgt.protection_set(1) := (
53         size => 1, length => 1,
54         entries => (1 => (rights => (true, true, true),
55             id => world_id)));
56
57     -- Create the authority list
58     aut_list: constant
59         Authority_List_Mgt.authority_list_AD :=
60         Authority_List_Mgt.Create_authority(entries);
61     aut_untyped: System.untyped_word;
62     FOR aut_untyped USE AT aut_list'address;
63
64 begin
65     Transaction_Mgt.Start_transaction;
66     begin
67         Directory_Mgt.Store(aut_list_path, aut_untyped);
68         Passive_Store_Mgt.Request_update(aut_untyped);
69         Transaction_Mgt.Commit_transaction;
70     exception
71         when others =>
72             Transaction_Mgt.Abort_transaction;
73             RAISE;
74

```

## PRELIMINARY

```
75     end;  
76     Authority_List_Mgt.Set_object_authority(  
77         obj, aut_list);  
78 end Make_object_public_ex;
```

## X-A.3.5 Show\_current\_directory\_cmd\_ex Procedure

```

1  with Byte_Stream_AM,
2      Device_Defs,
3      Directory_Mgt,
4      Process_Mgt,
5      Process_Mgt_Types,
6      System,
7      System_Defs,
8      Text_Mgt;
9
10 procedure Show_current_directory_cmd_ex
11 --
12 -- Function:
13 -- Gets and displays the pathname of the
14 -- current directory.
15 --
16 -- Command Definition:
17 -- The command has the form:
18 --   show.current_directory
19 --
20 --*C* create.command \
21 --*C*   :cmd_def = show.cur_dir.inv_cmd \
22 --*C*   :cmd_name = show.current_directory
23 --*C* end
24 --*C*
25 --*C* run "store.command_definitions \
26 --*C*   :exec_unit = show.cur_dir \
27 --*C*   :invocation_cmd = show.cur_dir.inv_cmd"
28 --
29 is
30
31 standard_output: Device_Defs.opened_device :=
32   Device_Defs.opened_device(
33     Process_Mgt.Get_process_globals_entry(
34       Process_Mgt_Types.standard_output));
35 -- User's standard output.
36
37 current_dir: Directory_Mgt.directory_AD :=
38   Directory_Mgt.directory_AD(
39     Process_Mgt.Get_process_globals_entry(
40       Process_Mgt_Types.current_dir));
41 -- Current directory's AD.
42
43 current_dir_untyped: System.untyped_word;
44 FOR current_dir_untyped USE AT
45   current_dir'address;
46 -- Current directory's AD as an untyped word.
47
48 dir_name: System_Defs.text(252);
49 -- Current directory's name.
50
51 begin
52
53 -- Get current directory's pathname:
54 --
55 Directory_Mgt.Get_name(
56   obj => current_dir_untyped,
57   name => dir_name);
58
59 -- Add a line-feed to pathname for displaying:
60 --
61 Text_Mgt.Append(
62   dest => dir_name,
63   source => Standard.ASCII.LF);
64
65 -- Display pathname:
66 --
67 Byte_Stream_AM.Ops.Write(
68   opened_dev => standard_output,
69   buffer_VA => dir_name.value'address,
70   length => System.ordinal(
71     dir_name.length));
72
73 end Show_current_directory_cmd_ex;
74

```

## X-A.4 I/O Services

## X-A.4.1 DBMS\_Support\_Ex Package Specification

```

1  with Device_Defs,
2      System,
3      System_Defs;
4
5  package DBMS_Support_EX is
6  --
7  -- Function:
8  -- Shows how to use the record processing and
9  -- DBMS support operations in applications.
10 --
11 -- History:
12 -- 08-15-87, Paul Schwabe: initial version.
13 -- 12-01-87, Paul Schwabe: reorganized.
14 --
15 pragma external;
16
17 procedure Selection(
18     opened_file: Device_Defs.opened_device;
19     read_procedure: System.subprogram_type);
20     -- An opened device, opened for input on an
21     -- employee file.
22 --
23 -- Function:
24 -- Do a Record_AM.Keyed_Ops.Set_key_range using
25 -- the Dept index. Do a
26 -- Record_Processing_Support.Set_oriented_read.
27 -- Returns a set of records for the range of
28 -- departments indicated.
29
30
31 procedure Projection(
32     opened_file: Device_Defs.opened_device;
33     projection_DDef_name: System_Defs.text);
34     -- An opened device, opened for input on an
35     -- employee file.
36 --
37 -- Function:
38 -- Grabs only certain fields for each record
39 -- that is read from the employee file. Set
40 -- the filter up using the following call:
41
42
43 procedure Sort_records(
44     inventory_file: Device_Defs.opened_device;
45     inventory_DDef_name: System_Defs.text);
46     -- An opened device, opened for input on an
47     -- inventory file. Uses
48 --
49 -- Function:
50 -- Sort_Merge_Interface.Sort to sort records
51 -- from an inventory file (writes to standard
52 -- out).
53
54
55 procedure Merge_and_sort_records(
56     inventory_file: Device_Defs.opened_device;
57     employee_file: Device_Defs.opened_device;
58     sort_DDef_name: System_Defs.text);
59     -- Two opened devices, opened for input on an
60     -- inventory file and employee file.
61 --
62 -- Function:
63 -- Uses Sort_Merge_Interface.Sort_merge to merge
64 -- and sort records from two (the inventory and
65 -- the employee) files (writes to standard out).
66
67
68
69 end DBMS_Support_EX;
70

```

## X-A.4.2 DBMS\_Support\_Ex Package Body

```

1  with Employee_Filing_Ex,
2      Data_Definition_Mgt,
3      Device_Defs,
4      Process_Globals_Support_Ex,
5      Record_AM,
6      Record_Processing_Support,
7      Sort_Merge_Interface,
8      Trusted_Record_Processing_Support,
9      System,
10     System_Defs,
11     Unchecked_conversion;
12
13 use System;
14
15 package body DBMS_Support_Ex is
16     --
17     -- Logic:
18     -- Shows how to do record processing
19     -- support operations.
20     --
21
22
23 procedure Selection(
24     opened_file: Device_Defs.opened_device;
25     read_procedure: System.subprogram_type)
26     -- An opened device, opened for input on an
27     -- employee file.
28     -- Logic:
29     -- Do a Record_AM.Keyed_Ops.Set_key_range using
30     -- the Dept index. Do a
31     -- Record_Processing_Support.Set_oriented_read.
32     -- Returns a set of records for the range of
33     -- departments indicated.
34 is
35     start_key_value: constant
36     Employee_Filing_Ex.dept_key_buffer := (
37         dept => 100);
38     -- Lowest dept for ascending key field.
39
40     start_key_descr: constant
41     Record_AM.key_value_descr := (
42         start_key_value'address,
43         start_key_value'size / 8);
44
45     stop_key_value: constant
46     Employee_Filing_Ex.dept_key_buffer := (
47         dept => 305);
48     -- Highest dept value
49     -- for ascending key field.
50
51     stop_key_descr: constant
52     Record_AM.key_value_descr := (
53         stop_key_value'address,
54         stop_key_value'size / 8);
55
56 begin
57     Trusted_Record_Processing_Support.Associate_read_procedure(
58         opened_dev => opened_file,
59         user_info => System.null_address,
60         read_procedure => read_procedure);
61
62
63     Record_AM.Keyed_Ops.Set_key_range(
64         opened_dev => opened_file,
65         index =>
66             Employee_Filing_Ex.dept_index_name,
67         select_range => (
68             start_comparison => Record_AM.inclusive,
69             start_value => start_key_descr,
70             stop_comparison => Record_AM.inclusive,
71             stop_value => stop_key_descr));
72
73     Record_Processing_Support.Set_oriented_read(
74         opened_dev => opened_file,

```

## PRELIMINARY

```

75     modifier      => Record_AM.next,
76     output_device => Process_Globals_Support_Ex.
77         Get_standard_output,
78         -- Normally defaulted.
79     alt_output    => System.null_word,
80     no_record_lock => false,
81     lock          => Record_AM.read_lock,
82     unlock        => Record_AM.no_unlock,
83     timeout       => Record_AM.wait_forever);
84     -- DO ANY NEEDED PROCESSING HERE.
85
86 exception
87     when Device_Defs.end_of_file =>
88         null;
89
90 end Selection;
91
92
93 procedure Projection(
94     opened_file: Device_Defs.opened_device;
95     projection_DDef_name: System_Defs.text)
96     -- An opened device, opened for input on an
97     -- employee file.
98     -- Logic:
99     --   Grabs only certain fields for each record
100    --   that is read from the employee file.
101    --
102 is
103     projection_DDef_ref:   Data_Definition_Mgt.
104                           node_reference;
105
106     buffer: string(1 .. integer(Employee_Filing_Ex.max_rec_size));
107     -- Buffer is large enough to hold any employee
108     -- record.
109
110     current_record_addr: constant
111         System.address := buffer'address;
112     current_record_VA: constant
113         Employee_Filing_Ex.employee_record_VA :=
114             Employee_Filing_Ex.
115                 Employee_record_VA_from_VA(
116                     current_record_addr);
117
118     bytes_read: System.ordinal;
119     -- Number of bytes in current record.
120
121 begin
122     --
123     -- Open projection data definition.
124     --
125
126     projection_DDef_ref :=
127         Record_AM.Ops.Get_DDef(
128             opened_dev => Record_AM.Open_by_name(
129                 name      => projection_DDef_name,
130                 input_output => Device_Defs.input,
131                 allow      => Device_Defs.readers,
132                 block      => true));
133
134     -- Filters out all fields except those specified
135     -- in the DDef.
136     Record_Processing_Support.
137         Associate_primary_data_projection(
138             opened_dev => opened_file,
139             record_ID_output => false,
140             primary_fields => projection_DDef_ref);
141
142
143     loop
144         -- Only reads the fields specified in
145         -- the DDef.
146         bytes_read := Record_AM.Ops.Read(
147             opened_dev => opened_file,
148             modifier   => Record_AM.next,
149             -- Normally defaulted.
150             buffer_VA  => current_record_addr,
151             length     => System.ordinal(

```

## PRELIMINARY

```

152         Employee_Filing_Ex.max_rec_size));
153
154     -- DO ANY NEEDED PROCESSING HERE.
155
156     end loop;
157 exception
158     when Device_Defs.end_of_file =>
159         null;
160
161 end Projection;
162
163
164
165 procedure Sort_records(
166     inventory_file: Device_Defs.opened_device;
167     inventory_DDef_name: System_Defs.text)
168     -- An opened device, opened for input on an
169     -- inventory file.
170     -- Logic:
171     -- Uses Sort_Merge_Interface.Sort to sort
172     -- records from an inventory file (writes to
173     -- standard out).
174 is
175     opened_inventory_ddef: Device_Defs.opened_device;
176     inventory_ddef_ref: Data_Definition_Mgt.
177         node_reference;
178 begin
179     --
180     -- Open inventory definition.
181     --
182     opened_inventory_DDef :=
183     Record_AM.Open_by_name(
184         name => inventory_DDef_name,
185         input_output => Device_Defs.input,
186         allow => Device_Defs.readers,
187         block => true);
188
189     inventory_DDef_ref :=
190     Record_AM.Ops.Get_DDef(
191         opened_dev => opened_inventory_DDef);
192
193     Sort_Merge_Interface.Sort(
194         input_device => inventory_file,
195         DDef => inventory_DDef_ref,
196         output_device => Process_Globals_Support_Ex.
197             Get_standard_output,
198         stable_sort => true,
199         tuning_opts => Sort_Merge_Interface.
200             no_tuning);
201     --
202     -- Close inventory file.
203     --
204     Record_AM.Ops.Close(
205         opened_dev => opened_inventory_DDef);
206
207 end Sort_records;
208
209
210 procedure Merge_and_sort_records(
211     inventory_file: Device_Defs.opened_device;
212     employee_file: Device_Defs.opened_device;
213     sort_DDef_name: System_Defs.text)
214     -- Two opened devices, opened for input on an
215     -- inventory file and employee file. Uses
216     -- Logic:
217     -- Sort_Merge_Interface.Sort_merge to merge
218     -- and sort records from two (the inventory
219     -- and the employee) files (writes to
220     -- standard out).
221 is
222     opened_sort_DDef: Device_Defs.opened_device;
223     sort_DDef_ref: Data_Definition_Mgt.
224         node_reference;
225     sort_input_array: Sort_Merge_Interface.
226         sort_merge_input_array(1 .. 2) :=
227         (1 => (input_device => inventory_file,
228             presorted => false,

```

## PRELIMINARY

```
229         sorted_by_index => false),
230         2 => (input_device => employee_file,
231             presorted    => false,
232             sorted_by_index => false));
233 begin
234     --
235     -- Open sort data definition.
236     --
237     opened_sort_DDef :=
238         Record_AM.Open_by_name(
239             name      => sort_DDef_name,
240             input_output => Device_Defs.input,
241             allow     => Device_Defs.readers,
242             block    " => true);
243
244     sort_DDef_ref :=
245         Record_AM.Ops.Get_DDef(
246             opened_dev => opened_sort_DDef);
247
248     -- Perform the sort-merge.
249     Sort_Merge_Interface.Sort_merge(
250         input_devices => sort_input_array,
251         DDef         => sort_DDef_ref,
252         output_device => Process_Globals_Support_EX.
253             Get_standard_output,
254         stable_sort  => true,
255         tuning_opts  => Sort_Merge_Interface.
256             no_tuning);
257
258     --
259     -- Close inventory file.
260     --
261     Record_AM.Ops.Close(
262         opened_dev => opened_sort_DDef);
263
264 end Merge_and_sort_records;
265
266
267 end DBMS_Support_EX;
```

## X-A.4.3 Employee\_Filing\_Ex Package Specification

```

1  with Data_Definition_Mgt,
2      File_Defs,
3      System,
4      System_Defs,
5      Unchecked_conversion;
6
7  use System;
8
9  package Employee_Filing_Ex is
10
11     -- Function:
12     --   Defines an employee file structure.
13     --
14     --
15     --   Contains declarations for employee records and
16     --   indexes.   Contains subprograms for creating
17     --   needed DDefs and for creating an employee file
18     --   with indexes.
19     --
20     --   The "employee_record" type defines the record
21     --   format.
22     --
23     --   An employee file has two indexes:
24     --
25     --   "Dept index" - A b-tree index sorted by salary
26     --   ascending department.  Allows duplicates.
27     --
28     --   "Dept-salary" index - A b-tree index
29     --   sorted by ascending department and descending
30     --   salary.  Allows duplicates.
31     --
32     pragma external;
33
34     --
35     -- CONSTANTS
36     --
37
38     max_text_length:    constant := 25;
39     --   The maximum length for a person's
40     --   name.
41     max_job_desc_length: constant := 200;
42     --   The maximum length of a job description
43     --   string.
44
45     --
46     -- FIELD SUBTYPES OR TYPES
47
48     subtype department_number is
49         System.ordinal range 0 .. 1000;
50     --   A work group within the company.
51
52     subtype person_name is
53         System_Defs.text(max_text_length);
54
55     --   Format is: last-name, first-name middle-name
56     --   [suffix ] This format is used so that records
57     --   can be ordered alphabetically on last name then
58     --   first name.
59
60     subtype job_description_length is
61         integer range 0 .. max_job_desc_length;
62     --   String length allowed for a job
63     --   description.
64
65     subtype monthly_salary is float;
66     --   The monthly salary for an employee.
67
68     --
69     -- RECORD DECLARATIONS
70
71     type employee_record(
72         length: job_description_length) is
73         record
74             dept:          department_number;

```

## PRELIMINARY

```

75     name:      person_name;
76     job_descr: string(1 .. length);
77     salary:    monthly_salary;
78     end record;
79
80 -- This specific representation assures the
81 -- record is correctly represented for the
82 -- DDef. The fields must be word aligned.
83
84 FOR employee_record USE
85     record
86         dept      at 0 range 0 .. 31;
87         name      at 4 range 0 .. 231;
88         salary    at 36 range 0 .. 63;
89     end record;
90
91
92 max_rec_size: constant System.ordinal := 241;
93 -- Maximum number of bytes in the employee record.
94 -- Used to determine the buffer size when
95 -- reading an employee record.
96
97 type employee_record_VA is access employee_record;
98 pragma access_kind(employee_record_VA, virtual);
99 -- Type contains virtual pointers to employee
100 -- records.
101
102 employee_DDef: Data_Definition_Mgt.node_reference;
103 -- Data definition for the employee record.
104
105 --
106 -- DECLARATIONS FOR INDEXES
107 -- A simple index declaration.
108 dept_index_DDef: Data_Definition_Mgt.
109                 node_reference;
110
111 dept_index_name: constant
112     File_Defs.index_name :=
113     (max_length => File_Defs.index_name_length,
114      length => 14,
115      value => "Dept_Index_DDef");
116
117 type dept_key_buffer is
118     record
119         dept:      department_number;
120     end record;
121
122 -- A composite index declaration.
123 dept_salary_index_DDef:
124     Data_Definition_Mgt.node_reference;
125
126 dept_salary_index_name: constant
127     File_Defs.index_name :=
128     (max_length => File_Defs.index_name_length,
129      length => 21,
130      value => "Dept_Salary_Index_DDef");
131
132 type dept_salary_key_buffer is
133     record
134         dept:      department_number;
135         salary:    monthly_salary;
136     end record;
137 -- This specific representation assures the
138 -- buffer is correctly represented for the
139 -- DDef. There is no padding between fields.
140 FOR dept_salary_key_buffer USE
141     record
142         dept      at 0 range 0 .. 31;
143         salary    at 4 range 0 .. 63;
144     end record;
145
146 --
147 -- CALLS
148 --
149
150
151 function Employee_record_VA_from_VA is new

```

## PRELIMINARY

```
152     Unchecked_conversion(  
153         source => System.address,  
154         target => employee_record_VA);  
155  
156  
157 function VA_from_employee_record_VA is new  
158     Unchecked_conversion(  
159         source => employee_record_VA,  
160         target => System.address);  
161  
162  
163  
164 procedure Create_employee_DDef;  
165     --  
166     -- Function:  
167     --   Creates DDefs for the employee record and all  
168     --   indexes.  
169     --  
170     --   The DDefs are in a single DDef object, which  
171     --   is passivated with the specified pathname.  
172     --  
173     --   "Create_employee_DDefs" assigns all the  
174     --   "ddef" variables in this package.  
175     --  
176     -- Notes:  
177     --   "Create_employee_DDefs" is normally called  
178     --   only once in the lifetime of a system.  
179     --  
180     --   The same DDefs can be used by multiple  
181     --   employee files.  
182  
183  
184 procedure Create_dept_DDef;  
185     --  
186     -- Function:  
187     --   Sets up an index key DDef for an employee  
188     --   file by deriving fields from an existing  
189     --   record DDef.  
190     --  
191     --   The index key DDef requires the properties  
192     --   indicated by the following pseudo-DDef  
193     --   language:  
194     --  
195     --   define "Dept"  
196     --       record Import from (  
197     --           "Employee_Data",  
198     --           "Employee_DDef"),  
199     --           Derive_all is false;  
200     --           Maps to "Dept",  
201     --           This simple index key is set up by mapping  
202     --           DDef nodes from "Employee_DDef" to a new  
203     --           record DDef called "Index_2_DDef"  
204     --           that consists of one field:  
205     --           * "Dept" in ascending order.  
206  
207 procedure Create_dept_salary_DDef;  
208     --  
209     -- Function:  
210     --   Sets up an index key DDef for an employee  
211     --   file by deriving fields from an existing  
212     --   record DDef.  
213     --  
214     --   The index key DDef requires the properties  
215     --   indicated by the following pseudo-DDef  
216     --   language:  
217     --  
218     --   define "Dept-Salary"  
219     --       record Import from (  
220     --           "Employee_Data",  
221     --           "Employee_DDef"),  
222     --           Derive_all is false;  
223     --           Maps to "Dept",  
224     --           Maps to "Salary",  
225     --           descending is true;  
226     --  
227     --   This composite index key is set up by mapping  
228     --   DDef nodes from "Employee_DDef" to a new
```

## PRELIMINARY

```
229      -- record DDef called "Dept-Salary"
230      -- that consists of two fields:
231      --   * "Dept" in ascending order.
232      --   * "Salary" in descending order.
233
234
235  procedure Create_file_and_indexes(
236      file_name:      System_Defs.text;
237      org_index_name: System_Defs.text);
238      -- New file's pathname.
239      --
240      -- Function:
241      --   Creates an employee file with all needed
242      --   indexes. The employee file is a clustered
243      --   organization.
244      --
245      --   The new file is initially empty.
246      --
247      --   "Create_employee DDefs" must have been called
248      --   *before* any call to "Create_employee_file".
249      --
250      -- Note:
251      --   The index is built after the file is created.
252      --
253      --   The file uses DDefs defined in the
254      --   Employee_Filing_Ex package.
255
256  end Employee_Filing_Ex;
257
258
```

## X-A.4.4 Employee\_Filing\_Ex Package Body

```

1  with Data_Definition_Mgt,
2      Directory_Mgt,
3      File_Admin,
4      File_Defs,
5      Passive_Store_Mgt,
6      System,
7      System_Defs,
8      Text_Mgt;
9
10 package body Employee_Filing_Ex is
11
12     max_employee_count: System.ordinal := 1_000;
13     -- A new employee file is limited to this many
14     -- employees.
15
16
17     procedure Store_DDef(
18         DDef:    Data_Definition_Mgt.DDef_AD;
19         name:    System_Defs.text)
20     is
21         -- Logic:
22         --   Stores a DDef and updates its passive
23         --   version.
24         --
25         untyped_DDef:    untyped_word;
26         FOR untyped_DDef USE AT DDef'address;
27         --
28     begin
29         begin
30             Directory_Mgt.Delete(name);
31         exception
32             when Directory_Mgt.no_access =>
33                 null;
34
35             when others =>
36                 RAISE;
37         end;
38         Directory_Mgt.Store(name, untyped_DDef);
39
40         Passive_Store_Mgt.Request_update(untyped_DDef);
41     end Store_DDef;
42
43     procedure Create_employee_DDef
44         -- New DDef object's pathname.
45     --
46     -- Logic:
47     --   Sets up a self-contained record DDef. This
48     --   DDef requires the properties indicated by
49     --   the following pseudo-DDef language:
50     --
51     --   define Employee_Data
52     --       record
53     --           Dept:    Type is ord_2,
54     --                   lower_bound is 100,
55     --                   upper_bound is 999;
56     --           Name:   Type is string,
57     --                   (System_Defs.text)
58     --                   Header_for_max_length is true,
59     --                   Varying is true,
60     --                   length is 25;
61     --           Job_Desc: Type is string,
62     --                   length is 200;
63     --           Salary:  Type is real4,
64     --                   default_value is 0;
65     --       end record;
66     --
67     --   This structure is equivalent to the following
68     --   Ada record declaration:
69     --
70     --   subtype Job_Desc length is
71     --       integer range 0.. 200;
72     --
73     --   Employee_Data(
74

```

PRELIMINARY

```

75  --      length: Job_Desc_length) is
76  --      record
77  --      dept:      short_ordinal range 100 .. 999;
78  --      name:      System_Defs.text(25);
79  --      job_Desc: string(1 .. length);
80  --      salary:   float;
81  --      end record;
82  --
83  -- "Data_Definition_Mgt" assigns layout
84  -- properties to the record that correspond to
85  -- the following Ada rep spec (note that the
86  -- holes in the record allow fields to be placed
87  -- on natural boundaries):
88  --
89  -- for Employee_Data use
90  -- record
91  --      dept      at 0 range 0 .. 15;
92  --      name      at 4 range 0 ..
93  --      8*(max_text_length+4)-1;
94  --      length    at 40 range 0 .. 15;
95  --      job_desc  at 42 range 0 ..
96  --      8*(job_desc_length)-1;
97  --      salary    at 36 range 0 .. 31;
98  --      end record;
99  is
100 dd:      Data_Definition_Mgt.DDef_AD;
101 name:    System_Defs.text(40);
102 rec_node: Data_Definition_Mgt.node_reference;
103 field_node: Data_Definition_Mgt.node_reference;
104 pv: Data_Definition_Mgt.property_value(100);
105 begin
106
107     dd := Data_Definition_Mgt.Create_DDef;
108         -- Create a new DDef object.
109
110
111     Text_Mgt.Set(name,"Employee_Data");
112     rec_node := Data_Definition_Mgt.Create_node(
113         -- Create a DDef node for the record layout.
114         dd,
115         -- AD to a DDef object
116         Data_Definition_Mgt.mt_record,
117         -- Record metatype and property value for
118         -- the "node_name" property ID.
119         name,
120         Data_Definition_Mgt.public_root_node);
121         -- Property value for the "root_value"
122         -- property ID.
123
124
125     Text_Mgt.Set (name,"Dept");
126
127     -- Create a simple metatype node with
128     -- "root_value" set to "non_root_node" for the
129     -- "Dept" field.
130     field_node := Data_Definition_Mgt.
131         Create_simple_field(
132             rec_node,
133             -- DDef object open for definition.
134             Data_Definition_Mgt.t_ord2,
135             -- Property value for "pi_type" property
136             -- ID (short ordinal of type "type_t").
137             name);
138             -- Property value for the "node_name"
139             -- property ID.
140
141     pv.simple_pv := (
142         pv_type => Data_Definition_Mgt.pv_int4,
143         int4_value => 100);
144         -- Set "pi_lower_bounds" (type integer) to
145         -- 100.
146
147     -- Add "pi_lower_bounds" and its value to the
148     -- "Dept" node.
149     Data_Definition_Mgt.Add_property_value(
150         field_node,
151         Data_Definition_Mgt.pi_lower_bounds,

```

## PRELIMINARY

```
152         pv);
153
154     pv.simple_pv.int4_value := 999;
155     -- Set "upper_bounds" property value.
156
157     Data_Definition_Mgt.Add_property_value(
158     -- Add "pi_upper_bounds" and its value to the
159     -- node.
160     field_node,
161     Data_Definition_Mgt.pi_upper_bounds,
162     pv);
163
164
165     Text_Mgt.Set (name,"Name");
166
167     -- Create a simple metatype node with
168     -- "root_value" set to "non_root_node" for the
169     -- "Name" field.
170     field_node := Data_Definition_Mgt.
171     Create_simple_field_with_prop(
172     rec_node,
173     -- DDef object that is open for
174     -- definition.
175     Data_Definition_Mgt.t_string,
176     -- Value for "pi_type" (uses byte-string
177     -- for "type_t").
178     name,
179     -- Value for "node_name".
180     Data_Definition_Mgt.
181     pi_header_for_max_length,
182     (Data_Definition_Mgt.pv_boolean,true));
183     -- True if string is represented in
184     -- SIL 'text' type.
185
186     pv.simple_pv := (
187     pv_type => Data_Definition_Mgt.pv_int4,
188     int4_value => 25);
189     -- Property value (type integer) is set to
190     -- 25.
191
192     Data_Definition_Mgt.Add_property_value(
193     field_node,
194     -- Node within an open DDef object.
195     Data_Definition_Mgt.pi_length,
196     pv);
197     -- Sets "pi_length" (maximum length of string in
198     -- bytes). Because "pi_header_for_max_length"
199     -- requires "pi_varying" to be false, "name" is
200     -- a fixed-size field.
201
202
203     Text_Mgt.Set (name,"Job_Desc");
204
205     -- Create a simple metatype node with
206     -- "root_value" set to "non_root_node".
207     field_node := Data_Definition_Mgt.
208     Create_simple_field_with_prop(
209     rec_node,
210     -- DDef object that is open for
211     -- definition.
212     Data_Definition_Mgt.t_string,
213     -- Value for "pi_type" (uses
214     -- byte-string for "type_t").
215     name,
216     -- Value for "pi_node_name".
217     Data_Definition_Mgt.pi_varying,
218     (Data_Definition_Mgt.pv_boolean,
219     true));
220     -- Varying-length string.
221
222     pv.simple_pv := (
223     pv_type => Data_Definition_Mgt.pv_int4,
224     -- Sets property value for "pi_length"
225     -- (maximum length of string in bytes) to
226     -- 200.
227     int4_value => 200);
228
```

**PRELIMINARY**

```

229 Data_Definition_Mgt.Add_property_value(
230   -- Adds "pi_length" and its value.
231   field_node,
232   -- Node within an open DDef object.
233   Data_Definition_Mgt.pi_length,
234   pv);
235
236
237 Text_Mgt.Set (name,"Salary");
238 field_node := Data_Definition_Mgt.
239   Create_simple_field_with_prop(
240     -- Create a simple metatype node with
241     -- "root_value" set to "non_root_node"
242     -- (defaults to 0).
243     rec_node,
244     -- DDef object that is open for
245     -- definition.
246     Data_Definition_Mgt.t_real8,
247     -- Value for "pi_type"
248     -- (uses real for "type_t").
249     name,
250     -- Value for "pi_node_name".
251     Data_Definition_Mgt.pi_default_value,
252     (Data_Definition_Mgt.pv_real8,0.0));
253
254 Data_Definition_Mgt.Close(dd);
255   -- Close and bind DDef object.
256
257   -- Save created DDef as "Employee_DDef".
258 Text_Mgt.Set (name,"Employee_DDef");
259 Store_DDef(DDef => dd, name => name);
260
261 end Create_employee_DDef;
262
263
264 procedure Create_dept_DDef
265   --
266   -- Logic:
267   -- Sets up an index key DDef for an employee
268   -- file by deriving fields from an existing
269   -- record DDef.
270   --
271 is
272   dd:      Data_Definition_Mgt.DDef_AD;
273   name:    System_Defs.text(40);
274   rec_node: Data_Definition_Mgt.node_reference;
275   field_node: Data_Definition_Mgt.node_reference;
276   pv:      Data_Definition_Mgt.
277             property_value(100);
278 begin
279   -- Create AD to a DDef object
280   dd := Data_Definition_Mgt.Create_DDef;
281   -- Create node for Index_2_DDef record
282   Text_Mgt.Set (name,"Index_2_DDef");
283   rec_node := Data_Definition_Mgt.Create_node(
284     dd,
285     -- AD to a DDef object
286     Data_Definition_Mgt.mt_record,
287     -- meta_type of 'record'
288     name,
289     -- value for the node_name
290     -- property
291     Data_Definition_Mgt.private_root_node);
292   -- can be referenced from
293   -- other DDef objects
294
295   -- Set DDef_name property
296   pv.simple_pv := (
297     pv_type => Data_Definition_Mgt.pv_string);
298
299   Text_Mgt.Set (pv.text_value, "Employee_Data");
300
301   Data_Definition_Mgt.Add_property_value(
302     rec_node,
303     -- node within an open DDef
304     Data_Definition_Mgt.pi_DDef_name,
305     -- requested property

```

## PRELIMINARY

```

306         pv);
307         -- value to be assigned
308     Text_Mgt.Set (pv.text value, "Employee_DDef");
309     Data_Definition_Mgt.Add_property_value(
310         rec_node,
311         -- node within an open DDef
312         Data_Definition_Mgt.pi_DDef_name,
313         -- requested property
314         pv);
315         -- value to be assigned
316
317     -- Set derive all property; false: all fields not
318     -- referred to.
319     pv.simple_pv := (
320         pv_type => Data_Definition_Mgt.pv_boolean,
321         -- property value has type boolean
322         boolean_value => false);
323     Data_Definition_Mgt.Add_property_value(
324         rec_node,
325         -- node within an open DDef
326         Data_Definition_Mgt.pi_derive_all,
327         -- requested property
328         pv);
329         -- value to be assigned
330
331     -- Create node for key field "Dept"
332     field_node := Data_Definition_Mgt.
333         Create_field(rec_node);
334     -- first key.
335
336     -- Set maps_to property
337     pv.simple_pv := (
338         pv_type => Data_Definition_Mgt.pv_string);
339     Text_Mgt.Set (pv.text_value, "Dept_DDef");
340     Data_Definition_Mgt.Add_property_value(
341         field_node,
342         -- node within an open DDef
343         Data_Definition_Mgt.pi_maps_to,
344         -- requested property
345         pv);
346         -- value to be assigned
347         -- Descending defaults to false;
348         -- it needn't be set.
349
350     -- close and bind DDef
351     Data_Definition_Mgt.Close(dd);
352
353     -- Save created DDef under the symbolic name
354     -- "Index_2_DDef"
355     Text_Mgt.Set(name, "Dept_Index_DDef");
356     Store_DDef(DDef => dd, name => name);
357
358
359 end Create_dept_DDef;
360
361
362
363 procedure Create_dept_salary_DDef
364     --
365     -- Logic:
366     -- Sets up an index key DDef for an employee
367     -- file by deriving fields from an existing
368     -- record DDef.
369     --
370 is
371     dd:      Data_Definition_Mgt.DDef_AD;
372     name:    System_Defs.text(40);
373     -- New DDef object's pathname.
374     rec_node: Data_Definition_Mgt.node_reference;
375     field_node: Data_Definition_Mgt.node_reference;
376     pv:      Data_Definition_Mgt.property_value(100);
377 begin
378     -- Create AD to a DDef object
379     dd := Data_Definition_Mgt.Create_DDef;
380
381     -- Create node for Employee_DDef record
382     Text_Mgt.Set(name, "Employee_DDef");

```

PRELIMINARY

```

383   rec_node := Data_Definition_Mgt.Create_node(
384     dd,
385     -- AD to a DDef object
386     Data_Definition_Mgt.mt_record,
387     -- meta_type = record
388     name,
389     -- Value for the node_name property
390     Data_Definition_Mgt.private_root_node);
391     -- Can be referenced from other DDef objects.
392
393   -- Set DDef_name property
394   pv.simple_pv := (
395     pv_type => Data_Definition_Mgt.pv_string);
396   Text_Mgt.Set (pv.text_value, "Employee Data");
397   Data_Definition_Mgt.Add_property_value(
398     rec_node,
399     -- Node within an open DDef.
400     Data_Definition_Mgt.pi_DDef_name,
401     -- Requested property.
402     pv);
403     -- Value to be assigned.
404   Text_Mgt.Set (pv.text_value, "Employee DDef");
405   Data_Definition_Mgt.Add_property_value(
406     rec_node,
407     -- Node within an open DDef.
408     Data_Definition_Mgt.pi_DDef_name,
409     -- Requested property.
410     pv);
411     -- Value to be assigned.
412
413   -- Set derive_all property; false: all fields not
414   -- referred to.
415   pv.simple_pv := (
416     pv_type => Data_Definition_Mgt.pv_boolean,
417     -- property value has type boolean
418     boolean_value => false);
419   Data_Definition_Mgt.Add_property_value(
420     rec_node,
421     -- node within an open DDef
422     Data_Definition_Mgt.pi_derive_all,
423     -- requested property
424     pv);
425     -- value to be assigned
426
427   -- Create node for key field "Dept"
428   field_node := Data_Definition_Mgt.
429     Create_field(rec_node);
430     -- first key.
431
432   -- Set maps_to property
433   pv.simple_pv := (
434     pv_type => Data_Definition_Mgt.pv_string);
435   Text_Mgt.Set (pv.text_value, "Dept");
436   Data_Definition_Mgt.Add_property_value(
437     field_node,
438     -- node within an open DDef
439     Data_Definition_Mgt.pi_maps_to,
440     -- requested property
441     pv);
442     -- value to be assigned
443     -- Descending defaults to false;
444     -- it needn't be set.
445
446   -- Create node for key field "Salary"
447   field_node := Data_Definition_Mgt.Create_field(
448     rec_node);
449
450   -- Set maps_to property
451   pv.simple_pv := (
452     pv_type => Data_Definition_Mgt.pv_string);
453   Text_Mgt.Set (pv.text_value, "Salary");
454   Data_Definition_Mgt.Add_property_value(
455     field_node,
456     -- node within an open DDef
457     Data_Definition_Mgt.pi_maps_to,
458     -- requested property
459     pv);

```

## PRELIMINARY

```

460         -- value to be assigned
461
462     -- Set descending property; true: order is
463     -- descending
464     pv.simple_pv := (
465         pv_type => Data_Definition_Mgt.pv_boolean,
466         -- property value has type boolean
467         boolean_value => true);
468     Data_Definition_Mgt.Add_property_value(
469         field_node,
470         -- node within an open DDef
471         Data_Definition_Mgt.pi_descending,
472         -- requested property
473         pv);
474         -- value to be assigned
475
476     -- close and bind DDef
477     Data_Definition_Mgt.Close(dd);
478
479     Text_Mgt.Set(name, "Dept_Salary_Index_DDef");
480     Store_DDef(DDef => dd, name => name);
481     -- Save created DDef under the symbolic name
482     -- "Index_DDef"
483
484
485 end Create_dept_salary_DDef;
486
487
488 procedure Create_file_and_indexes(
489     file_name:      System_Defs.text;
490     -- New file's pathname.
491     org_index_name: System_Defs.text)
492     -- Organization index's name.
493     --
494     -- Logic:
495     -- Define descriptors for the file, the organization index,
496     -- and the alternate index. Create the file, build the
497     -- organization index, and build the alternate index.
498     --
499     -- Note:
500     -- You build the organization index built after creating
501     -- the file, and the alternate index after creating the
502     -- organization index.
503     --
504 is
505     new_file: File_Defs.file_AD;
506 begin
507     -- Create the file first.
508     new_file := File_Admin.Create_file(
509         name => file_name,
510         logical_file_descr => (
511             -- Set the file's logical
512             -- file descriptor.
513             file_org      => File_Defs.unordered,
514             DDef_specified => true,
515             term_char     => File_Defs.term_char,
516             record_DDef   => employee_DDef,
517             record_layout => (
518                 DDef_specified => true),
519             lock_escalation_count => 0,
520             xm_locking      => true,
521             -- Required for any record locking,
522             -- including transaction locking.
523             short_term_logging => true,
524             -- Required for transaction support.
525             long_term_logging  => false,
526             max_rec_num       =>
527                 max_employee_count,
528             bytes_per_bucket  => 4096,
529             fill_factor       =>
530                 File_Admin.fill_factor_dont_care,
531             org_index        => org_index_name));
532
533     -- Build the organization index for the file.
534     File_Admin.Build_index(
535         file => new_file,
536         logical_index_descr => (

```

## PRELIMINARY

```
537         -- Set the index descriptor for Department.
538         name           => dept_index_name,
539         active          => true,
540         index_org       =>
541             File_Defs.btree_index,
542         duplicates_allowed => false,
543         duplicate_order  =>
544             File_Defs.by_increasing_record_ID,
545         null_attribute   => File_Defs.none,
546         DDef             => dept_index_DDef,
547         phantom_protected => false,
548         utilization_maintenance => true,
549         bytes_per_bucket =>
550             File_Defs.page_size));
551
552     -- Build an alternate index for the file.
553     File_Admin.Build_index(
554         file => new_file,
555         logical_index_descr => (
556             name           =>
557                 dept_salary_index_name,
558             active          => true,
559             index_org       =>
560                 File_Defs.btree_index,
561             -- A unordered org index with
562             -- a b-tree index.
563             duplicates_allowed => false,
564             duplicate_order  =>
565                 File_Defs.by_increasing_record_ID,
566             null_attribute   =>
567                 File_Defs.none,
568             DDef             =>
569                 dept_salary_index_DDef,
570             phantom_protected => true,
571             -- Uses bucket-level locking.
572             utilization_maintenance => true,
573             bytes_per_bucket =>
574                 File_Defs.page_size));
575
576     end Create_file_and_indexes;
577
578 end Employee_Filing_Ex;
579
```

**X-A.4.5 Hello\_ada\_ex Procedure**

```
1 with Text_IO;
2
3 procedure Hello_ada_ex is
4   --
5   -- Function:
6   -- Write "Hello, world!" on a separate line to the
7   -- standard output, using Ada's "Text_IO" package.
8 begin
9   Text_IO.Put_line("Hello, world!");
10 end Hello_ada_ex;
```

## X-A.4.6 Hello\_OS\_ex Procedure

```
1  with Byte_Stream_AM,
2      Device_Defs,
3      Process_Mgt,
4      Process_Mgt_Types,
5      System;
6
7  procedure Hello_OS_ex is
8      --
9      -- Function:
10     -- Write "Hello, world!" on a separate line to the
11     -- standard output, using OS packages.
12
13     hello: constant string := "Hello, world!" & ASCII.LF;
14     stdout: constant Device_Defs.opened_device :=
15         Process_Mgt.Get_process_globals_entry(
16             Process_Mgt_Types.standard_output);
17 begin
18     Byte_Stream_AM.Ops.Write(
19         opened_dev => stdout,
20         buffer_VA  => hello(1)'address,
21         length     => System.ordinal(hello'length));
22 end Hello_OS_ex;
```

## X-A.4.7 Join\_File\_Ex Package Specification

```

1  with Join_Interface,
2      System;
3  package Join_File_Ex is
4      --
5      -- Function:
6      --   This package provides examples using
7      --   the DBMS support operations.
8      --
9      -- History:
10     --   08-10-87, Paul Schwabe: initial revision.
11     --   11-30-87, Paul Schwabe: update.
12     --
13     pragma external;
14
15     -- Define some user buffer.
16     --
17     type stuff_buffer_type is
18         array(1 .. 256) of character;
19
20     -- Define local data structures.
21     --
22     type some_other_type is
23         array(1 .. 256) of character;
24
25     type user_info_type is
26         record
27             first_call:    boolean := true;
28             -- This is reset by the user join procedure
29             -- during the first call.
30             comm_block:    Join_Interface.communication_block_VA;
31             -- This is returned by the user join
32             -- procedure.
33             user_specific: some_other_type;
34             -- Needed for the user's join algorithm.
35         end record;
36
37     function Join_ex(
38         buffers_available:    System.ordinal;
39         -- Number of 4kbyte file buffers reserved
40         -- for this join.
41         user_info:           System.address;
42         -- Object for user process specific storage.
43         records:             Join_Interface.record_lists_AD)
44         -- The list of record locations for each
45         -- input device. Those are null the first time
46         -- this routine is called.
47         return Join_Interface.communication_block_VA;
48         -- Contains the 'next block list' and the
49         -- output buffers.
50     pragma subprogram_value(Join_Interface.Block_join, Join_ex);
51     --
52     -- Function:
53     --   The function Join_ex (subprogram type
54     --   Join_Interface.Block_join) will be called
55     --   from inside the Join_Interface.Join. (After
56     --   having locked all the participating input
57     --   devices on file level, we call the Join).
58
59
60     procedure Join_call(
61         num_input_devices: System.short_ordinal);
62         -- Number of participating devices.
63     -- Function:
64     --   Calls the Join procedure.
65     --
66
67 end Join_File_Ex;
68
69
70

```

## X-A.4.8 Join\_File\_Ex Package Body

```

1  with Device_Defs,
2      Join_Interface,
3      System,
4      Unchecked_Conversion;
5  package body Join_File_Ex is
6      --
7      -- Logic:
8      --   This package body contains the implementations
9      --   for the examples using the DBMS support
10     --   operations.
11     --
12
13     --
14     --           UNCHECKED CONVERSIONS
15     --
16
17     function Convert_comm_block_VA_to_address is
18         new Unchecked_Conversion(
19             source => Join_Interface.
20                 communication_block_VA,
21             target => System.address);
22
23     function Convert_address_to_comm_block_VA is
24         new Unchecked_Conversion(
25             source => System.address,
26             target => Join_Interface.
27                 communication_block_VA);
28
29     function Convert_address_to_next_block_VA is
30         new Unchecked_Conversion(
31             source => System.address,
32             target => Join_Interface.
33                 next_block_list_VA);
34
35     function Convert_next_block_VA_to_address is
36         new Unchecked_Conversion(
37             source => Join_Interface.
38                 next_block_list_VA,
39             target => System.address);
40
41
42     -----
43     --           BODY FOR THE SUBPROGRAM TYPE BLOCK_JOIN           --
44     -----
45
46     function Join_ex(
47         buffers_available: System.ordinal;
48         -- Number of 4kbyte file buffers reserved
49         -- for this join.
50         user_info: System.address;
51         -- Object for user specific storage.
52         records: Join_Interface.record_lists_AD)
53         -- The list of record locations for each
54         -- input device. Those are null the first time
55         -- this routine is called.
56     return Join_Interface.communication_block_VA
57         -- Contains the 'next block list' and the
58         -- output buffers.
59     --
60     -- Operation:
61     --
62     is
63         u_info:         user_info_type;
64         FOR u_info USE AT user_info;
65         -- Retypes the address to user_info_type.
66
67         comm_block:    Join_Interface.communication_block;
68         FOR comm_block USE AT
69             Convert_comm_block_VA_to_address(
70                 u_info.comm_block);
71         -- Just a rename.
72
73         num_devices:   System.short_ordinal :=
74             records.num_devices;

```

## PRELIMINARY

```
75         -- Number of input devices for this Join.
76
77     begin
78
79         -- First distribute the 'buffers_available' among
80         -- the input devices in some manner. Make sure the
81         -- number of buffers requested at a time does not
82         -- exceed the numbers of buffers available.
83         --
84         -- .... lets say 2 buckets per block per input
85         -- file is the result.
86
87
88     if u_info.first_call then
89         -- This is the first time this function is
90         -- called. (This can also be recognized by
91         -- checking the ADs in 'records', which are null
92         -- at this time).
93
94     for i in 1 .. num_devices loop
95         -- Set up the communication block to condition
96         -- Join for the next call.
97
98         comm_block.position_blocks.next_blocks(i).
99             block_size := 2;
100        -- Two buckets per block.
101
102        comm_block.position_blocks.next_blocks(i).
103            position := Join_Interface.next;
104        -- We want to trace through the files from
105        -- the beginning to the end. The Join will
106        -- call this function the next time with
107        -- record locations of those records
108        -- contained in the first two buckets of the
109        -- input file i. "Current" would deliver
110        -- empty record location arrays at this
111        -- stage. "Previous" would start with the
112        -- last two buckets in the file.
113
114    end loop;
115
116    else
117        -- This is not the first call to this function.
118
119        -- Here is where a join algorithm takes place.
120        --
121        -- If i counts the devices from 1 ..
122        -- num_devices, and if j counts the number of
123        -- entries in one record_location_array (1 ..
124        -- num_records), then the necessary data for the
125        -- join algorithm can be retrieved
126        -- via the following paths:
127
128        -- num_records := records.rec_list_array(i).
129        -- num_entries;
130        -- Number of records per record location array.
131
132        -- One record can be found in:
133        --
134        -- records.rec_list_array(i).rec_loc_array(j).
135        --     record_VA
136        -- records.rec_list_array(i).rec_loc_array(j).
137        --     record_length
138        -- records.rec_list_array(i).rec_loc_array(j).
139        --     record_ID
140
141        -- If the buckets scanned do not contain any
142        -- records then the "number_of_entries" will be
143        -- 0. It will be
144        -- "Join_Interface.null_num_entries" when the
145        -- end of the file has been exceeded.
146
147        -- Now, join the records into the
148        -- 'buffer_with_stuff'. . . . .
149
150        -- Set up the comm_block with respect to the
151        -- output buffers.
```

**PRELIMINARY**

```

152      --
153      -- comm_block.out_buffers.output_length :=
154      -- some_value;
155      --The length of the buffer contents
156      -- in bytes. A non zero value provides for
157      -- flushing the buffer to the output device.
158
159      -- Set up the communication block with
160      -- positioning information for the
161      -- subsequent call:
162      --
163      -- comm_block.position_blocks(i).block_size := 2;
164      -- Two buckets per block.
165
166      -- comm_block.position_blocks(i).position :=
167      --   Join_Interface.next;
168      -- Makes the Join call this
169      -- function the next time with record locations
170      -- of those records contained in the next two
171      -- buckets of the input file i.
172      null;
173
174      end if;
175
176      return u_info.comm_block;
177
178      end Join_ex;
179
180
181      -----
182      --                               THE CALL                               --
183      -----
184
185      -- The function Join_ex (subprogram type
186      -- Join_Interface.Block_join) will be called from
187      -- inside the Join_Interface.Join.
188
189      -- (After having locked all the participating input
190      -- devices on file level, we call the Join).
191
192
193      procedure Join_call(
194          num_input_devices: System.short_ordinal)
195          -- Number of participating devices.
196      -- Operation:
197      --   Calls the Join procedure.
198      --
199      is
200          join_devices: Join_Interface.join_device_list(
201              num_input_devices);
202          -- Input devices for the Join.
203
204          out_file: Device_Defs.opened_device;
205          -- Output rec_ID_stream device.
206
207          buffer_reservation: Join_Interface.
208              buffer_reservation_block;
209          -- Block which determines the number of buffers
210          -- needed.
211
212          u_info: user_info_type;
213          -- Global storage for the Block_join procedure.
214          -- Will be passed to Block_join.
215
216          comm_block: Join_Interface.communication_block;
217          -- Instantiates the communication block.
218          -- Contains the next_block list.
219
220          buffer_with_stuff: stuff_buffer_type;
221          -- User records that will be copied to the output.
222
223          length_of_one_stuff_record: constant
224              System.ordinal := 8;
225          -- Constant size of the "stuff records".
226          -- the output buffers;
227
228          next_blocks: Join_Interface.next_block_list(

```

## PRELIMINARY

```

229         num_entries => num_input_devices);
230     -- The list that specifies which blocks to use
231     -- for the next call.
232
233 begin
234
235     -- Hook the comm_block into user info.
236     --
237     u_info.comm_block :=
238         Convert_address_to_comm_block_VA(
239             comm_block'address);
240
241     -- Initialize the comm_block.
242     --
243     comm_block.position_blocks :=
244         Convert_address_to_next_block_VA(
245             next_blocks'address);
246     -- Unchecked conversion; see Ada-G.
247
248     -- Set up the communication block with respect to
249     -- the output buffers.
250     --
251     comm_block.out_buffers.output_buffer      :=
252         buffer_with_stuff'address;
253     comm_block.out_buffers.record_size        :=
254         length_of_one_stuff_record;
255     comm_block.out_buffers.alt_output_buffer :=
256         System.null_address;
257     comm_block.out_buffers.alt_record_size    := 0;
258     --
259     -- Here, the descriptors for the output buffers
260     -- have to be set to make sure the buffers don't
261     -- get flushed, since they do not contain any
262     -- interesting data.
263     --
264     comm_block.out_buffers.output_length      := 0;
265     comm_block.out_buffers.alt_output_length := 0;
266
267     -- Get the ODOs for the input devices from somewhere.
268     --
269     join_devices := (. . .);
270
271     -- Calculate how much buffers should be reserved
272     -- by the Join at a time. Determine how many you
273     -- need as a minimum; what's the optimal number?
274     -- Do you want to wait until the buffers are
275     -- available?
276     --
277     buffer_reservation := (...);
278
279     -- Create and/or Open the output device
280     --
281     out_file := ....
282
283     -- Initialize the user info.
284     --
285     u_info := ....
286
287     -- And off we go:
288     --
289     Join_Interface.Join(
290         participating_devices => join_devices,
291         buffers_to_reserve    => buffer_reservation,
292         user_info              => u_info'address,
293         join_procedure         =>
294             Join_ex'subprogram_value,
295         join_output            => out_file,
296         alternate_output       => System.null_word);
297
298 end Join_call;
299
300
301 end Join_File_Ex;
302
303
304

```

**X-A.4.9 Record\_Locking\_Ex Package Specification**

```
1 with Device_Defs,
2   System_Defs;
3 package Record_Locking_Ex is
4   --
5   -- Function:
6   --   This package contains the examples for
7   --   using the record locking in your
8   --   applications.
9   --
10  -- History:
11  --   01-07-88, Paul Schwabe: initial version.
12  --
13  pragma external;
14
15  procedure Level_3_update(
16    file_name: System_Defs.text);
17    --
18    -- Function:
19    --   This example is designed to illustrate level
20    --   3 consistency. It reads the employee records
21    --   in a key range and updates the salaries.
22    --
23    --   Does an index-sequential read of an
24    --   unordered file using a single b-tree alternate
25    --   index. The read call uses a "write" lock mode
26    --   because the record will be updated after the read.
27    --
28
29  end Record_Locking_Ex;
30
```

## X-A.4.10 Record\_Locking\_Ex Package Body

```

1  with Device_Defs,
2      Employee_Filing_Ex,
3      File_Admin,
4      File_Defs,
5      Record_AM,
6      System,
7      System_Defs,
8      Text_Mgt,
9      Transaction_Mgt;
10
11 use System;
12
13 package body Record_Locking_Ex is
14     --
15     -- Logic:
16     -- This package body contains the
17     -- the implementations for the record
18     -- locking examples.
19     --
20     buffer: string(1 .. integer(
21         Employee_Filing_Ex.max_rec_size));
22     -- Buffer is large enough to hold any employee
23     -- record.
24
25     current_record_addr: constant
26         System.address := buffer'address;
27     current_record_VA: constant
28         Employee_Filing_Ex.employee_record_VA :=
29         Employee_Filing_Ex.Employee_record_VA_from_VA(
30             current_record_addr);
31
32     bytes_read: System.ordinal;
33     -- Number of bytes in current record.
34
35
36     procedure Level_3_update(
37         file_name: System_Defs.text)
38         -- An opened device for transaction T1, opened
39         -- for input on an employee file.
40         --
41         -- Operation:
42         -- Reads all records in a relative file and
43         -- totals the salaries.
44         --
45         -- Does an index-sequential read of an
46         -- unordered file using a single b-tree alternate
47         -- index. Transaction T1 (a reader) reads
48         -- employee records using the write_lock lock
49         -- mode, locking the file from other readers and
50         -- writers.
51         --
52     is
53         opened_file: Device_Defs.opened_device;
54
55         total_salary: Employee_Filing_Ex.monthly_salary
56             := 0.00;
57
58         start_key_value: constant Employee_Filing_Ex.
59             dept_salary_key_buffer := (
60             dept      => 100,
61             -- Lowest department, ascending.
62             salary    => 10_000.00);
63             --Highest salary, descending.
64
65         stop_key_value: constant Employee_Filing_Ex.
66             dept_salary_key_buffer := (
67             dept      => 500,
68             -- Highest department, ascending.
69             salary    => 1_000.00);
70             -- Lowest salary, descending.
71
72         level_3_mode: Record_AM.open_mode_value(Record_AM.level_3) :=
73             (mode_id => Record_AM.level_3,
74              value   => true);

```

```

75
76 begin
77   Transaction_Mgt.Start_transaction;
78   -- Started on behalf of transaction T1,
79   -- the level 3 reader.
80   -- Any updates, deletes or inserts
81   -- (not shown) within this transaction
82   -- can be rolled back if
83   -- the transaction aborts.
84
85   opened_file := Record_AM.Open_by_name(
86     name      => file_name,
87     input_output => Device_Defs.inout,
88     allow     => Device_Defs.anything);
89
90   Record_AM.Ops.Set_open_mode(
91     opened_dev => opened_file,
92     mode_value => level_3_mode);
93   -- Sets level 3 consistency.
94
95   Record_AM.Keyed_Ops.Set_key_range(
96     opened_file,
97     index => Employee_Filing_Ex.
98       dept_salary_index_name,
99     select_range => (
100      start_comparison =>
101        Record_AM.inclusive,
102      start_value      => (
103        start_key_value'address,
104        start_key_value'size / 8),
105      stop_comparison  =>
106        Record_AM.inclusive,
107      stop_value       => (
108        stop_key_value'address,
109        stop_key_value'size / 8)));
110
111   loop
112     bytes_read := Record_AM.Ops.Read(
113       opened_dev => opened_file,
114       buffer_VA  => current_record_addr,
115       length     => Employee_Filing_Ex.
116         max_rec_size,
117       lock       => Record_AM.write_lock,
118       unlock     => Record_AM.no_unlock);
119     -- Another caller cannot read or update
120     -- the same record at any time.
121
122     if current_record_VA.salary = 3_000.00 then
123       current_record_VA.salary :=
124         current_record_VA.salary + 300.00;
125
126       Record_AM.Ops.Update(
127         opened_dev => opened_file,
128         modifier   => Record_AM.current,
129         buffer_VA  => current_record_addr,
130         length     => Employee_Filing_Ex.
131           max_rec_size,
132         timeout    => Record_AM.wait_forever,
133         status     => null);
134     end if;
135   end loop;
136
137   exception
138     when Device_Defs.end_of_file =>
139       Transaction_Mgt.Commit_transaction;
140     -- Everthing's OK.
141
142     when others =>
143       -- Something's bad.
144       null;
145   end Level_3_update;
146 end Record_Locking_Ex;

```

## X-A.4.11 Output\_bytes\_ex Procedure

```

1  with Byte_Stream_AM,
2     Device_Defs,
3     Process_Mgt,
4     Process_Mgt_Types,
5     System,
6     System_Defs,
7     Unchecked_conversion;
8
9  procedure Output_bytes_ex(
10     name: System_Defs.text)
11     -- Input device to read.
12     --
13     -- Function:
14     --   Opens the named input device and
15     --   copies bytes from it to the caller's
16     --   standard output, until end-of-file.
17     is
18     source_opened_device: Device_Defs.opened_device;
19     dest_opened_device:   Device_Defs.opened_device;
20     function Opened_device_from_untyped is new
21         Unchecked_conversion(
22             source => System.untyped_word,
23             target => Device_Defs.opened_device);
24     BUFSIZE: constant System.ordinal := 4_096;
25     buffer:   array(1 .. BUFSIZE) of
26         System.byte_ordinal;
27     bytes_read: System.ordinal;
28     begin
29         source_opened_device :=
30             Byte_Stream_AM.Open_by_name(
31                 name => name,
32                 input_output => Device_Defs.input,
33                 allow => Device_Defs.readers);
34         dest_opened_device := Opened_device_from_untyped(
35             Process_Mgt.Get_process_globals_entry(
36                 Process_Mgt_Types.standard_output));
37
38         loop
39             bytes_read := Byte_Stream_AM.Ops.Read(
40                 source_opened_device,
41                 buffer'address,
42                 BUFSIZE);
43             Byte_Stream_AM.Ops.Write(
44                 dest_opened_device,
45                 buffer'address,
46                 bytes_read);
47         end loop;
48     exception
49         when Device_Defs.end_of_file =>
50             Byte_Stream_AM.Ops.Close(
51                 source_opened_device);
52     end Output_bytes_ex;

```

## X-A.4.12 Output\_records\_ex Procedure

```

1  with Device_Defs,
2      Object_Mgt,
3      Process_Mgt,
4      Process_Mgt_Types,
5      Record_AM,
6      System,
7      System_Defs,
8      Unchecked_conversion;
9
10 procedure Output_records_ex(
11     name: System_Defs.text)
12     -- Pathname of device. Caller must have
13     -- read rights.
14     --
15     -- Operation:
16     -- Opens a named device, reads a stream
17     -- of records, and writes the records to
18     -- the caller's standard output, until
19     -- end-of-file.
20     --
21     -- Notes:
22     -- The record buffer is dynamically sized
23     -- so that records of any length can be
24     -- handled. Recovery from buffer overflow
25     -- uses the "rest_of_current" rather than
26     -- "current" read option, because some
27     -- devices, such as pipes, do not support
28     -- the "current" option.
29     --
30     -- Exceptions:
31     -- Device_Defs.device_in_use -
32     --   The device is being used by
33     --   an application that does not
34     --   allow concurrent readers.
35     -- Device_Defs.open_mode_conflict -
36     --   The named object does not
37     --   allow opens for input.
38     -- Device_Defs.device_inconsistent
39     -- Device_Defs.device_offline
40     -- Device_Defs.device_inoperative
41     -- Device_Defs.transfer_error
42     -- Directory_Mgt.no_access -
43     --   There is no such pathname
44     --   or the caller does not have
45     --   access to the named device.
46     -- Directory_Mgt.name_too_long -
47     --   The pathname or some part of it
48     --   exceeds an OS size limit.
49     -- File_Defs.volume_space_exhausted
50     -- Record_AM.XXX -
51     --   Many "Record AM" exceptions
52     --   can be raised. See "Read" and
53     --   "Insert" in "Record_AM.Ops".
54 is
55 use System; -- Import ordinal operators.
56 source_opened_device: Device_Defs.opened_device;
57 dest_opened_device: Device_Defs.opened_device;
58 buffer_size: System.ordinal := 256;
59 buffer_AD: System.untyped_word :=
60     Object_Mgt.Allocate(buffer_size/4);
61     -- 64 words (256 bytes) is the initial buffer
62     -- size. Buffer size is increased as needed.
63     -- The buffer is in a separate object for easy
64     -- resizing.
65 bytes_read: System.ordinal := 0;
66     -- If record requires multiple "Read" calls,
67     -- then this variable tracks bytes read so far.
68 read_status_VA: Record_AM.operation_status_VA :=
69     new Record_AM.operation_status_record;
70 read_position: Record_AM.position_modifier :=
71     Record_AM.next;
72     -- If record requires multiple "Read" calls,
73     -- then this variable is assigned
74     -- "Record_AM.rest_of_current" for the

```

```

75     -- 2nd through Nth reads.
76     function Opened_device_from_untyped is new
77        Unchecked_conversion(
78             source => System.untyped_word,
79             target => Device_Defs.opened_device);
80     begin
81         source_opened_device := Record_AM.Open_by_name(
82             name => name,
83             input_output => Device_Defs.input,
84             allow => Device_Defs.readers);
85         dest_opened_device := Opened_device_from_untyped(
86             Process_Mgt.Get_process_globals_entry(
87                 Process_Mgt_Types.standard_output));
88
89     loop
90
91         loop
92             begin
93                 bytes_read := bytes_read +
94                     Record_AM.Ops.Read(
95                         source_opened_device,
96                         read_position,
97                         System.address'(
98                             bytes_read,
99                             buffer_AD),
100                        buffer_size - bytes_read,
101                        status => read_status_VA);
102
103                 -- When control reaches this point, "Read"
104                 -- succeeded without a length error and
105                 -- this loop can be exited.
106                 EXIT;
107
108             exception
109                 when Device_Defs.length_error =>
110                     buffer_size := read_status_VA.rec_length;
111                     if buffer_size =
112                         Record_AM.unknown_length then
113                         buffer_size := 2 * 4 *
114                             Object_Mgt.Get_object_size(buffer_AD);
115                         -- Double the buffer size if an exact
116                         -- new size is not available.
117                     end if;
118                     Object_Mgt.Resize(
119                         buffer_AD,
120                         (buffer_size+3)/4);
121                     -- May make object even bigger than
122                     -- requested, but that's OK.
123                     read_position := Record_AM.rest_of_current;
124                 end;
125             end loop;
126
127             Record_AM.Ops.Insert(
128                 dest_opened_device,
129                 System.address'(0, buffer_AD),
130                 bytes_read);
131
132             -- Reset variables to read the next record
133             -- into the beginning of the buffer:
134             --
135             bytes_read := 0;
136             read_position := Record_AM.next;
137         end loop;
138
139     exception
140         when Device_Defs.end_of_file =>
141             Record_AM.Ops.Close(source_opened_device);
142     end Output_records_ex;

```

## X-A.4.13 Print\_cmd\_ex Procedure

```

1  with Byte_Stream_AM,
2      CL_Defs,
3      Command_Handler,
4      Device_Defs,
5      Directory_Mgt,
6      Print_Cmd_Messages, -- Message package.
7      Incident_Defs,
8      Message_Services,
9      Process_Mgt,
10     Process_Mgt_Types,
11     Spool_Defs,
12     Spool_Device_Mgt,
13     String_List_Mgt,
14     System,
15     System_Defs,
16     Text_Mgt;
17
18 procedure Print_cmd_ex
19 --
20 -- Function:
21 --   Defines a command to print from a file or other
22 --   byte stream source
23 --
24 -- Command Definition:
25 --   The command has the form:
26 --
27 --   print
28 --       [source=<pathname>]
29 --       [on=<pathname>]
30 --
31 -- The on argument can either be a spool queue or a
32 -- printer (for direct printing). The default is a
33 -- system standard spooling device. The source
34 -- argument will default to standard input.
35 --
36 --*C* set.message_file :file = \
37 --*C*   /examples/msg/example_messages
38 --*C*
39 --*C* create.command :cmd_def = print.inv_cmd \
40 --*C*   :cmd_name = print
41 --*C*
42 --*C*   define.argument source
43 --*C*     :type = string
44 --*C*     set.lexical_class symbolic_name
45 --*C*     set.maximum_length 252
46 --*C*     set.value_default ""
47 --*C*   end
48 --*C*
49 --*C*   define.argument on
50 --*C*     :type = string
51 --*C*     set.lexical_class symbolic_name
52 --*C*     set.maximum_length 80
53 --*C*     set.value_default ""
54 --*C*   end
55 --*C* end
56 --*C*
57 --*C* run "store.command_definitions \
58 --*C*   :program = print \
59 --*C*   :invocation_cmd = print.inv_cmd"
60 --*C*
61 --*C* run "store.default_message_file \
62 --*C*   print \
63 --*C*   print.msg"
64
65 is
66
67   use System;
68
69
70   opened_cmd:      Device_Defs.opened_device;
71                   -- Opened command input device.
72
73   -- source variables
74   source:          System_Defs.text (252);

```

PRELIMINARY

```

75         -- Pathname of file or device
76         -- print from
77
78     open_source:      Device_Defs.opened_device;
79
80     -- "on" variables
81     on_device:        System_Defs.text(Incident_Defs.txt_length);
82         -- Pathname of spool queue or
83         -- printer
84
85     on_untyped:       System.untyped_word;
86
87     spool_queue:      Device_Defs.device;
88
89     print_device:     Device_Defs.device;
90
91     no_print_device:  exception;
92
93     sheet_size:       constant Spool_Defs.size_t :=
94         (132,66);
95
96     open_print:       Device_Defs.opened_device;
97
98     -- buffer variables
99     buffer_size:      constant System.ordinal := 4_096;
100    buffer:            array(1..buffer_size) of
101        System.byte_ordinal;
102    bytes_read:        System.ordinal;
103
104    begin
105
106        -- Get command arguments:
107        --
108        opened_cmd :=
109            Command_Handler.
110            Open_invocation_command_processing;
111        Command_Handler.Get_string(opened_cmd, 1,
112            arg_value => source);
113        Command_Handler.Get_string(opened_cmd, 2,
114            arg_value => on_device);
115        Command_Handler.Close(opened_cmd);
116
117        -- assign defaults if parameter was not specified
118
119        if source.length = 0 then
120            open_source :=
121                Process_Mgt.Get_process_globals_entry(
122                Process_Mgt.Types.standard_input);
123            -- standard input from terminal
124        else
125            open_source := Byte_Stream_AM.Open_by_name(
126                name => source,
127                input_output => Device_Defs.input);
128        end if;
129
130        if on_device.length = 0 then
131            Text_Mgt.Set(on_device, "/dev/lpq");
132            -- Correct name of default system spool queue is
133            -- TBD
134        end if;
135
136        -- check the "on_device" for spooled or direct
137        -- printing, else error
138
139        on_untyped := Directory_Mgt.Retrieve(on_device);
140        if Spool_Defs.Is_spool_queue(on_untyped) then
141            print_device :=
142                Spool_Device_Mgt.Create_print_device(
143                spool_queue => spool_queue,
144                pixel_units => false,
145                print_area => sheet_size);
146        elsif Spool_Defs.Is_print_device(on_untyped) then
147            print_device :=
148                Spool_Device_Mgt.Create_print_device(
149                spool_queue => spool_queue,
150                pixel_units => false,

```

PRELIMINARY

```

152         print_area => sheet_size,
153         print_mode => Spool_Defs.page_wise);
154         -- direct printing
155
156     else
157         RAISE no_print_device;
158     end if;
159
160     open_print := Byte_Stream_AM.Ops.Open(
161                 print_device,
162                 Device_Defs.output);
163
164     while not
165         Byte_Stream_AM.Ops.At_end_of_file(open_source)
166     loop
167         bytes_read := Byte_Stream_AM.Ops.Read(
168             opened_dev => open_source,
169             buffer_VA => buffer'address,
170             length     => buffer_size);
171
172         Byte_Stream_AM.Ops.Write(
173             opened_dev => open_print,
174             buffer_VA  => buffer'address,
175             length     => bytes_read);
176     end loop;
177
178     Byte_Stream_AM.Ops.Close(open_source);
179     Byte_Stream_AM.Ops.Close(open_print);
180
181     exception
182     when no_print_device =>
183         Message_Services.Write_msg(
184             Print_Cmd Messages.no_print_device_code,
185             Incident_Defs.message_parameter(
186                 typ => Incident_Defs.txt,
187                 len => on_device.max_length)' (
188                 typ => Incident_Defs.txt,
189                 len => on_device.max_length,
190                 txt_val => on_device));
191
192     when Spool_Device_Mgt.units_not_supported =>
193         Message_Services.Write_msg(
194             Print_Cmd Messages
195                 .units_not_supported_code,
196             Incident_Defs.message_parameter(
197                 typ => Incident_Defs.txt,
198                 len => on_device.max_length)' (
199                 typ => Incident_Defs.txt,
200                 len => on_device.max_length,
201                 txt_val => on_device));
202     end Print_cmd_ex;
203

```

## X-A.4.14 Print\_Cmd\_Messages Package

```

1  with Incident_Defs,
2      System,
3      System_Defs;
4
5  package Print_Cmd_Messages is
6      --
7      -- Function:
8      --   Define messages used by Print_cmd_ex
9      --   All messages defined use a module ID of 0.
10
11     print_msg_pathname: constant System_Defs.text_AD :=
12         new System_Defs.text'(
13             32,32,"/examples/msg/print_cmd_messages");
14     -- AD to pathname of message file, bound to
15     -- "msg_obj", following.
16     -- *This will go away when "pragma bind" changes.*
17
18     msg_obj: constant System.untyped_word :=
19         System.null_word;
20     pragma bind(msg_obj,
21         "example messages.print_msg_pathname");
22     -- Message object for incident codes in
23     -- example programs, bound to above
24     -- "message_file_pathname".
25     --
26     -- *When the resident compiler and linker are*
27     -- *ready, this pragma will become:*
28     -- | pragma bind(msg_obj,
29     -- |     "/examples/msg/print_cmd_messages");
30
31
32     no_print_device_code:
33         constant Incident_Defs.incident_code :=
34             (0, 1, Incident_Defs.information, msg_obj);
35     --
36     ---M* store :module=0 :number=1 \
37     ---M*       :msg_name=name_space_created_code \
38     ---M*       :short = \
39     ---M*       "Print Device $pl<on> does not exist."
40
41     units_not_supported_code:
42         constant Incident_Defs.incident_code :=
43             (0, 2, Incident_Defs.information, msg_obj);
44     --
45     ---M* store :module=0 :number=2 \
46     ---M*       :msg_name=units_not_supported_code \
47     ---M*       :short = \
48     ---M*       "Unit $pl<on> not supported."
49
50 end Print_Cmd_Messages;

```

## X-A.4.15 Record\_AM\_Ex Package Specification

```

1  with Device_Defs,
2      Employee_Filing_Ex,
3      Record_AM,
4      System,
5      System_Defs;
6
7  package Record_AM_Ex is
8      --
9      -- Function:
10     -- This package contains the example subprograms
11     -- for using the Record_AM package.
12     --
13     -- History:
14     -- 08-10-87, Paul Schwabe: initial version.
15     -- 11-23-87, Paul Schwabe: revision.
16     --
17     pragma external;
18
19     function Get_record_ID(
20         opened_file: Device_Defs.opened_device)
21         -- An opened device, opened for input on an
22         -- employee file.
23         return Record_AM.record_ID;
24     --
25     -- Operation:
26     -- Returns a record ID from the operation status
27     -- information. The record ID can be used in
28     -- subsequent retrieval operations to maximize
29     -- access time to the specified record.
30
31
32     function Get_record_number(
33         opened_file: Device_Defs.opened_device)
34         -- An opened device, opened for input on an
35         -- employee file.
36         return System.ordinal;
37     --
38     -- Operation:
39     -- Returns a record number from the operation
40     -- status information. The record number can be
41     -- used in subsequent retrieval operations for
42     -- relative files.
43
44
45     procedure Insert_record(
46         opened_file: Device_Defs.opened_device);
47         -- An opened device, opened for input on an
48         -- employee file.
49     --
50     -- Function:
51     -- Inserts a record into a structured file.
52     --
53     -- Applicable for any file organization.
54     -- Position of the inserted record in the file
55     -- is determined by the system. The new record
56     -- is automatically assigned a record ID.
57
58
59     procedure Read_random_by_record_ID(
60         opened_file: Device_Defs.opened_device;
61         rec_id:      Record_AM.record_ID);
62         -- An opened device, opened for input on an
63         -- employee file.
64     --
65     -- Function:
66     -- Reads a record randomly using a previously
67     -- retrieved record ID from the operation status
68     -- information. This is the fastest possible
69     -- random access to a record using any
70     -- structured file organization.
71
72
73     procedure Read_random_by_record_number(
74         opened_file: Device_Defs.opened_device;

```

PRELIMINARY

```

75     rec_number: System.ordinal);
76     -- An opened device, opened for input on an
77     -- employee file.
78     --
79     -- Function:
80     -- Reads a record randomly from a relative file
81     -- using a previously retrieved record ID from
82     -- the operation status information. Record
83     -- numbers are only applicable for relative
84     -- files.
85
86
87
88 procedure Read_next_simple_index(
89     opened_file: Device_Defs.opened_device);
90     -- An opened device, opened for input on an employee
91     -- file.
92     --
93     -- Function:
94     -- Reads a range of records in the "Dept" index.
95     --
96     -- Positions to the beginning of the range and
97     -- reads successive records until the end. The
98     -- start value is to the left of the index.
99     -- This composite index is read by ascending key
100    -- values starting at the lowest key value in
101    -- the range.
102    --
103    -- Dept (asc) A B ... X Y
104    --           ---> EOF
105    -- The position_modifier value is Record_AM.next
106    --
107    -- Notes:
108    -- This function replaces any previous key range
109    -- and changes the file's record pointer.
110    --
111    -- The "Dept" index is ascending on department.
112    -- Returns all employee records for the
113    -- departments in the specified range.
114    --
115
116
117
118 procedure Read_prior_simple_index(
119     opened_file: Device_Defs.opened_device);
120     -- An opened device, opened for input on an
121     -- employee file.
122     --
123     -- Function:
124     -- Reads a range of records in the "Dept" index.
125     --
126     -- Positions to the end of the range and reads
127     -- successive records until the beginning. The
128     -- start value is to the right of the index.
129     -- This composite index is read by ascending key
130     -- values starting at the lowest key value in
131     -- the range.
132     --
133     -- Dept (asc) A B ... X Y
134     --           EOF <---
135     -- The position_modifier value is
136     -- Record_AM.prior
137     --
138     -- Notes:
139     -- This function replaces any previous key range
140     -- and changes the file's record pointer.
141     --
142     -- The "Dept" index is ascending on department.
143     -- Returns all employee records for the
144     -- departments in the specified range.
145     --
146
147
148
149 procedure Read_duplicates(
150     opened_file: Device_Defs.opened_device);
151     -- An opened device, opened for input on an

```

```

152     -- employee file.
153     --
154     -- Function:
155     -- Reads a duplicate records in the specified
156     -- "Dept" index.
157     --
158     -- Positions to the specified record and reads
159     -- all duplicates until the end.
160     --
161     -- Dept (asc)  A A ... A A
162     --             ----> EOF
163     -- The position_modifier value is Record_AM.next
164     --
165     -- Notes:
166     -- This function replaces any previous key range
167     -- and changes the file's record pointer.
168     --
169     -- The "Dept" index is ascending on department.
170     -- Returns all employee records for the
171     -- departments in the specified range.
172     --
173     -- The range contains employees in "Accounting"
174     -- through "Marketing".
175     --
176     -- If the "Dept" index were specified as
177     -- non-unique, returns duplicate records for a
178     -- particular "Dept" key value. For example,
179     -- one record might contain fields on
180     -- management, cost control, and history. A
181     -- second record might simply hold text.
182     --
183
184
185     procedure Delete_records_sequential(
186         opened_file: Device_Defs.opened_device);
187     -- An opened device, opened for input on an
188     -- employee file.
189     --
190     -- Function:
191     -- Deletes a range of records using the
192     -- department name as a key. This example shows
193     -- that a Read or Set_position is not required
194     -- to preface each Delete. The current record
195     -- pointer advances after each Delete.
196
197     procedure Read_and_update_by_key(
198         opened_file: Device_Defs.opened_device);
199     -- An opened device, opened for input on an
200     -- employee file.
201     --
202     -- Function:
203     -- Updates a record within a range of records.
204     -- This example shows that the current record
205     -- pointer does NOT advance after the
206     -- Update_by_key.
207
208
209
210     procedure Read_records_reverse_sequential(
211         opened_file: Device_Defs.opened_device);
212     -- An opened device, opened for input on an
213     -- employee file.
214     --
215     -- Function:
216     -- Reads all records in a reverse sequence.
217     -- Shows Shows physical-sequential access.
218     --
219     -- Positions to the end of the sequence and
220     -- reads successive records until the beginning.
221     -- After each read, the current record pointer
222     -- is positioned to the prior record.
223
224
225     procedure Read_records_sequential(
226         opened_file: Device_Defs.opened_device);
227     -- An opened device, opened for input on an
228     -- employee file.

```

```

229  --
230  -- Function:
231  -- Reads all records in a sequence. Shows
232  -- physical-sequential access.
233  --
234  -- Positions to the start of the sequence and
235  -- reads successive records until the end.
236  --
237  -- Notes:
238  -- Advances the file's current record pointer
239  -- forward after each read.
240  --
241
242
243  procedure Read_and_delete_records(
244      opened_file: Device_Defs.opened_device);
245      -- An opened device, opened for input on an
246      -- employee file.
247  --
248  -- Function:
249  -- Reads and deletes selected records in a
250  -- sequence.
251  --
252  -- Positions to the beginning of the sequence
253  -- and reads successive records until the end.
254  -- After each read, a record is checked and then
255  -- deleted if it satisfies the specified
256  -- conditions. The current record pointer is
257  -- positioned to the next record after the
258  -- deleted record.
259
260
261  procedure Read_and_update_records(
262      opened_file: Device_Defs.opened_device);
263      -- An opened device, opened for input on an
264      -- employee file.
265  --
266  -- Function:
267  -- Reads and updates records in a sequence.
268  --
269  -- Positions to the beginning of the sequence
270  -- and reads successive records until the end.
271  -- After each read, the current record pointer
272  -- is positioned to the next record.
273
274
275  procedure Update_salary_example(
276      T2_opened_file: Device_Defs.opened_device);
277      -- An opened device for transaction T1,
278      -- opened for input on an employee file.
279  --
280  -- Function:
281  -- Does an index-random update of a record in an
282  -- indexed relative file.
283  --
284  -- The Update_salary_example procedure starts
285  -- transaction T2 to double an employee's
286  -- salary. If transaction T2 aborts, then the
287  -- update is rolled back.
288  --
289  -- Notes:
290  -- The example relative file is created with the
291  -- following parameters:
292  --     xm locking => true
293  --     short_term_logging => true
294  -- The example index (with a key built on
295  -- "employee ID") is built with
296  -- phantom_protected => false.
297  --
298
299  end Record_AM_Ex;
300

```

## X-A.4.16 Record\_AM\_Ex Package Body

```

1  with Device_Defs,
2      Employee_Filing_Ex,
3      File_Admin,
4      File_Defs,
5      Record_AM,
6      System,
7      System_Defs,
8      Transaction_Mgt;
9
10 -- For Importing operations.
11 use Employee_Filing_Ex,
12     System,
13     System_Defs;
14
15 package body Record_AM_Ex is
16 --
17 -- Logic:
18 -- Provides the implementation code for the
19 -- Record_AM examples.
20 --
21 --
22 --
23 -- CONSTANT AND VARIABLE DECLARATIONS
24 --
25
26 buffer: string(1 .. integer(Employee_Filing_EX.max_rec_size));
27 -- Buffer is large enough to hold any employee
28 -- record.
29
30 current_record_addr: constant System.address :=
31     buffer'address;
32 current_record_va: constant Employee_Filing_EX.
33     employee_record_va := Employee_Filing_EX.
34     Employee_record_va_from_va(
35         current_record_addr);
36
37 pay_raise: constant float := 2.0;
38
39 bytes_read: System.ordinal;
40 -- Number of bytes in current record.
41
42 read_status_va: Record_AM.operation_status_va :=
43     new Record_AM.operation_status_record;
44 -- Virtual address of status record.
45
46 -- Employee name constant.
47 employee: constant Employee_Filing_EX.person_name :=
48     (Employee_Filing_EX.max_text_length,
49     10,
50     "Einstein, Albert      ");
51
52 --
53 -- SUBPROGRAM DECLARATIONS
54 --
55
56 function Get_record_ID(
57     opened_file: Device_Defs.opened_device)
58 -- An opened device, opened for input on an
59 -- employee file.
60 return Record_AM.record_ID
61 -- Note:
62 -- Records in any structured file can have
63 -- record IDs, but only records in relative
64 -- files can have record numbers!
65 is
66 begin
67     Record_AM.Ops.Set_position(
68         opened_dev => opened_file,
69         where => Record_AM.record_specifier(
70             type_of_specifier => Record_AM.first)'(
71             type_of_specifier => Record_AM.first));
72     loop
73         bytes_read := Record_AM.Ops.Read(
74             opened_dev => opened_file,

```

```

75         buffer_VA => buffer'address,
76         length    => buffer'length,
77         status    => read_status VA);
78     if current_record_VA.name = employee then
79         RETURN read_status_VA.rec_ID;
80
81     end if;
82 end loop;
83
84 exception
85     when Device_Defs.end_of_file =>
86         RETURN Record_AM.null_record_ID;
87
88 end Get_record_ID;
89
90
91 function Get_record_number(
92     opened_file: Device_Defs.opened_device)
93     -- An opened device, opened for input on an
94     -- employee file.
95     return System.ordinal
96     --
97 is
98 begin
99     Record_AM.Ops.Set_position(
100     opened_dev => opened_file,
101     where => Record_AM.record_specifier(
102     type_of_specifier => Record_AM.first)'(
103     type_of_specifier => Record_AM.first));
104     loop
105     bytes_read := Record_AM.Ops.Read(
106     opened_dev => opened_file,
107     buffer_VA => buffer'address,
108     length    => buffer'length,
109     status    => read_status VA);
110     if current_record_VA.name = employee then
111         RETURN read_status_VA.rec_num;
112
113     end if;
114     end loop;
115
116 exception
117     when Device_Defs.end_of_file =>
118         RETURN 0;
119
120 end Get_record_number;
121
122
123
124 procedure Insert_record(
125     opened_file: Device_Defs.opened_device)
126     -- An opened device, opened for input on an
127     -- employee file.
128     --
129 is
130 begin
131     -- Obtain the new record from
132     -- somewhere (form or file)
133     -- and load the record buffer.
134
135     Record_AM.Ops.Insert(
136     opened_dev => opened_file,
137     buffer_VA => buffer'address,
138     length    => System.ordinal(
139     Employee_Filing_EX.max_rec_size));
140
141 end Insert_record;
142
143
144 procedure Read_random_by_record_ID(
145     opened_file: Device_Defs.opened_device;
146     rec_ID:      Record_AM.record_ID)
147     -- An opened device, opened for input on an
148     -- employee file.
149 is
150 begin
151     Record_AM.Ops.Set_position(

```

```

152         opened_file,
153         where => Record_AM.record_specifier(
154             type_of_specifier => Record_AM.id)' (
155             type_of_specifier => Record_AM.id,
156             rec_id => rec_ID));
157
158     bytes_read := Record_AM.Ops.Read(
159         opened_dev => opened_file,
160         buffer_VA => buffer'address,
161         length => buffer'length);
162
163 end Read_random_by_record_ID;
164
165
166 procedure Read_random_by_record_number(
167     opened_file: Device_Defs.opened_device;
168     rec_number: System.ordinal)
169     -- An opened device, opened for input on an
170     -- employee file.
171 is
172 begin
173     Record_AM.Ops.Set_position(
174         opened_file,
175         where => Record_AM.record_specifier(
176             type_of_specifier => Record_AM.number)' (
177             type_of_specifier => Record_AM.number,
178             rec_num => rec_number));
179     bytes_read := Record_AM.Ops.Read(
180         opened_dev => opened_file,
181         buffer_VA => buffer'address,
182         length => buffer'length);
183
184 end Read_random_by_record_number;
185
186
187 procedure Read_next_simple_index(
188     opened_file: Device_Defs.opened_device)
189     -- An opened device, opened for input on an
190     -- employee file.
191     --
192 is
193     start_key_value: constant Employee_Filing_EX.
194     dept_key_buffer := (dept => 100);
195     -- Lowest department for
196     -- ascending key field.
197
198     start_key_descr: constant
199     Record_AM.key_value_descr := (
200     start_key_value'address,
201     start_key_value'size / 8);
202
203     stop_key_value: constant Employee_Filing_EX.
204     dept_key_buffer := (dept => 500);
205     -- High end for ascending key field.
206
207     stop_key_descr: constant
208     Record_AM.key_value_descr := (
209     stop_key_value'address,
210     stop_key_value'size / 8);
211 begin
212     Record_AM.Keyed_Ops.Set_key_range(
213         opened_dev => opened_file,
214         index =>
215         Employee_Filing_EX.dept_index_name,
216         select_range => (
217             start_comparison => Record_AM.exclusive,
218             start_value => start_key_descr,
219             stop_comparison => Record_AM.inclusive,
220             stop_value => stop_key_descr));
221
222     loop
223         bytes_read := Record_AM.Ops.Read(
224             opened_dev => opened_file,
225             modifier => Record_AM.next,
226             -- Next is normally defaulted.
227             buffer_VA => buffer'address,
228             length => buffer'length);

```

```

229
230     -- DO ANY NEEDED PROCESSING HERE.
231
232     end loop;
233
234 exception
235     when Device_Defs.end_of_file =>
236         null;
237 end Read_next_simple_index;
238
239
240 procedure Read_prior_simple_index(
241     opened_file: Device_Defs.opened_device)
242     -- An opened device, opened for input on an
243     -- employee file.
244     --
245 is
246     start_key_value: constant Employee_Filing_EX.
247     dept_key_buffer := (dept => 500);
248     -- High end for ascending key field.
249
250     start_key_descr: constant
251     Record_AM.key_value_descr := (
252     start_key_value'address,
253     start_key_value'size / 8);
254
255     stop_key_value: constant Employee_Filing_EX.
256     dept_key_buffer := (dept => 100);
257     -- Lowest department for
258     -- ascending key field.
259
260     stop_key_descr: constant
261     Record_AM.key_value_descr := (
262     stop_key_value'address,
263     stop_key_value'size / 8);
264
265 begin
266     Record_AM.Keyed_Ops.Set_key_range(
267     opened_dev => opened_file,
268     index      =>
269     Employee_Filing_EX.dept_index_name,
270     select_range => (
271     start_comparison => Record_AM.exclusive,
272     start_value      => start_key_descr,
273     stop_comparison  => Record_AM.inclusive,
274     stop_value       => stop_key_descr));
275
276     loop
277     bytes_read := Record_AM.Ops.Read(
278     opened_dev => opened_file,
279     modifier   => Record_AM.prior,
280     -- Sets read modifier to prior.
281     buffer_VA  => buffer'address,
282     length     => buffer'length);
283
284     -- DO ANY NEEDED PROCESSING HERE.
285
286     end loop;
287 exception
288     when Device_Defs.end_of_file =>
289     null;
290 end Read_prior_simple_index;
291
292
293
294 procedure Read_duplicates(
295     opened_file: Device_Defs.opened_device)
296     -- An opened device, opened for input on an
297     -- employee file.
298 is
299     start_key_value: constant Employee_Filing_EX.
300     dept_key_buffer := (dept => 305);
301     -- Start value for duplicate
302     -- key field.
303
304
305     start_key_descr: constant Record_AM.

```

```

306     key_value_descr := (
307         start_key_value'address,
308         start_key_value'size / 8);
309
310 stop_key_value: constant Employee_Filing_EX.
311 dept_key_buffer := (dept => 305);
312 -- Stop value for duplicate
313 -- key field.
314
315
316 stop_key_descr: constant Record_AM.
317     key_value_descr := (
318         stop_key_value'address,
319         stop_key_value'size / 8);
320
321 begin
322     Record_AM.Keyed_Ops.Set_key_range(
323         opened_dev => opened_file,
324         index      => Employee_Filing_EX.
325             dept_index_name,
326         select_range => (
327             start_comparison => Record_AM.inclusive,
328             start_value      => start_key_descr,
329             stop_comparison  => Record_AM.inclusive,
330             stop_value       => stop_key_descr));
331
332 loop
333     bytes_read := Record_AM.Ops.Read(
334         opened_dev => opened_file,
335         modifier   => Record_AM.next,
336         -- Normally defaulted.
337         buffer_VA  => buffer'address,
338         length     => buffer'length);
339
340     -- DO ANY PROCESSING HERE
341
342 end loop;
343 exception
344 when Device_Defs.end_of_file =>
345     null;
346
347 end Read_duplicates;
348
349 procedure Delete_records_sequential(
350     opened_file: Device_Defs.opened_device)
351 -- An opened device, opened for input on an
352 -- employee file.
353 -- Logic:
354 -- Do a Set_key_range for a range of departments
355 -- to delete. Set up a loop for the deletes with
356 -- the position_modifier = current. (Key point: a
357 -- Read or Set_position is not required to
358 -- preface each Delete in the loop. The current
359 -- record pointer advances after each Delete)
360 is
361 start_key_value: constant Employee_Filing_EX.
362     dept_key_buffer := (dept => 150);
363     -- Low end for ascending key field.
364
365 start_key_descr: constant Record_AM.
366     key_value_descr := (
367         start_key_value'address,
368         start_key_value'size / 8);
369
370 stop_key_value: constant Employee_Filing_EX.
371     dept_key_buffer := (dept => 200);
372     -- High end for ascending
373     -- key field.
374
375 stop_key_descr: constant Record_AM.
376     key_value_descr := (
377         stop_key_value'address,
378         stop_key_value'size / 8);
379
380 begin
381     Record_AM.Keyed_Ops.Set_key_range(
382         opened_dev => opened_file,
383         index      => Employee_Filing_EX.
384             dept_index_name,

```

```

383     select_range => (
384         start_comparison => Record_AM.inclusive,
385         start_value      => start_key_descr,
386         stop_comparison  => Record_AM.inclusive,
387         stop_value       => stop_key_descr));
388 loop
389     -- CRP is updated after each delete
390     -- (no read is necessary to preface
391     -- the Delete).
392     Record_AM.Ops.Delete(
393         opened_dev => opened_file,
394         modifier   => Record_AM.current,
395         -- Normally defaulted.
396         timeout    => Record_AM.wait_forever,
397         status     => null);
398
399 end loop;
400
401 exception
402     when Device_Defs.end_of_file =>
403         null;
404
405 end Delete_records_sequential;
406
407 procedure Read_and_update_by_key(
408     opened_file: Device_Defs.opened_device)
409     -- An opened device, opened for input on an
410     -- employee file.
411
412     --
413     -- Logic:
414     -- Do a Set_key_range for a range of departments
415     -- to update. Set up a read loop using
416     -- position_modifier = next. Do a comparison
417     -- to trap a record to update. When rec_in =
418     -- record_of_interest, do an Update_by_key.
419     -- (Key point: the current record pointer does
420     -- NOT advance after the Update_by_key.)
421 is
422     start_key_value: constant Employee_Filing_EX.
423     dept_key_buffer := (dept => 100);
424     -- Lowest dept for ascending key field.
425
426     start_key_descr: constant Record_AM.
427     key_value_descr := (
428         start_key_value'address,
429         start_key_value'size / 8);
430
431     stop_key_value: constant Employee_Filing_EX.
432     dept_key_buffer := (dept => 200);
433     -- High end for ascending
434     -- key field.
435
436     stop_key_descr: constant Record_AM.
437     key_value_descr := (
438         stop_key_value'address,
439         stop_key_value'size / 8);
440 begin
441     Record_AM.Keyed_Ops.Set_key_range(
442         opened_dev => opened_file,
443         index      => Employee_Filing_EX.dept_index_name,
444         select_range => (
445             start_comparison => Record_AM.inclusive,
446             start_value      => start_key_descr,
447             stop_comparison  => Record_AM.inclusive,
448             stop_value       => stop_key_descr));
449 loop
450     bytes_read := Record_AM.Ops.Read(
451         opened_dev => opened_file,
452         modifier   => Record_AM.next,
453         buffer_VA  => buffer'address,
454         length     => buffer'length);
455
456     if current_record_VA.dept = 175 then
457         -- CRP does not advance to next record
458         -- after the Update_by_key (it advances on
459         -- next read).

```

```

460         Record_AM.Keyed_Ops.Update_by_key(
461             opened_dev => opened_file,
462             buffer_VA  => buffer'address,
463             length     => buffer'length,
464             index      => Employee_Filing_EX.
465                 dept_index_name);
466             -- Employee ID index (hashed).
467         end if;
468     end loop;
469 exception
470     when Device_Defs.end_of_file =>
471         null;
472 end Read_and_update_by_key;
473
474
475 procedure Read_records_reverse_sequential(
476     opened_file: Device_Defs.opened_device)
477     -- An opened device, opened for input on an
478     -- employee file.
479     --
480 is
481 begin
482     Record_AM.Ops.Set_position(
483         opened_dev => opened_file,
484         where      => Record_AM.record_specifier(
485             type_of_specifier => Record_AM.last)'(
486                 type_of_specifier => Record_AM.last));
487     -- Positions current record pointer
488     -- to last record in file.
489     loop
490         bytes_read := Record_AM.Ops.Read(
491             opened_dev => opened_file,
492             modifier   => Record_AM.prior,
493             buffer_VA  => buffer'address,
494             length     => buffer'length);
495
496         -- DO ANY NEEDED PROCESSING HERE.
497
498     end loop;
499
500
501 exception
502     when Device_Defs.end_of_file =>
503         null;
504 end Read_records_reverse_sequential;
505
506
507 procedure Read_records_sequential(
508     opened_file: Device_Defs.opened_device)
509     -- An opened device, opened for input on an
510     -- employee file.
511     --
512 is
513 begin
514     Record_AM.Ops.Set_position(
515         opened_dev => opened_file,
516         where      => Record_AM.record_specifier(
517             type_of_specifier => Record_AM.first)'(
518                 type_of_specifier => Record_AM.first));
519     loop
520         bytes_read := Record_AM.Ops.Read(
521             opened_dev => opened_file,
522             buffer_VA  => buffer'address,
523             length     => buffer'length);
524
525         -- DO ANY NEEDED PROCESSING HERE.
526
527     end loop;
528
529 exception
530     when Device_Defs.end_of_file =>
531         null;
532
533 end Read_records_sequential;
534
535
536 procedure Read_and_delete_records(

```

```

537     opened_file: Device_Defs.opened_device)
538     -- An opened device, opened for input on an
539     -- employee file.
540 is
541 begin
542     Record_AM.Ops.Set_position(
543     opened_dev => opened_file,
544     where => Record_AM.record_specifier(
545     type_of_specifier => Record_AM.first)'(
546     type_of_specifier => Record_AM.first));
547     loop
548     bytes_read := Record_AM.Ops.Read(
549     opened_dev => opened_file,
550     buffer_VA => buffer'address,
551     length => buffer'length);
552
553     if current_record_VA.dept = 175 then
554     Record_AM.Keyed_Ops.Delete_by_key(
555     opened_dev => opened_file,
556     index => Employee_Filing_Ex.
557     dept_index_name);
558
559     end if;
560
561     end loop;
562
563 exception
564
565     when Device_Defs.end_of_file =>
566     null;
567
568 end Read_and_delete_records;
569
570
571 procedure Read_and_update_records(
572     opened_file: Device_Defs.opened_device)
573     -- An opened device, opened for input on an
574     -- employee file.
575     --
576 is
577 begin
578     Record_AM.Ops.Set_position(opened_file,
579     where => Record_AM.record_specifier(
580     type_of_specifier => Record_AM.first)'(
581     type_of_specifier => Record_AM.first));
582     loop
583     bytes_read := Record_AM.Ops.Read(
584     opened_dev => opened_file,
585     buffer_VA => buffer'address,
586     length => buffer'length);
587
588     current_record_VA.salary :=
589     pay_raise * current_record_VA.salary;
590
591     Record_AM.Ops.Update(
592     opened_dev => opened_file,
593     buffer_VA => buffer'address,
594     length => buffer'length);
595     end loop;
596
597
598 exception
599     when Device_Defs.end_of_file =>
600     null;
601
602 end Read_and_update_records;
603
604
605 procedure Update_salary_example(
606     T2_opened_file: Device_Defs.opened_device)
607     -- An opened device for transaction T1, opened
608     -- for input on an employee file.
609     --
610 is
611 begin
612     Transaction_Mgt.Start_transaction;
613     -- Started on behalf of transaction T2, the
614     -- updater.

```

```
614
615      -- The record must have been positioned to by a
616      -- previous read, otherwise a
617      -- Record_AM.key_value_descr must be specified.
618      -- No key range is necessary. The current record
619      -- pointer is not affected.
620
621      current_record VA.salary :=
622          pay_raise * current_record_VA.salary;
623
624      -- Default is the current record.
625      Record_AM.Keyed_Ops.Update_by key(
626          opened_dev => T2_opened_file,
627          buffer_VA  => buffer'address,
628          length     => buffer'length,
629          index      => Employee_Filing_EX.
630                  dept_salary_index_name);
631          -- Employee ID index.
632
633      exception
634          when Device_Defs.end_of_file =>
635              Transaction_Mgt.Commit_transaction;
636
637          when others =>
638              Transaction_Mgt.Abort_transaction;
639
640      end Update_salary_example;
641
642
643 end Record_AM_Ex;
```

## X-A.4.17 simple\_editor\_cmd\_ex Procedure

```
1 with Command_Handler,
2   Device_Defs,
3   Simple_Editor_Ex;
4
5 -----
6 --                               SIMPLE EDITOR                               --
7 -----
8 procedure Simple_editor_cmd_ex
9   --
10  -- Function:
11  -- This procedure implements a simple text
12  -- editor for the purpose of demonstrating certain
13  -- aspects of the Character Display Access Method.
14  --
15  -- Command Definition:
16  -- The command has the form:
17  --
18  --     simple_editor_cmd_ex :name=<symbolic_name(1..80)>
19  --
20  --*D*
21  --*D* manage.commands
22  --*D*   create.invocation_command
23  --*D*
24  --*D*   define.argument name \
25  --*D*     :type = string
26  --*D*     set.lexical_class symbolic_name
27  --*D*     set.maximum_length 80
28  --*D*     set.mandatory
29  --*D*   end
30  --*D* end
31  --*D* exit
32  --
33  -- End of Header
34
35 is
36
37   opened_cmd: Device_Defs.opened_device;
38
39 begin
40
41   -- Get command arguments:
42   --
43   opened_cmd := Command_Handler.
44     Open_invocation_command_processing;
45
46   Command_Handler.Get_string(
47     cmd_odo   => opened_cmd,
48     arg_number => 1,
49     arg_value  => Simple_Editor_Ex.file_name);
50
51   Command_Handler.Close(opened_cmd);
52
53   -- NOTE: allocation is done here rather than at the
54   -- declaration due to the exception
55   -- "Object has no representation" being raised
56   -- if the Get_object_size is called before the object
57   -- is accessed
58   Simple_Editor_Ex.edit_buffer :=
59     new Simple_Editor_Ex.edit_buffer object'(
60       max_lines => Simple_Editor_Ex.resize_lines,
61       num_lines => 0,
62       lines     => (others => (others => ASCII.NUL)));
63
64   Simple_Editor_Ex.Read_file;
65
66   Simple_Editor_Ex.Make_window;
67
68   Simple_Editor_Ex.Handle_input;
69
70 end Simple_editor_cmd_ex;
```

## X-A.4.18 Simple\_Editor\_Ex Package Specification

```

1  with Incident_Defs,
2      System_Defs,
3      System,
4      Terminal_Defs;
5
6  package Simple_Editor_Ex is
7      --
8      -- Function:
9      -- This package implements procedures to support a
10     -- simple text editor for the purpose of demonstrating
11     -- certain aspects of the Character Display Access Method.
12     --
13     -- The editor has the following attributes:
14     --
15     -- 1. The file is read into an array of lines of characters.
16     --    Each line in 80 characters (screen width)
17     --
18     -- 2. If the file does not exist it will be created.
19     --
20     -- 3. The array will expand to any size file.
21     --
22     -- 4. The array is null-filled before the
23     --    file is read in. (Character_Display_AM
24     --    will ignore the nulls)
25     --
26     -- 4. Each line in the file is read into
27     --    one row in the array. Long lines (>80) will be
28     --    preserved but they cannot be altered by the editor.
29     --
30     -- 5. The frame buffer is 24 by 80 (screen size).
31     --
32     -- 6. If changes have been made since the last save
33     --    it will prompt the user if ok to exit.
34     --
35     -- 7. The bell will ring for illegal commands.
36     --
37     -- The operations available in the editor are:
38     --
39     -- * Move forward      (Control F)
40     -- * Move backward    (Control B)
41     -- * Move up          (Control P)
42     -- * Move down        (Control N)
43     -- * Page up          (Control U)
44     -- * Page down        (Control V)
45     -- * Delete forward   (Control D)
46     -- * Delete backward  (Control H)
47     -- * Insert text
48     -- * Save file        (Control W)
49     -- * Quit editor      (Control C)
50     --
51     -- History:
52     -- 11/??/86, G. Taylor   : Initial version
53     -- 12/??/87, E. Sassone : Revised version
54     -- 12/19/87, G. Taylor  : Added tagged comments
55     -- 06/15/88, E. Sassone : working version
56     --
57     -- Exception Codes:
58     --
59     new_file_code: constant Incident_Defs.incident_code := (
60         module      => 0,
61         number      => 1,
62         severity    => Incident_Defs.information,
63         message_object => System.null_word);
64
65     not_saved_code: constant Incident_Defs.incident_code := (
66         module      => 0,
67         number      => 2,
68         severity    => Incident_Defs.warning,
69         message_object => System.null_word);
70
71     no_long_lines_code: constant Incident_Defs.incident_code := (
72         module      => 0,
73         number      => 3,
74         severity    => Incident_Defs.information,

```

```

75     message_object => System.null_word);
76
77 editor_error_code: constant Incident_Defs.incident_code := (
78     module      => 0,
79     number      => 4,
80     severity    => Incident_Defs.error,
81     message_object => System.null_word);
82
83 -- Exceptions:
84 --
85 ---*D* manage.messages
86 --
87 no_access: exception;
88 ---*D* store :module=0 :number=1 \
89 ---*D*   :msg_name=new_file_code \
90 ---*D*   :short = \
91 ---*D*   "$pl<pathname> is a new file."
92 --
93 ---*D* store :module=0 :number=2 \
94 ---*D*   :msg_name=not_saved_code \
95 ---*D*   :short = \
96 ---*D*   "Changes have not been saved.  Exit anyway? "
97 --
98 ---*D* store :module=0 :number=3 \
99 ---*D*   :msg_name=no_long_lines_code \
100 ---*D*   :short = \
101 ---*D*   "Changes to long lines NYI"
102 --
103 editor_error: exception;
104 ---*D* store :module=0 :number=4 \
105 ---*D*   :msg_name=editor_error_code \
106 ---*D*   :short = \
107 ---*D*   "Editor_error - please save your file and quit"
108 --
109 -- End of Header
110
111 -----
112 --                          CONSTANTS                          --
113 -----
114 origin:                    constant Terminal_Defs.point_info := (1, 1);
115   -- frame buffer origin
116
117 first_row:                  constant integer := 1;
118
119 first_column:               constant integer := 1;
120 last_column:                constant integer := 80;
121
122 frame_rows:                 constant integer := 24;
123   -- screen size
124
125 preferred_window_rows:     constant integer := 10;
126   -- initial window size
127
128 linear_buf_size:           constant := 4_096;
129   -- size of read/write buffer
130
131 resize_lines:               constant := 100;
132   -- number of lines to add for resizing edit buffer
133   -- object
134
135 -----
136 --                          TYPES                          --
137 -----
138 subtype row_delta is integer range -1 .. 1;
139 subtype row_range is positive;
140 subtype column_range is integer range 1 .. last_column;
141
142 -- position in edit buffer
143 type cursor_location is
144   record
145     row:      row_range;
146     column:   column_range;
147   end record;
148
149 -- edit buffer
150 type line is array (column_range) of character;
151 type edit_array is array (integer range <>) of line;

```

```

152 type edit_buffer_object(
153     max_lines: integer) is
154     record
155         num_lines: integer := 0;
156         lines:     edit_array (first_row .. max_lines);
157     end record;
158
159 type edit_buffer_AD is access edit_buffer_object;
160 pragma access_kind(edit_buffer_AD, AD);
161
162 -- for input of command and insertions chars
163 type char_array      is array (1 .. 120) of character;
164 type char_array_AD  is access char_array;
165 pragma access_kind(char_array_AD, AD);
166
167 -----
168 --                VARIABLES                --
169 -----
170 file_name: System_Defs.text(Incident_Defs.txt_length);
171 edit_buffer: edit_buffer_AD;
172
173 -----
174 --                PROCEDURES              --
175 -----
176
177 function Move_page(
178     direction: row_delta)
179     return boolean;    -- operation successful
180     --
181     -- Function:
182     -- Move up or down by the size of the view
183     --
184
185
186 function Move_up
187     return boolean;    -- operation successful
188     --
189     -- Function:
190     -- Moves the cursor up one line, but not
191     -- beyond the beginning of the file.
192     --
193
194
195 function Move_down
196     return boolean;    -- operation successful
197     --
198     -- Function:
199     -- Moves the cursor down one line, but not
200     -- beyond the end of the file.
201     --
202
203
204 function Move_forward
205     return boolean;    -- operation successful
206     --
207     -- Function:
208     -- Moves the cursor forward one character
209     -- but not beyond the end of the line.
210     --
211
212
213 function Move_back
214     return boolean;    -- operation successful
215     --
216     -- Function:
217     -- Moves the cursor backward one character, but not
218     -- beyond the beginning of the line.
219     --
220
221
222 function Delete_forward
223     return boolean;    -- operation successful
224     --
225     -- Function:
226     -- Deletes the character at the cursor's current
227     -- position. Cursor position in unchanged.
228     --

```

```

229
230
231 function Delete_backward
232     return boolean;      -- operation successful
233     --
234     -- Function:
235     --   Deletes the character to the left of the cursor,
236     --   but not beyond the beginning of the line.
237     --
238
239
240 function Insert(
241     insert_char: character)
242     return boolean;      -- operation successful
243     --
244     -- Function:
245     --   Insert printable characters to the left of the
246     --   cursor.
247     --
248
249
250 procedure Save_file;
251     --
252     -- Function:
253     --   Writes the file from the edit buffer.
254     --
255
256
257 procedure Quit_editor;
258     --
259     -- Function:
260     --   Exits the editor if changes have been made
261     --   since the last save it will ask the user
262     --   whether the unsaved changes should be saved or
263     --   not. Returns cursor to old window.
264     --
265
266
267 procedure Read_file;
268     --
269     -- Function:
270     --   Reads the sections of the input file into the
271     --   edit buffer.
272     --
273
274
275 procedure Make_window;
276     --
277     -- Function:
278     --   Creates a new window for editing.
279     --
280
281
282 procedure Handle_input;
283     --
284     -- Function:
285     --   Loops waiting for editor keyboard and menu input.
286     --
287
288 procedure Key_input(
289     key: character);
290     --
291     -- Function:
292     --   Calls the appropriate procedure based on the
293     --   key input.
294     --
295
296 end Simple_Editor_Ex;

```

## X-A.4.19 Simple\_Editor\_Ex Package Body

```

1  with Byte_Stream_AM,
2      Character_Display_AM,
3      Device_Defs,
4      Directory_Mgt,
5      File_Defs,
6      Incident_Defs,
7      Long_Integer_Defs,
8      Message_Services,
9      Object_Mgt,
10     Process_Mgt,
11     Process_Mgt_Types,
12     Simple_File_Admin,
13     System,
14     System_Defs,
15     Terminal_Defs,
16     Text_Mgt,
17     Window_Services;
18
19 package body Simple_Editor_Ex is
20
21     -----
22     --                VARIABLES                --
23     -----
24     -- position of frame buffer in edit_buffer
25     frame_begin:      row_range := first_row;
26     frame_end:        row_range := frame_rows;
27
28     edit_buf_pos:     cursor_location := (first_row, first_column);
29
30     old_window:       Device_Defs.device;
31     -- window editor was invoked from
32     edit_window:      Device_Defs.device;
33     open_edit_window: Device_Defs.opened_device;
34     saved:            boolean := true;
35     -- true if current version has been saved
36
37     -----
38     --                LAST CHAR IN ROW                --
39     -----
40     function Last_char_in_row(row: row_range)
41         return column_range
42     --
43     -- Logic:
44     -- Starts from the last column of the given row and works
45     -- toward the start of the line to detect the first non-null
46     -- character.
47     --
48     is
49
50         column:      column_range := last_column;
51
52     begin
53
54         while edit_buffer.lines(row)(column) = ASCII.NUL
55             loop
56             if column = first_column then
57                 EXIT;
58             else
59                 column := column - 1;
60             end if;
61         end loop;
62         return (column);
63     end Last_char_in_row;
64
65     -----
66     --                MOVE FRAME                --
67     -----
68     procedure Move_frame(direction: integer)
69     --
70     -- Logic:
71     -- Move frame in edit buffer and rewrite frame buffer.
72     -- Reposition cursor appropriately
73     --
74     is

```

```

75     column: column_range := edit_buf_pos.column;
76     -- holds cursor position in previous row
77     begin
78         frame_begin := frame_begin + direction;
79         frame_end := frame_end + direction;
80         edit_buf_pos.row := edit_buf_pos.row + direction;
81
82         Character_Display_AM.Ops.Clear(open_edit_window);
83
84         -- Rewrite frame buffer
85         -- NOTE: cursor will be at the end of the frame buffer
86         Character_Display_AM.Ops.Write(
87             opened_dev => open_edit_window,
88             buffer VA =>
89                 edit_buffer.lines(frame_begin)(first_column)'address,
90             length => System.ordinal((last_column * (frame_rows - 1)) +
91                 Last_char_in_row(frame_end) - 1));
92
93         if direction > 0 then
94             -- down:
95             -- position at the first column of the last line
96             if column > Last_char_in_row(frame_end) then
97                 column := Last_char_in_row(frame_end);
98             end if;
99             Character_Display_AM.Ops.Move_cursor_absolute(
100                 opened_dev => open_edit_window,
101                 new_pos => Terminal_Defs.point_info'
102                     (column, integer(frame_rows)));
103         end if;
104         if direction < 0 then
105             -- up:
106             -- after write, cursor will be at last char written
107             -- for upward movement we want it at the first char in
108             -- the frame buffer
109             if column > Last_char_in_row(frame_begin) then
110                 column := Last_char_in_row(frame_begin);
111             end if;
112             Character_Display_AM.Ops.Move_cursor_absolute(
113                 opened_dev => open_edit_window,
114                 new_pos => (column, first_row));
115         end if;
116     end Move_frame;
117
118     -----
119     --                               MOVE PAGE                               --
120     -----
121     function Move_page(direction: row_delta)
122         return boolean
123     is
124         window_status: Window_Services.window_status :=
125             Window_Services.Ops.Get_window_status(
126                 window => edit_window,
127                 pixel_units => false);
128         displacement: integer :=
129             window_status.window_dimensions.vert * direction;
130         cursor_pos: Terminal_Defs.point_info :=
131             Character_Display_AM.Ops.Get_cursor_position(open_edit_window);
132
133     begin
134         if direction > 0 then
135             -- if too close to the bottom move by less than window size
136             if frame_end + displacement > edit_buffer.max_lines then
137                 displacement := edit_buffer.max_lines - frame_end;
138             end if;
139         end if;
140         if direction < 0 then
141             -- if too close to the top move by less than window size
142             if frame_begin + displacement < first_row then
143                 displacement := first_row - frame_begin;
144             end if;
145         end if;
146
147         Move_frame(displacement);
148         Character_Display_AM.Ops.Move_cursor_absolute(
149             opened_dev => open_edit_window,
150             new_pos => cursor_pos);
151         edit_buf_pos.row := frame_begin + (cursor_pos.vert - 1);

```

```

152     if displacement = 0 then
153         return false;
154     else
155         return true;
156     end if;
157 end Move_page;
158
159 -----
160 --                                MOVE CURSOR                                --
161 -----
162
163 procedure Move_cursor(direction: row_delta)
164 is
165
166     -- used for current cursor position
167     cursor_pos: Terminal_Defs.point info :=
168         Character_Display_AM.Ops.Get_cursor_position(open_edit_window);
169     -- last column of row where cursor will be
170     last_col: column_range := Last_char_in_row(edit_buf_pos.row +
171         direction);
172
173 begin
174
175     edit_buf_pos.row := edit_buf_pos.row + direction;
176
177     if cursor_pos.horiz <= last_col then
178         -- Move cursor in frame buffer straight up or down
179         Character_Display_AM.Ops.Move_cursor_relative(
180             opened_dev => open_edit_window,
181             delta_col => 0,
182             delta_row => direction);
183     else
184         -- Move cursor to end of line
185         Character_Display_AM.Ops.Move_cursor_absolute(
186             opened_dev => open_edit_window,
187             new_pos => (last_col, edit_buf_pos.row));
188         edit_buf_pos.column := last_col;
189     end if;
190 end Move_cursor;
191
192 -----
193 --                                MOVE UP                                    --
194 -----
195
196 function Move_up
197     return boolean
198
199 is
200     success: boolean := true;
201 begin
202     if edit_buf_pos.row <= first_row then
203         success := false;
204     elsif edit_buf_pos.row <= frame_begin then
205         Move_frame(-1);
206     else
207         Move_cursor(-1);
208     end if;
209     return success;
210 end Move_up;
211
212 -----
213 --                                MOVE DOWN                                    --
214 -----
215
216 function Move_down
217     return boolean
218
219 is
220     success: boolean := true;
221 begin
222
223     if edit_buf_pos.row >= edit_buffer.num_lines then
224         success := false;
225     elsif edit_buf_pos.row >= frame_end then
226         Move_frame(+1);
227     else
228         Move_cursor(+1);

```

```

229     end if;
230     return success;
231 end Move_down;
232
233
234 -----
235 --                MOVE FORWARD                --
236 -----
237 function Move_forward
238     return boolean
239     --
240     -- Logic:
241     --   If cursor is at end of row then move cursor to
242     --   first column of next row; else move cursor
243     --   forward one column.  If cursor is at the end of
244     --   of the buffer return false.
245     --
246 is
247     current_pos: Terminal_Defs.point_info;
248     success: boolean := true;
249 begin
250
251     if edit_buf_pos.column = Last_char_in_row(edit_buf_pos.row) then
252     if edit_buf_pos.row = edit_buffer.num_lines then
253         success := false;  -- at the end of buffer
254     else
255         -- Move cursor to next row in frame and
256         -- frame buffer
257         success := Move_down;
258         if not success then return success; end if;
259         -- Move cursor to beginning of row in frame
260         -- and frame buffer
261         current_pos := Character_Display_AM.Ops.
262             Get_cursor_position(open_edit_window);
263         current_pos.horiz := first_column;
264         Character_Display_AM.Ops.Move_cursor_absolute(
265             opened_dev => open_edit_window,
266             new_pos    => current_pos);
267         edit_buf_pos.column := first_column;
268     end if;
269     else
270         -- move cursor to next column
271         edit_buf_pos.column := edit_buf_pos.column + 1;
272         Character_Display_AM.Ops.Move_cursor_relative(
273             opened_dev => open_edit_window,
274             delta_col  => 1,
275             delta_row  => 0);
276     end if;
277     return success;
278 end Move_forward;
279
280
281 -----
282 --                MOVE BACK                    --
283 -----
284 function Move_back
285     return boolean
286     --
287     -- Logic:
288     --   If cursor is at beginning of row then move cursor
289     --   to last column of previous row; else move cursor
290     --   back one column.  If cursor is at the beginning of
291     --   the file then return false.
292
293 is
294     current_pos: Terminal_Defs.point_info;
295     success:    boolean := true;
296 begin
297
298     if edit_buf_pos.column = first_column then
299     if edit_buf_pos.row = first_row then
300         Character_Display_AM.Ops.Ring_bell(
301             open_edit_window);
302         success := false;
303     else
304         -- Move cursor to previous row in frame and
305         -- frame buffer

```

```

306      success := Move_up;
307      if not success then return success; end if;
308      -- Move cursor to end of row
309      edit_buf_pos.column := last_char_in_row(edit_buf_pos.row);
310      current_pos := Character_Display_AM.Ops.
311      Get_cursor_position(open_edit_window);
312      current_pos.horiz := edit_buf_pos.column;
313      Character_Display_AM.Ops.Move_cursor_absolute(
314      opened_dev => open_edit_window,
315      new_pos    => current_pos);
316  end if;
317  else
318      -- move cursor to previous column
319      edit_buf_pos.column := edit_buf_pos.column - 1;
320      Character_Display_AM.Ops.Move_cursor_relative(
321      opened_dev => open_edit_window,
322      delta_col  => -1,
323      delta_row  => 0);
324  end if;
325  return success;
326  end Move_back;
327
328
329  -----
330  --          DELETE FORWARD          --
331  -----
332  function Delete_forward
333  return boolean
334  --
335  -- Logic:
336  -- Procedure will not delete characters from long
337  -- lines. It then determines if the the character
338  -- to be deleted is a line feed or not. If not it
339  -- simple deletes the character and shifts
340  -- characters beyond it one position to the left.
341  -- If the character is a line feed it determines if
342  -- the line is empty or not. If so if deletes the
343  -- line. If not it joins the current line with the
344  -- next line. In both cases lines beyond the
345  -- current line are shifted up by one row.
346  --
347  is
348  -- place holders for line joins
349  cursor_pos: Terminal_Defs.point_info :=
350  Character_Display_AM.Ops.Get_cursor_position( open_edit_window);
351  edit_pos: cursor_location := edit_buf_pos;
352
353  begin
354
355  -- no deletes on long lines
356  if Last_char_in_row(edit_buf_pos.row) = last_column then
357  Message_Services.Write_msg(no_long_lines_code);
358  return false;
359  end if;
360  if edit_buf_pos.column = Last_char_in_row(edit_buf_pos.row) then
361  if edit_buf_pos.row = edit_buffer.num_lines then
362  return false;
363  end if;
364  end if;
365
366  -- not a line feed
367  if edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column)
368  /= ASCII.LF then
369  -- Delete the character from the frame.
370  if edit_buf_pos.column = last_column then
371  edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column)
372  := ASCII.NUL;
373  else
374  for col in edit_buf_pos.column..last_column - 1 loop
375  edit_buffer.lines(edit_buf_pos.row)(col) :=
376  edit_buffer.lines(edit_buf_pos.row)(col + 1);
377  end loop;
378  end if;
379
380  edit_buffer.lines(edit_buf_pos.row)(last_column) := ASCII.NUL;
381
382  -- Delete the character from the window.

```

```

383 Character_Display_AM.Ops.Delete_char(
384   opened_dev => open_edit_window);
385
386 -- line feed
387 else
388   -- not the last line
389   if edit_buf_pos.row < edit_buffer.num_lines then
390     -- empty line delete
391     if edit_buf_pos.column = first_column then
392       -- shift rows down by one
393       for row in edit_buf_pos.row .. edit_buffer.num_lines - 1
394         loop
395           edit_buffer.lines(row) := edit_buffer.lines(row + 1);
396         end loop;
397         edit_buffer.lines(edit_buffer.num_lines) :=
398           (others => ASCII.NUL);
399         edit_buffer.num_lines := edit_buffer.num_lines - 1;
400         Character_Display_AM.Ops.Delete_line(open_edit_window);
401         -- join current line and next line
402         else
403           -- don't join if line will be too long
404           if Last_char_in_row(edit_buf_pos.row) +
405             Last_char_in_row(edit_buf_pos.row + 1) >= last_column then
406             return false;
407           end if;
408           for col in first_column .. Last_char_in_row(
409             edit_buf_pos.row + 1)
410             loop
411               edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column) :=
412                 edit_buffer.lines(edit_buf_pos.row + 1)(col);
413               edit_buf_pos.column := edit_buf_pos.column + 1;
414               EXIT when edit_buf_pos.column = last_column;
415             end loop;
416             edit_buf_pos.row := edit_buf_pos.row + 1;
417             -- shift rows down by one
418             for row in edit_buf_pos.row .. edit_buffer.num_lines - 1
419               loop
420                 edit_buffer.lines(row) := edit_buffer.lines(row + 1);
421               end loop;
422               edit_buffer.lines(edit_buffer.num_lines) :=
423                 (others => ASCII.NUL);
424               edit_buffer.num_lines := edit_buffer.num_lines - 1;
425               Move_frame(0); -- redraw
426               edit_buf_pos := edit_pos;
427               Character_Display_AM.Ops.Move_cursor_absolute(
428                 opened_dev => open_edit_window,
429                 new_pos => cursor_pos);
430             end if;
431             -- last line
432             else
433               edit_buffer.lines(edit_buf_pos.row)(edit_buf_pos.column) :=
434                 ASCII.NUL;
435             end if;
436           end if;
437           return true;
438         end Delete_forward;
439
440
441 -----
442 --          DELETE BACKWARD          --
443 -----
444 function Delete_backward
445   return boolean
446   --
447   -- Logic:
448   -- Very similar to Delete_forward except the cursor
449   -- is move back before the delete is performed.
450   --
451   is
452     success: boolean := true;
453     res:      boolean;
454   begin
455
456     if Move_back then -- back up cursor
457       success := Delete_forward; -- Delete the character.
458       -- leave cursor pos unchanged if unsuccessful
459       if not success then res := Move_forward; end if;

```

```

460     else
461         success := false;
462     end if;
463     return success;
464 end Delete_backward;
465
466
467 -----
468 --                INSERT                --
469 -----
470 function Insert(insert_char: character)
471     return boolean
472     --
473     -- Logic:
474     -- Shifts the string of characters beginning at the
475     -- cursor's location one character position to the
476     -- right. It then inserts a printable ASCII character
477     -- to the left of the cursor. If a line is already
478     -- 80 characters the insert is refused. Line feeds
479     -- are inserted by first moving all the rows beyond the
480     -- current row down by one. If there are characters
481     -- on the current line beyond the insert point they
482     -- are copied to the new line. If not just a line-
483     -- feed is put into the new line. If the file grows
484     -- beyond the current max_line size it is expanded by
485     -- resize lines.
486     --
487 is
488
489     use System; -- for adding System.ordinals
490
491     max_lines: integer;
492     For max_lines USE AT edit_buffer.max_lines'address;
493
494     edit_buffer_untyped: System.untyped_word;
495     FOR edit_buffer_untyped USE AT edit_buffer'address;
496
497     -- place holders for line splits
498     cursor_pos: Terminal_Defs.point_info :=
499         Character_Display_AM.Ops.Get_cursor_position(open_edit_window);
500     edit_pos: cursor_location := edit_buf_pos;
501     column: column_range := first_column;
502
503     success: boolean := true;
504
505 begin
506
507     -- inserts on long lines NYI
508     if Last_char_in_row(edit_buf_pos.row) = last_column then
509         Message_Services.Write_msg(no_long_lines_code);
510         return false;
511     end if;
512
513     -- If the current column is the last column in the
514     -- view, insert the new character in the frame;
515     -- else shift trailing characters one column to
516     -- the right and insert the new character.
517     --
518     if insert_char /= ASCII.LF then
519         if edit_buf_pos.column = last_column then
520             edit_buffer.lines(edit_buf_pos.row)
521                 (edit_buf_pos.column) := insert_char;
522         else
523             -- right shift characters to the right of insert position
524             for index in reverse edit_buf_pos.column + 1 .. last_column
525             loop
526                 edit_buffer.lines(edit_buf_pos.row)(index) :=
527                     edit_buffer.lines(edit_buf_pos.row)(index - 1);
528             end loop;
529
530             edit_buffer.lines(edit_buf_pos.row)
531                 (edit_buf_pos.column) := insert_char;
532             edit_buf_pos.column := edit_buf_pos.column + 1;
533         end if;
534
535         -- Insert the character in the frame buffer
536         -- (Frame buffer cursor is moved automatically)

```

```

537 Character_Display_AM.Ops.Insert_char(
538     opened_dev => open_edit_window,
539     buffer_VA  => insert_char'address,
540     num_char   => 1);
541
542 -- return
543 else
544     -- shift buffer lines beyond current row down by one
545     if edit_buffer.num_lines + 1 >= edit_buffer.max_lines then
546         -- add resize lines lines to current edit buffer size
547         Object_Mgt.Resize(
548             obj => edit_buffer_untyped,
549             size => (Object_Mgt.Get_object_size(
550                 edit_buffer_untyped) +
551                 ordinal((resize_lines * last_column) / 4));
552             max_lines := edit_buffer.num_lines + resize_lines;
553             edit_buffer.lines(edit_buffer.num_lines + 1 ..
554                 edit_buffer.max_lines) := (others => (others => ASCII.NUL));
555         end if;
556
557     -- move row down one
558     for row in reverse edit_buf_pos.row + 1 .. edit_buffer.num_lines
559     loop
560         edit_buffer.lines(row + 1) := edit_buffer.lines(row);
561     end loop;
562     -- blank fill line below current line
563     edit_buffer.lines(edit_buf_pos.row + 1) := (others => ASCII.NUL);
564     edit_buffer.num_lines := edit_buffer.num_lines + 1;
565
566     -- add return to end of line
567     if edit_buf_pos.column = Last_char_in_row(edit_buf_pos.row) then
568         success := Move_down;
569         -- first char of new line in LF
570         edit_buffer.lines(edit_buf_pos.row)(first_column) := ASCII.LF;
571         edit_buf_pos.column := first_column;
572         Character_Display_AM.Ops.Insert_line(open_edit_window);
573     -- insert return in the middle of the line (split line)
574     else
575         -- copy characters past point of insert to the next line
576         for col in edit_buf_pos.column .. Last_char_in_row(edit_buf_pos.row)
577         loop
578             edit_buffer.lines(edit_buf_pos.row + 1)(column) :=
579                 edit_buffer.lines(edit_buf_pos.row)(col);
580             -- clear line past point of insert
581             edit_buffer.lines(edit_buf_pos.row)(col) := ASCII.NUL;
582             edit_buf_pos.column := edit_buf_pos.column + 1;
583             column := column + 1;
584         end loop;
585         edit_buffer.lines(edit_pos.row)(edit_pos.column) := ASCII.LF;
586         Move_frame(0); -- redraw
587         edit_buf_pos.row := edit_buf_pos.row + 1;
588         edit_buf_pos.column := first_column;
589         Character_Display_AM.Ops.Move_cursor_absolute(
590             opened_dev => open_edit_window,
591             new_pos    => Terminal_Defs.point_info'(
592                 first_column, cursor_pos.vert + 1));
593     end if;
594 end if;
595 return success;
596 end Insert;
597
598
599 -----
600 --                               SAVE FILE                               --
601 -----
602 procedure Save_file
603 --
604 -- Logic:
605 -- Writes the file in linear_buf_size amounts copied
606 -- from the edit_buffer which is an array of lines
607 -- to the linear buffer. It checks for backslashes
608 -- in the last column and rejoins long lines.
609 -- Before writing the new file, it must be truncated
610 -- and the pointer moved back to zero.
611 --
612 is
613

```

```

614     opened_file: Device_Defs.opened_device;
615     file_ptr:    Long_Integer_Defs.long_integer;
616     linear_buffer: array (1 .. linear_buf_size) of
617         character := (others => ASCII.NUL);
618     index:       integer := 1;
619
620 begin
621
622     opened_file := Byte_Stream_AM.Open_by_name(
623         name      => file_name,
624         input_output => Device_Defs.output,
625         allow     => Device_Defs.nothing);
626
627     -- delete data in original file
628     Byte_Stream_AM.Ops.Truncate(
629         opened_dev => opened_file,
630         new_length => Long_Integer_Defs.zero);
631
632     file_ptr := Byte_Stream_AM.Ops.Set_position(
633         opened_dev => opened_file,
634         pos       => Long_Integer_Defs.zero,
635         mode      => Byte_Stream_AM.from_begin);
636
637     for row in 1 .. edit_buffer.num_lines
638     loop
639         -- write each line to linear buffer until LF
640         for col in first_column .. last_column
641         loop
642             -- write out linear buffer when full;
643             if index > linear_buf_size then
644                 Byte_Stream_AM.Ops.Write(
645                     opened_dev => opened_file,
646                     buffer_VA  => linear_buffer'address,
647                     length     => System.ordinal(linear_buffer'size / 8));
648                 linear_buffer := (others => ASCII.NUL);
649                 index := 1;
650             end if;
651             -- reproduce long lines
652             if col < last_column or
653                edit_buffer.lines(row)(last_column) /= '\' then
654                 linear_buffer(index) := edit_buffer.lines(row)(col);
655                 index := index + 1;
656                 EXIT when edit_buffer.lines(row)(col) = ASCII.LF;
657             end if;
658         end loop;
659     end loop;
660
661     Byte_Stream_AM.Ops.Write(
662         opened_dev => opened_file,
663         buffer_VA  => linear_buffer'address,
664         length     => System.ordinal(index));
665
666     Byte_Stream_AM.Ops.Close(opened_file);
667
668 exception
669
670     when Directory_Mgt.no_access =>
671         Message_Services.Write_msg(
672             msg_id => new_file_code,
673             param1 => Incident_Defs.message_parameter'(
674                 typ  => Incident_Defs.txt,
675                 len  => file_name.length,
676                 txt_val => file_name));
677
678 end Save_file;
679
680
681 -----
682 --                               QUIT EDITOR                               --
683 -----
684 procedure Quit_editor
685
686 is
687     quit:          exception;
688 begin
689
690     Window_Services.Ops.Transfer_input_focus(

```

```

691     source_window => edit_window,
692     target_window => old_window);
693
694 if not saved then
695     if not Message_Services.Acknowledge_Msg(not_saved_code) then
696         Window_Services.Ops.Transfer_input_focus(
697             source_window => old_window,
698             target_window => edit_window);
699         return;
700     end if;
701 end if;
702
703 Character_Display_AM.Ops.Close(open_edit_window);
704 Window_Services.Ops.Destroy_window(edit_window);
705 RAISE quit;
706
707 exception
708
709     when quit =>
710         RAISE;
711
712 end Quit_editor;
713
714 -----
715 --                               CLOSE INPUT                               --
716 -----
717 procedure Close_input
718     -- NYI (for menus)
719 is
720 begin
721     null;
722 end Close_input;
723
724 -----
725 --                               READ FILE                               --
726 -----
727 procedure Read_file
728     --
729     -- Logic:
730     -- Reads the input file into a linear buffer.
731     -- That is read one line feed to a row into
732     -- the edit buffer. The edit buffer is expanded
733     -- by resize lines increments as needed.
734     -- A backslash is place in the last column for
735     -- lines over 80 characters long.
736     --
737 is
738
739     use System;    -- for adding System.ordinals
740
741     characters_read:    System.ordinal;
742     opened_file:       Device_Defs.opened_device;
743     linear_buffer:     array (1 .. linear_buf_size) of character;
744     col, row:          integer := 1;
745     file:              File_Defs.file_AD;
746
747     max_lines:         integer;
748     For max_lines USE AT edit_buffer.max_lines'address;
749
750     edit_buffer_untyped: System.untyped_word;
751     FOR edit_buffer_untyped USE AT edit_buffer'address;
752
753 begin
754
755     opened_file := Byte_Stream_AM.Open_by_name(
756         name      => file_name,
757         input_output => Device_Defs.input);
758
759     loop
760
761         -- read by linear_buf_size blocks
762         characters_read := Byte_Stream_AM.Ops.Read(
763             opened_dev => opened_file,
764             buffer_VA  => linear_buffer'address,
765             length     => System.ordinal(linear_buffer'size / 8));
766
767         for index in 1 .. integer(characters_read)

```

```

768     loop
769     if row > max_lines then
770         -- add resize_lines lines to current edit buffer size
771         Object_Mgt.Resize(
772             obj => edit_buffer_untyped,
773             size => (Object_Mgt.Get_object_size(
774                 edit_buffer_untyped) +
775                 ordinal((resize_lines * last_column) / 4)));
776         max_lines := edit_buffer.num_lines + resize_lines;
777         -- initialize expanded area
778         edit_buffer.lines(edit_buffer.num_lines + 1 ..
779             edit_buffer.max_lines) :=
780             (others => (others => ASCII.NUL));
781         end if;
782
783     if linear_buffer(index) = ASCII.LF then
784         edit_buffer.lines(row)(col) := linear_buffer(index);
785         edit_buffer.num_lines := edit_buffer.num_lines + 1;
786         col := 1;
787         row := row + 1;
788     else
789         if col < last_column then
790             edit_buffer.lines(row)(col) := linear_buffer(index);
791             col := col + 1;
792         else -- long line
793             edit_buffer.lines(row)(last_column) := '\';
794             edit_buffer.num_lines := edit_buffer.num_lines + 1;
795             col := 1;
796             row := row + 1;
797             edit_buffer.lines(row)(col) := linear_buffer(index);
798         end if;
799     end if;
800     end loop;
801 end loop;
802
803 Byte_Stream_AM.Ops.Close(opened_file);
804
805 exception
806
807     -- make a new file
808     when Directory_Mgt.no_access =>
809         Message_Services.Write_msg(
810             msg_id => new_file_code,
811             param1 => Incident_Defs.message_parameter' (
812                 typ    => Incident_Defs.txt,
813                 len    => file_name.length,
814                 txt_val => file_name));
815         file := Simple_File_Admin.Create_file(file_name);
816         RETURN;
817
818     -- successful completion
819     when Device_Defs.end_of_file =>
820         Byte_Stream_AM.Ops.Close(opened_file);
821
822 end Read_file;
823
824 -----
825 --                               MAKE WINDOW                               --
826 -----
827 procedure Make_window
828 is
829
830     underlying_terminal:    Device_Defs.device;
831     new_window_info:      Window_Services.window_style_info;
832     window_attributes:    Terminal_Defs.window_attr :=
833         Terminal_Defs.default_window_attr;
834
835 begin
836
837     -- Create new window from old opened window.
838     old_window := Character_Display_AM.Ops.
839         Get_device_object(Process_Mgt.Get_process_globals_entry(
840             Process_Mgt.Types.standard_input));
841     underlying_terminal := Window_Services.Ops.
842         Get_terminal(old_window);
843     edit_window := Window_Services.Ops.Create_window(
844         terminal    => underlying_terminal,

```

```

845     pixel_units      => false,
846     fb_size          => Terminal_Defs.point_info'(
847         last_column, frame_rows),
848     desired_window_size => Terminal_Defs.point_info'(
849         last_column, preferred_window_rows),
850     window_pos       => origin,
851     view_pos         => origin);
852 -- Set window's input and output attributes
853 -- change from default:
854 window_attributes.enable_signal := false; -- for ^C ^B
855 window_attributes.line_editing := false;  -- for ^H
856 window_attributes.echo := false;
857 -- NOTE: track_cursor NYI (use user agent to change view)
858 window_attributes.track_cursor := true;
859 Window_Services.Ops.Set_window_attr(
860     window => edit_window,
861     attr   => window_attributes,
862     attr_mask => (others => true));
863 -- Set Title and Info lines
864 Text_Mgt.Set(new_window_info.title, file_name);
865 Window_Services.Ops.Set_window_style(
866     window => edit_window,
867     new_info => new_window_info,
868     style_list => (others => true));
869
870 -- Open the edit window
871 open_edit_window := Character_Display_AM.Ops.Open(
872     device      => edit_window,
873     input_output => Device_Defs.inout,
874     exclusive   => true);
875
876 -- Clear window on terminal screen.
877 Character_Display_AM.Ops.Clear(open_edit_window);
878
879 -- Write from edit buffer to frame buffer.
880 -- NOTE: There cannot be more line_feeds in the length
881 -- of characters written than there are rows in
882 -- the frame buffer, otherwise some of the first
883 -- characters will be overwritten in the frame buffer
884 -- The last line is written up to the line feed to
885 -- avoid having a blank line at bottom of the window
886 Character_Display_AM.Ops.Write(
887     opened_dev => open_edit_window,
888     buffer_VA  => edit_buffer.lines'address,
889     length     => System.ordinal((last_column * (frame_rows - 1))
890         + (Last_char_in_row(frame_end) - 1)));
891
892 -- Home the cursor (1,1 position).
893 Character_Display_AM.Ops.Move_cursor_absolute(
894     opened_dev => open_edit_window,
895     new_pos    => origin);
896
897 Window_Services.Ops.Transfer_input_focus(
898     source_window => old_window,
899     target_window => edit_window);
900
901 end Make_window;
902
903
904
905 -----
906 --                               HANDLE INPUT                               --
907 -----
908 procedure Handle_input
909
910 is
911
912     event_num:      System.ordinal;
913     event_type:    Terminal_Defs.input_enum;
914     char_buffer_AD: char_array_AD := new char_array'(others => ' ');
915
916 begin
917
918     -- Enter the basic read and process loop
919     loop
920         -- Read the next input event
921         -- default input mask is keyboard

```

```

922     Character_Display_AM.Ops.Read(
923         opened_dev => open_edit_window,
924         buffer_VA  => char_buffer_AD.all'address,
925         max_events => 1,
926         max_bytes  => 0,
927         block      => true,
928         type_read  => event_type,
929         num_read   => event_num);
930     case event_type is
931     when Terminal_Defs.keyboard =>
932         -- ...
933         key_input(char_buffer_AD(1));
934     when Terminal_Defs.menu_item_picked =>
935         -- ...
936         key_input(char_buffer_AD(1));
937     when others =>
938         null;
939     end case;
940 end loop;
941 end Handle_input;
942
943 -----
944 --                               IS PRINTABLE                               --
945 -----
946 function Is_printable(c: character)
947     return boolean
948 --
949 -- Logic:
950 -- Checks if character entered in printable
951 --
952 is
953 begin
954
955     if c >= ' ' or c = ASCII.LF then return true;
956     else return false;
957     end if;
958
959 end Is_printable;
960 pragma inline(Is_printable);
961 -----
962 --                               KEY INPUT                               --
963 -----
964 procedure Key_input(key: character)
965 is
966     result:    boolean := true;
967     cursor_pos: Terminal_Defs.point_info;
968 begin
969
970     -- Process the event
971     case key is
972     when ASCII.ACK =>
973         result := Move_forward;           -- Control F
974     when ASCII.STX =>
975         result := Move_back;             -- Control B
976     when ASCII.DLE =>
977         result := Move_up;               -- Control P
978     when ASCII.SO  =>
979         result := Move_down;            -- Control N
980     when ASCII.NAK =>
981         result := Move_Page(-1);        -- Control U
982     when ASCII.SYN =>
983         result := Move_page(+1);        -- Control V
984     when ASCII.EOT =>
985         result := Delete_forward;       -- Control D
986         saved := false;
987     when ASCII.BS  =>
988         result := Delete_backward;      -- Control H
989         saved := false;
990     when ASCII.ETB =>
991         Save_file;                       -- Control W
992         saved := true;
993     when ASCII.ETX =>
994         Quit_editor;                     -- Control C
995     when others =>
996         --Insert text.
997         if Is_printable(key) then
998

```

```

999         result := Insert(key);
1000         saved := false;
1001         else Character_Display_AM.Ops.Ring_bell(open_edit_window);
1002         end if;
1003     end case;
1004     if not result then
1005         Character_Display_AM.Ops.Ring_bell(open_edit_window);
1006     end if;
1007     -- cursor check
1008     cursor_pos := Character_Display_AM.Ops.Get_cursor_position(
1009         open_edit_window);
1010     if edit_buf_pos.row /= frame_begin + (cursor_pos.vert - 1) or
1011        edit_buf_pos.column /= cursor_pos.horiz then
1012         RAISE editor_error;
1013     end if;
1014
1015     exception
1016
1017         when editor_error =>
1018             Message_Services.Write_msg(editor_error_code);
1019             return;
1020
1021     end Key_input;
1022
1023 end Simple_Editor_Ex;

```

## X-A.4.20 Stream\_file\_ex Procedure

```
1  with Directory_Mgt,
2      File_Defs,
3      Passive_Store_Mgt,
4      Process_Mgt,
5      Process_Mgt_Types,
6      Simple_File_Admin,
7      System_Defs,
8      Text_Mgt;
9
10 procedure Stream_file_ex is
11     --
12     -- Function:
13     -- Provide example calls for stream files.
14
15     filename: System_Defs.text(60);
16     file1:    File_Defs.file_AD;
17     file2:    File_Defs.file_AD;
18     file3:    File_Defs.file_AD;
19     begin
20         Text_Mgt.Set(filename, "my_file_1");
21         file1 := Simple_File_Admin.Create_file(filename);
22         -- Creates a stream file in the current
23         -- directory.
24
25         -- Code to write something into the file
26         -- could go here.
27
28         Text_Mgt.Set(filename, "my_file_2");
29         file2 := Simple_File_Admin.Create_file(filename);
30         Simple_File_Admin.Copy_file(source_file => file1,
31                                     target_file => file2);
32         -- Creates a second file in the current directory,
33         -- and then copies the contents of the first file
34         -- to the second.
35
36         Simple_File_Admin.Empty_file(file1);
37         -- Empties the first file.
38
39         Text_Mgt.Set(filename, "my_file_2");
40         Directory_Mgt.Delete(filename);
41         -- The second file's pathname is deleted. The
42         -- second file is destroyed when the last
43         -- reference to it goes away.
44
45         file2 := Simple_File_Admin.Create_unnamed_file(
46             Passive_Store_Mgt.Home_volume_set(
47                 Process_Mgt.Get_process_globals_entry(
48                     Process_Mgt_Types.current_dir)));
49         -- Creates a temporary file in the current
50         -- directory using the current directory's
51         -- volume set.
52
53         Text_Mgt.Set(filename, "my_local_name");
54         Simple_File_Admin.Save_unnamed_file(
55             name => filename,
56             file => file2);
57         -- Names and saves the temporary file so that it
58         -- can be used in future jobs.
59
60         file3 := Simple_File_Admin.Create_unnamed_file(
61             Passive_Store_Mgt.Home_volume_set(
62                 Process_Mgt.Get_process_globals_entry(
63                     Process_Mgt_Types.current_dir)));
64         -- Creates another temporary file in the current
65         -- directory.
66
67         Simple_File_Admin.Destroy_file(file3);
68         -- Destroys the temporary file before its job
69         -- terminates. If it is not destroyed or saved,
70         -- it goes away when the job terminates.
71
72     end Stream_file_ex;
```

## **X-A.5 Human Interface Services**

## X-A.5.1 Inventory\_main Procedure

```

1  with Character_Display_AM,
2      Device_Defs,
3      Incident_Defs,
4      Inventory_Files,
5      Inventory_Menus,
6      Inventory_Messages,
7      Inventory_Windows,
8      Message_Services,
9      System,
10     Terminal_Defs;
11
12  --
13  -- Function:
14  -- Main (top-level) procedure for Inventory
15  -- Example Program.
16  --
17  -- The procedure "Inventory_main" is called from
18  -- CLEX. "Inventory_main" performs the top-level
19  -- processing for the Inventory Example Program:
20  -- program initialization, main processing loop,
21  -- and termination.
22  --
23  -- History:
24  -- 05-20-87, William A. Rohm:  Written.
25  -- 10-27-87, WAR:             Revised.
26  --
27  -- End of Header
28
29  procedure Inventory_main
30  --
31  -- Logic:
32  -- 1. Define incident codes.
33  -- 2. Open windows and files.
34  -- 3. Install and enable menu group, enable menu
35  --    selection
36  -- 4. Process each menu selection until Exit
37  -- 5. Close files and windows.
38  is
39
40  -- Incident codes for messages:
41  --
42  module: constant := 1;
43  -- Message module index number.
44
45  -- *M* set.language :language = English
46  -- *M* create.variable module :value = 1
47
48
49  welcome_code: constant
50      Incident_Defs.incident_code := (
51          message_object =>
52              Inventory_Messages.message_object,
53          module          => module,
54          number          => 0,
55          severity        => Incident_Defs.information);
56
57  -- *M* store :module = $module :number = 0\
58  -- *M* :msg_name = welcome \
59  -- *M* :short = "Welcome to the Inventory
60  -- *M* Example Program."
61
62
63  terminated_code: constant
64      Incident_Defs.incident_code := (
65          message_object =>
66              Inventory_Messages.message_object,
67          module          => module,
68          number          => 1,
69          severity        => Incident_Defs.information);
70
71  -- *M* store :module = $module :number = 1\
72  -- *M* :msg_name = terminated \
73  -- *M* :short = "Inventory Example Program
74  -- *M* terminated."

```

```

75
76
77 -- Variables:
78 --
79 menu_select: Terminal_Defs.menu_selection;
80 -- Menu selection record for receiving user
81 -- input from "Character_Display_AM.Ops.Read".
82 --
83 -- Contains user's menu group, menu, and item
84 -- selection numbers.
85
86 event_type: Terminal_Defs.input_enum;
87 -- Type of user input event (returned from
88 -- "Character_Display_AM.Ops.Read").
89
90 event_num: System.ordinal;
91 -- Number of user input events (returned from
92 -- "Character_Display_AM.Ops.Read").
93
94
95 -- Inventory_main procedure:
96 --
97 begin
98
99 -- Open both main and message windows:
100 --
101 Inventory_Windows.Open_program_windows;
102
103 -- Display "Welcome" message:
104 --
105 Message_Services.Write_msg(
106     msg_id => welcome_code);
107
108
109 -- Open files:
110 --
111 Inventory_Files.Open_parts_file;
112
113 Inventory_Files.Open_log_file;
114
115
116 -- Retrieve and install menu group:
117 --
118 Inventory_Menus.Set_up_menu_group;
119
120
121 -- Set input event type mask for menu item selection
122 -- only:
123 --
124 Character_Display_AM.Ops.Set_input_type_mask(
125     opened_dev => Inventory_Windows.main_window,
126     new_mask   => Terminal_Defs.input_type_mask'(
127         Terminal_Defs.menu_item_picked => true,
128         others                          => false));
129
130
131 -- Main processing loop:
132 --
133 loop
134
135     -- Wait for and read next input event (must have
136     -- been a menu selection):
137     --
138     Character_Display_AM.Ops.Read(
139         opened_dev => Inventory_Windows.main_window,
140         buffer_VA  => menu_select'address,
141         max_events => 1,
142         max_bytes  => 0,
143         block      => true, -- Wait . . .
144         type_read  => event_type,
145         num_read   => event_num);
146
147
148 -- Act on menu selection:
149 --
150 case menu_select.menu is
151

```

```

152
153     when Inventory_Menus.inquiry_menu_ID =>
154         Inventory_Menus.Process_inquiry_menu(
155             selection => menu_select.item);
156
157     when Inventory_Menus.posting_menu_ID =>
158         Inventory_Menus.Process_posting_menu(
159             selection => menu_select.item);
160
161     when Inventory_Menus.update_menu_ID =>
162         Inventory_Menus.Process_update_menu(
163             selection => menu_select.item);
164
165     when Inventory_Menus.report_menu_ID =>
166         Inventory_Menus.Process_report_menu(
167             selection => menu_select.item);
168
169     when Inventory_Menus.housekeeping_menu_ID =>
170         Inventory_Menus.Process_housekeeping_menu(
171             selection => menu_select.item);
172
173     when Inventory_Menus.exit_menu_ID =>
174         EXIT;
175
176     when others =>
177         null;
178
179     end case;          -- "case menu_select.menu is"
180
181 end loop;
182
183 -- Close files:
184 --
185 Inventory_Files.Close_parts_file;
186
187 Inventory_Files.Close_log_file;
188
189
190 -- Write "terminated" message:
191 --
192 Message_Services.Write_msg(
193     msg_id => terminated_code);
194
195
196 -- Close both program windows.  When the main
197 -- window is closed, the menu group is deallocated:
198 --
199 Inventory_Windows.Close_program_windows;
200
201 end Inventory_main;
202
203
204

```

## X-A.5.2 Inventory\_Files Package Specification

```
1  with Device_Defs,
2      Incident_Defs,
3      Inventory_Messages,
4      System,
5      System_Defs,
6      Timing_Conversions;
7
8  package Inventory_Files is
9      --
10     -- Function:
11     -- Contains all operations related to
12     -- Inventory Program files.
13     --
14     -- This package contains the necessary calls
15     -- to open and close the two inventory files
16     -- (parts file and log file), and calls to
17     -- read and write records in the parts file,
18     -- and to write records to the log file.
19     --
20     -- History:
21     -- 05-20-87, William A. Rohm: Written.
22     -- 11-02-87, WAR: Revised.
23     --
24     -- End of Header
25
26     -- Incident codes for messages:
27     --
28     module: constant := 3;
29     -- Message module index.
30
31     --*M*  set.language :language=english
32     --*M*  create.variable module :value = 3
33
34     no_modify_rights_code: constant
35         Incident_Defs.incident_code := (
36             message_object =>
37                 Inventory_Messages.message_object,
38             module          => module,
39             number          => 1,
40             severity        => Incident_Defs.error);
41
42
43     --*M*  store :module = $module \
44     --*M*         :number = 1 \
45     --*M*         :msg_name = no_mod_rights \
46     --*M*         :short = "No modify rights for
47     --*M*           parts file '$pl<parts file
48     --*M*           name>'."
49
50
51
52     no_parts_file_code: constant
53         Incident_Defs.incident_code := (
54             message_object =>
55                 Inventory_Messages.message_object,
56             module          => module,
57             number          => 2,
58             severity        => Incident_Defs.error);
59
60     --*M*  store :module = $module \
61     --*M*         :number = 2 \
62     --*M*         :msg_name = no_parts_file \
63     --*M*         :short = "Parts file '$pl<parts
64     --*M*           file name>' does not
65     --*M*           exist."
66
67
68     no_log_file_code: constant
69         Incident_Defs.incident_code := (
70             message_object =>
71                 Inventory_Messages.message_object,
72             module          => module,
73             number          => 3,
74             severity        => Incident_Defs.error);
```

```

75
76  --*M*   store :module = $module \
77  --*M*   :number = 3 \
78  --*M*   :msg_name = no_log_file \
79  --*M*   :short = "Log file '$pl<log
80  --*M*   file name>' does not
81  --*M*   exist."
82
83
84  index_inconsistent_code: constant
85      Incident_Defs.incident_code := (
86          message_object =>
87              Inventory_Messages.message_object,
88          module          => module,
89          number          => 4,
90          severity       => Incident_Defs.error);
91
92  --*M*   store :module = $module \
93  --*M*   :number = 4 \
94  --*M*   :msg_name = \
95  --*M*   index_inconsistent \
96  --*M*   :short = "Parts file
97  --*M*   '$pl<parts file name>' index
98  --*M*   is inconsistent and must be
99  --*M*   redone. Select the
100 --*M*   Housekeeping Menu's item
101 --*M*   'Index Parts File'."
102
103
104  not_on_file_code: constant
105      Incident_Defs.incident_code := (
106          message_object =>
107              Inventory_Messages.message_object,
108          module          => module,
109          number          => 5,
110          severity       => Incident_Defs.error);
111
112  --*M*   store :module = $module \
113  --*M*   :number = 5 \
114  --*M*   :msg_name = not_on_file \
115  --*M*   :short = "There is no parts
116  --*M*   record for part ID '$pl<part
117  --*M*   ID (index value)>' does not
118  --*M*   exist."
119
120  not_on_file: exception;
121      pragma exception_value(not_on_file,
122                             not_on_file_code);
123      -- Raised by "Read_parts_record" and
124      -- "Rewrite_parts_record".
125
126
127  invalid_part_ID_code: constant
128      Incident_Defs.incident_code := (
129          message_object =>
130              Inventory_Messages.message_object,
131          module          => module,
132          number          => 6,
133          severity       => Incident_Defs.error);
134
135  --*M*   store :module = $module \
136  --*M*   :number = 6 \
137  --*M*   :msg_name = invalid_part_ID \
138  --*M*   :short = "An invalid part ID,
139  --*M*   '$pl<part ID (index
140  --*M*   value)>', was entered."
141
142  invalid_part_ID: exception;
143      pragma exception_value(invalid_part_ID,
144                             invalid_part_ID_code);
145      -- Raised by "Read_parts_record",
146      -- "Write_parts_record", and
147      -- "Rewrite_parts_record".
148
149
150  already_on_file_code: constant
151      Incident_Defs.incident_code := (

```

```

152     message_object =>
153         Inventory_Messages.message_object,
154     module      => module,
155     number      => 7,
156     severity    => Incident_Defs.error);
157
158     --*M*   store :module = $module \
159     --*M*   :number = 7 \
160     --*M*   :msg_name = already_on_file \
161     --*M*   :short = "Parts record for part
162     --*M*   ID '$pl<part ID (index
163     --*M*   value)>' already exists.
164     --*M*   Either choose a new part ID,
165     --*M*   or update the current part's
166     --*M*   record."
167
168     already_on_file: exception;
169     pragma exception_value(already_on_file,
170                            already_on_file_code);
171     -- Raised by "Read_parts_record" and
172     -- "Write_parts_record".
173
174
175     -- Constants:
176     --
177     parts_file_str: constant string :=
178         "/example/inventory/parts_file";
179     -- String constant for parts file's
180     -- pathname.
181
182     parts_file_pathname: System_Defs.text(
183         Incident_Defs.txt_length) := (
184         Incident_Defs.txt_length,
185         parts_file_str'length,
186         parts_file_str);
187     -- Text constant from parts file's pathname
188     -- string.
189     part_ID_index_str: constant string :=
190         "part_ID_index";
191     -- String constant for parts file's
192     -- index's name.
193
194     part_ID_index_name: System_Defs.text(
195         part_ID_index_str'length) := (
196         part_ID_index_str'length,
197         part_ID_index_str'length,
198         part_ID_index_str);
199     -- Text constant from parts file's index's
200     -- name string.
201
202     log_file_str: constant string :=
203         "/example/inventory/log_file";
204     -- String constant for log file's
205     -- pathname.
206
207     log_file_pathname: System_Defs.text(
208         Incident_Defs.txt_length) := (
209         Incident_Defs.txt_length,
210         log_file_str'length,
211         log_file_str);
212     -- Text constant from log file's pathname
213     -- string.
214
215
216     -- Variables:
217     --
218     parts_file: Device_Defs.opened_device;
219     -- AD to inventory parts file.
220
221     log_file: Device_Defs.opened_device;
222     -- AD to inventory log file.
223
224
225     -----
226     --           Inventory Parts File Record Definition
227     -----
228

```

PRELIMINARY

```

229  -- Constants:
230  --
231  part_ID_length:    constant integer := Incident_Defs.txt_length;
232  desc_length:      constant integer := 30;
233  unit_length:      constant integer := 4;
234  loc_length:       constant integer := 12;
235  status_length:    constant integer := 7;
236  max_suppliers:    constant integer := 3;
237  supplier_ID_length: constant integer := 10;
238
239  qty_digits:       constant integer := 7;
240
241
242  -- Types:
243  --
244  subtype part_ID_type is System_Defs.text(
245     part_ID_length);
246
247  subtype supplier_ID_type is System_Defs.text(
248     supplier_ID_length);
249
250  subtype location_type is System_Defs.text(
251     loc_length);
252
253  --type qty_type is digits qty_digits;
254  subtype qty_type is System.ordinal
255     range 0..9_999_999;
256
257  --type cost_type is delta 0.01
258  --     range 0.0 .. 99_999_999.99;
259  subtype cost_type is float
260     range 0.0..99_999_999.99;
261
262  type supplier_array_type is
263     array (1..max_suppliers) of supplier_ID_type;
264     -- Array of supplier IDs.
265
266  type parts_record_type is
267     -- Record declaration for parts file
268     -- records.
269     record
270         part_ID:          part_ID_type;
271         -- Part identification code (ID).
272         desc:             System_Defs.text(
273             desc_length);
274         -- Description of part.
275         unit:             System_Defs.text(
276             unit_length);
277         -- Unit of measure.
278         location:        location_type;
279         -- Warehouse location of part.
280         qty_on_hand:     qty_type;
281         reorder_point:   qty_type;
282         reorder_qty:     qty_type;
283         suppliers:       supplier_array_type;
284         -- Array of suppliers for this part.
285         usage_this_month: qty_type;
286         usage_last_month: qty_type;
287         usage_last_year:  qty_type;
288         avg_unit_cost:    cost_type;
289         last_unit_cost:   cost_type;
290         date_first_act:
291             Timing_Conversions.numeric_time;
292         -- Date and time of first activity with
293         -- this part (entered into parts file).
294         date_last_act:
295             Timing_Conversions.numeric_time;
296         -- Date and time of last activity with
297         -- this part.
298         status:          System_Defs.text(
299             status_length);
300         -- Status of this part ("on order", "on
301         -- hold", "obsolete", ...).
302     end record;
303
304
305  -----

```

```

306 --      Inventory Log File Record Definition
307 -----
308
309 -- Constants:
310 --
311 doc_length:      constant integer := 12;
312 job_length:      constant integer := 32;
313
314 -- Types:
315 --
316 type action_type is (
317     create,
318     -- Create new parts record
319     update,
320     -- Update parts record
321     delete,
322     -- Delete parts record
323     receipt,
324     issue,
325     returns,
326     spoilage,
327     journal);
328
329
330 type log_record_type is
331 -- Record declaration for log file records.
332 record
333     part_ID:      part_ID_type;
334     -- Part ID used in current action.
335     action:      action_type;
336     -- Action performed with this part ID.
337     time:
338         Timing_Conversions.numeric_time;
339     -- Date and time of action.
340     doc_number:  System_Defs.text(
341         doc_length);
342     -- Supplier's document number.
343     qty:        qty_type;
344     -- Taken from
345     -- "parts_file_record.qty_on_hand".
346     job_ID:     System_Defs.text(
347         job_length);
348     -- ID of job which called Inventory
349     -- Example Program to perform action.
350     supplier_ID: supplier_ID_type;
351     -- ID of supplier for this part and
352     -- action.
353 end record;
354
355
356 -- Parts file procedures:
357 -- Open / Read / Write / Rewrite / Close file
358 --
359
360 procedure Open_parts_file;
361 --
362 -- Function:
363 -- Opens inventory parts file.
364
365
366 procedure Read_parts_record(
367     part_ID:      part_ID_type;
368     -- Part ID of record to be read.
369     msg_on_error: boolean := false;
370     -- Optional parameter specifying whether
371     -- a message is displayed when an
372     -- exception is raised. Default is no
373     -- message.
374     parts_record: out parts_record_type);
375     -- Variable that receives parts record.
376 --
377 -- Function:
378 -- Reads a record from the inventory parts
379 -- file.
380 --
381 -- Exceptions:
382 -- not_on_file - "part_ID" does not index

```

PRELIMINARY

```

383         --           an existing parts record.
384     --   invalid_part_ID - "part_ID" contains an
385     --           invalid value.
386
387
388     procedure Write_parts_record(
389         parts_record: parts_record_type);
390         -- Record to be written.
391     --
392     -- Function:
393     --   Writes a record to the inventory parts
394     --   file.
395     --
396     -- Exceptions:
397     --   already_on_file - "part_ID" indexes
398     --   an existing parts record.
399     --   invalid_part_ID - "part_ID" contains an
400     --   invalid value.
401
402
403     procedure Rewrite_parts_record(
404         parts_record: parts_record_type);
405         -- Record to be rewritten.
406     --
407     -- Function:
408     --   Rewrites a record in the inventory
409     --   parts file.
410     --
411     -- Exceptions:
412     --   not_on_file - "part_ID" does not index
413     --   an existing parts record.
414     --   invalid_part_ID - "part_ID" contains an
415     --   invalid value.
416
417
418     procedure Delete_parts_record(
419         part_ID: part_ID_type);
420         -- ID of record to be deleted.
421     --
422     -- Function:
423     --   Deletes a record in the inventory parts file.
424     --
425     -- Exceptions:
426     --   not_on_file - "part_ID" does not index
427     --   an existing parts record.
428     --   invalid_part_ID - "part_ID" contains an
429     --   invalid value.
430
431
432     procedure Close_parts_file;
433     --
434     -- Function:
435     --   Closes inventory parts file.
436
437
438     -- Log file procedures:
439     --   Open / Write / Close log file
440     --
441
442     procedure Open_log_file;
443     --
444     -- Function:
445     --   Opens inventory log file.
446
447
448     procedure Write_log_record(
449         parts_record: parts_record_type;
450         -- Affected parts record.
451         action: action_type);
452         -- Action taken with parts record.
453     --
454     -- Function:
455     --   Creates and writes a record to the inventory
456     --   log file.
457
458
459     procedure Close_log_file;

```

**PRELIMINARY**

```
460      --  
461      -- Function:  
462      --   Closes inventory log file.  
463  
464 end Inventory_Files;
```

## X-A.5.3 Inventory\_Files Package Body

```

1  with Access_Mgt,
2      Device_Defs,
3      Directory_Mgt,
4      Incident_Defs,
5      Inventory_Windows,
6      Message_Services,
7      Message_Stack_Mgt,
8      Object_Mgt,
9      Record_AM,
10     System,
11     System_Defs,
12     System_Exceptions,
13     Timed_Requests_Mgt,
14     Timing_Conversions,
15     Unchecked_conversion;
16
17
18 package body Inventory_Files is
19     --
20     -- Function:
21     --   Contains all operations related to Inventory
22     --   Program files.
23     --
24     -- History:
25     --   05-20-87, William A. Rohm:   Written.
26     --   10-27-87, WAR:              Revised.
27     --
28     -- End of Header
29
30
31 -- Generic function:
32 --
33 function Device_from_untyped_word is new
34     Unchecked_conversion(
35         source => System.untyped_word,
36         target => Device_Defs.device);
37
38
39 -- Parts file procedures:
40 --   Open / Read / Write / Rewrite / Close parts file
41
42 procedure Open_parts_file
43 is
44     parts_file_AD: System.untyped_word;
45
46 begin
47
48     -- Retrieve parts file, if possible:
49     --
50     parts_file_AD := Directory_Mgt.Retrieve(
51         name => parts_file_pathname);
52
53     -- Check for access (modify) rights for parts file:
54     --
55     if not Access_Mgt.Permits(
56         AD      => parts_file_AD,
57         rights => Object_Mgt.modify_rights)
58     then
59         Message_Services.Write_msg(
60             msg_id => no_modify_rights_code,
61             param1 => Incident_Defs.message_parameter(
62                 typ => Incident_Defs.txt,
63                 len => parts_file_pathname.length)' (
64                 typ      => Incident_Defs.txt,
65                 len      => parts_file_pathname.length,
66                 txt_val => parts_file_pathname));
67     end if;
68     -- Open parts file:
69     --
70     parts_file := Record_AM.Ops.Open(
71         dev      => Device_from_untyped_word(
72             parts_file_AD),
73         input_output => Device_Defs.inout,
74         allow      => Device_Defs.readers);

```

```

75
76 exception
77   -- Exceptions from "Directory_Mgt.Retrieve",
78   -- "Record_AM.Ops.Open":
79   --
80   when others =>
81     Message_Services.Write_msg(
82       msg_id => no_parts_file_code,
83       param1 => Incident_Defs.message_parameter(
84         typ => Incident_Defs.txt,
85         len => parts_file_pathname.length)' (
86         typ => Incident_Defs.txt,
87         len => parts_file_pathname.length,
88         txt_val => parts_file_pathname));
89
90   end Open_parts_file;
91
92
93   procedure Read_parts_record(
94     part_ID:      part_ID_type;
95     msg_on_error: boolean := false;
96     parts_record: out parts_record_type)
97   is
98     bytes_read: System.ordinal;
99
100   use System;  -- To import "/"=" for
101                -- "System.ordinal", and division for
102                -- "'size/8" constructions
103
104   begin
105
106     -- Read given record, if any:
107     --
108     bytes_read := Record_AM.Keyed_Ops.Read_by_key(
109       opened_dev => parts_file,
110       buffer_VA  => parts_record'address,
111       length     => parts_record'size/8,
112       index      => part_ID_index_name,
113       key_buffer => Record_AM.key_value_descr' (
114         buffer_VA => part_ID'address,
115         length    => part_ID'size/8));
116
117     -- if bytes_read /= parts_record'size/8 then
118     --   -- msg "Couldn't get record"
119     --   end if;
120
121   exception
122
123     when Record_AM.invalid_record_address =>
124
125       if msg_on_error then
126         Message_Services.Write_msg(
127           msg_id => not_on_file_code,
128           param1 => Incident_Defs.message_parameter(
129             typ => Incident_Defs.txt,
130             len => part_ID.length)' (
131             typ => Incident_Defs.txt,
132             len => part_ID.length,
133             txt_val => part_ID));
134         Message_Stack_Mgt.Push_msg_1_param(
135           not_on_file_code);
136       end if;
137
138       RAISE not_on_file;
139
140     when Record_AM.key_value_incomplete =>
141
142       if msg_on_error then
143         Message_Services.Write_msg(
144           msg_id => invalid_part_ID_code,
145           param1 => Incident_Defs.message_parameter(
146             typ => Incident_Defs.txt,
147             len => part_ID.length)' (
148             typ => Incident_Defs.txt,
149             len => part_ID.length,
150             txt_val => part_ID));
151         Message_Stack_Mgt.Clear_messages;

```

**PRELIMINARY**

```

152     Message_Stack Mgt.Push_msg_1_param(
153         message_id => invalid_part_ID_code,
154         param1    => Incident_Defs.message_parameter(
155             typ => Incident_Defs.txt,
156             len => part_ID.length)'(
157             typ    => Incident_Defs.txt,
158             len    => part_ID.length,
159             txt_val => part_ID));
160     end if;
161
162     RAISE invalid_part_ID;
163
164
165     when Record_AM.index_inconsistent =>
166
167         Message_Services.Write_msg(
168             msg_id => index_inconsistent_code,
169             param1 => Incident_Defs.message_parameter(
170                 typ => Incident_Defs.txt,
171                 len => parts_file_pathname.length)'(
172                 typ    => Incident_Defs.txt,
173                 len    => parts_file_pathname.length,
174                 txt_val => parts_file_pathname));
175
176         when others =>
177             RAISE;
178
179     end Read_parts_record;
180
181
182     procedure Write_parts_record(
183         parts_record: parts_record_type)
184     is
185
186     use System;    -- For "'size/8" constructions
187
188     begin
189
190         -- Write (insert in index key sequence) new record
191         -- into parts file:
192         --
193         Record_AM.Ops.Insert(
194             opened_dev => parts_file,
195             buffer_VA  => parts_record'address,
196             length     => parts_record'size/8);
197
198     exception
199     when Record_AM.invalid_duplicate =>
200         RAISE already_on_file;
201
202     when Record_AM.invalid_record_address |
203         Record_AM.key_value_incomplete =>
204         RAISE invalid_part_ID;
205
206     when Record_AM.index_inconsistent =>
207         Message_Services.Write_msg(
208             msg_id => index_inconsistent_code,
209             param1 => Incident_Defs.message_parameter(
210                 typ => Incident_Defs.txt,
211                 len => parts_file_pathname.length)'(
212                 typ    => Incident_Defs.txt,
213                 len    => parts_file_pathname.length,
214                 txt_val => parts_file_pathname));
215
216         when others =>
217             RAISE;
218
219     end Write_parts_record;
220
221
222     procedure Rewrite_parts_record(
223         parts_record: parts_record_type)
224     is
225
226     use System;    -- for "'size/8" constructions
227
228     begin

```

## PRELIMINARY

```

229
230
231 -- Rewrite (update) parts record:
232 --
233 Record_AM.Keyed_Ops.Update_by_key(
234     opened_dev => parts_file,
235     buffer_VA  => parts_record'address,
236     length     => parts_record'size/8,
237     index      => part_ID_index_name);
238
239 exception
240
241 when Record_AM.invalid_record_address =>
242     Message_Services.Write_msg(
243         msg_id => not_on_file_code,
244         param1 => Incident_Defs.message_parameter(
245             typ => Incident_Defs.txt,
246             len => part_ID_index_str.length)' (
247             typ      => Incident_Defs.txt,
248             len      => part_ID_index_str.length,
249             txt_val => part_ID_index_name));
250     RAISE not_on_file;
251
252 when Record_AM.key_value_incomplete =>
253     RAISE invalid_part_ID;
254
255 when Record_AM.index_inconsistent =>
256     Message_Services.Write_msg(
257         msg_id => index_inconsistent_code,
258         param1 => Incident_Defs.message_parameter(
259             typ => Incident_Defs.txt,
260             len => parts_file_pathname.length)' (
261             typ      => Incident_Defs.txt,
262             len      => parts_file_pathname.length,
263             txt_val => parts_file_pathname));
264
265     when others =>
266         RAISE;
267
268 end Rewrite_parts_record;
269
270
271 procedure Delete_parts_record(
272     part_ID: part_ID_type)
273 is
274
275 use System; -- for "'size/8" constructions
276
277 begin
278
279     -- Delete parts record:
280     --
281     Record_AM.Keyed_Ops.Delete_by_key(
282         opened_dev => parts_file,
283         index      => part_ID_index_name,
284         key_buffer => Record_AM.key_value_descr'(
285             buffer_VA => part_ID'address,
286             length    => part_ID'size/8));
287
288 exception
289
290 when Record_AM.invalid_record_address =>
291     RAISE not_on_file;
292
293 when Record_AM.key_value_incomplete =>
294     RAISE invalid_part_ID;
295
296 when Record_AM.index_inconsistent =>
297     Message_Services.Write_msg(
298         msg_id => index_inconsistent_code,
299         param1 => Incident_Defs.message_parameter(
300             typ => Incident_Defs.txt,
301             len => parts_file_pathname.length)' (
302             typ      => Incident_Defs.txt,
303             len      => parts_file_pathname.length,
304             txt_val => parts_file_pathname));
305

```

PRELIMINARY

```

306     when others =>
307         RAISE;
308
309     end Delete_parts_record;
310
311
312     procedure Close_parts_file
313     is
314
315     begin
316
317         if Record_AM.Ops.Is_open(parts_file) then
318             Record_AM.Ops.Close(
319                 opened_dev => parts_file);
320         end if;
321
322     end Close_parts_file;
323
324
325     -- Log file procedures:
326     -- Open / Write / Close log file
327
328     procedure Open_log_file
329     is
330         log_file_AD: System.untyped_word;
331
332     begin
333
334         -- Retrieve log file, if possible:
335         --
336         log_file_AD := Directory_Mgt.Retrieve(
337             log_file_pathname);
338
339         -- Check for access (modify) rights for log file:
340         --
341         if not Access_Mgt.Permits(
342             AD => log_file_AD,
343             rights => Object_Mgt.modify_rights)
344         then
345             Message_Services.Write_msg(
346                 msg_id => no_modify_rights_code,
347                 param1 => Incident_Defs.message_parameter(
348                     typ => Incident_Defs.txt,
349                     len => log_file_pathname.length)'(
350                     typ => Incident_Defs.txt,
351                     len => log_file_pathname.length,
352                     txt_val => log_file_pathname));
353         end if;
354
355         -- Open log file:
356         --
357         log_file := Record_AM.Ops.Open(
358             dev => Device_from_untyped_word(
359                 log_file_AD),
360             input_output => Device_Defs.inout,
361             allow => Device_Defs.nothing,
362             block => false);
363
364     exception
365         -- Exceptions from "Directory_Mgt.Retrieve",
366         -- "Record_AM.Ops.Open":
367         --
368         when others =>
369             Message_Services.Write_msg(
370                 msg_id => no_log_file_code,
371                 param1 => Incident_Defs.message_parameter(
372                     typ => Incident_Defs.txt,
373                     len => log_file_pathname.length)'(
374                     typ => Incident_Defs.txt,
375                     len => log_file_pathname.length,
376                     txt_val => log_file_pathname));
377     end Open_log_file;
378
379
380     procedure Write_log_record(
381         parts_record: parts_record_type;
382         action:       action_type)

```

PRELIMINARY

```
383 is
384   log_record: log_record_type;
385
386 use System;  -- for "'size/8" constructions
387
388 begin
389
390   log_record.part_ID := parts_record.part_ID;
391
392   log_record.action := action;
393
394   log_record.time := Timing_Conversions.
395     Convert_stu_to_numeric_time(
396     stu => Timed_Requests_Mgt.Get_time);
397
398   log_record.doc_number := System_Defs.text(doc_length)'
399     (doc_length, 0, (others => ' '));
400
401   log_record.qty := parts_record.qty_on_hand;
402
403   log_record.job_ID := System_Defs.text(job_length)'
404     (job_length, 0, (others => ' '));
405
406   log_record.supplier_ID :=
407     parts_record.suppliers(1);
408
409   Record_AM.Ops.Set_position(
410     opened_dev => log_file,
411     where      => Record_AM.record_specifier(
412       type_of_specifier => Record_AM.last)'(
413       type_of_specifier => Record_AM.last));
414
415   Record_AM.Ops.Insert(
416     opened_dev => log_file,
417     buffer_VA  => log_record'address,
418     length     => log_record'size/8);
419
420 end Write_log_record;
421
422
423
424 procedure Close_log_file
425 is
426
427 begin
428
429   if Record_AM.Ops.Is_open(log_file) then
430     Record_AM.Ops.Close(
431       opened_dev => log_file);
432   end if;
433
434 end Close_log_file;
435
436
437 end Inventory_Files;
```

## X-A.5.4 Inventory\_Forms Package Specification

```

1  with Device_Defs,
2      Form_Defs,
3      Incident_Defs,
4      Inventory_Files,
5      Inventory_Messages,
6      System,
7      System_Defs,
8      Terminal_Defs;
9
10 package Inventory_Forms is
11     --
12     -- Function:
13     --   Contains subprograms to display and process
14     --   Inventory Program forms.
15     --
16     --   Includes form handling routines
17     --   ("Process_*x*_form"), a form processing routine
18     --   ("Validate_cost"), and two key-catcher routines
19     --   ("Go_to_inquiry" and "Add_supplier_ID").
20     --
21     -- History:
22     --   07-06-87, William A. Rohm:   Written.
23     --   11-02-87, WAR:              Revised.
24     --
25     -- End of Header
26
27     -- Incident codes for messages:
28     --
29     module: constant := 5;
30     -- Message module index.
31
32     --*M*   set.language :language = English
33     --*M*   create.variable module :value = 5
34
35     invalid_output_device_code: constant
36         Incident_Defs.incident_code := (
37             message_object =>
38                 Inventory_Messages.message_object,
39             module          => module,
40             number          => 0,
41             severity        => Incident_Defs.error);
42
43     --*M*   store :module = 5 :number = 0\
44     --*M*         :msg_name = invalid_output_dev\
45     --*M*         :short = "Entered output device
46     --*M*           pathname '$pl<pathname>'
47     --*M*           does not exist, or does
48     --*M*           not support the record
49     --*M*           access method."
50
51
52     unit_cost_error_code: constant
53         Incident_Defs.incident_code := (
54             message_object =>
55                 Inventory_Messages.message_object,
56             module          => module,
57             number          => 1,
58             severity        => Incident_Defs.warning);
59
60     --*M*   store :module = 5 :number = 1\
61     --*M*         :msg_name = cost_error\
62     --*M*         :short = "Entered part's unit
63     --*M*           cost is not within
64     --*M*           $pl<allowed variation
65     --*M*           percentage>% of the average
66     --*M*           unit cost. Please re-enter
67     --*M*           $p2<total/unit> cost, or the
68     --*M*           number of units."
69
70
71     -- Constants:
72     --
73     inquiry_form_str: constant string :=
74         "/examples/inventory/forms/inquiry";

```

PRELIMINARY

```

75      -- String constant for inquiry form's
76      -- pathname.
77
78      inquiry_form_pathname: System_Defs.text(
79          inquiry_form_str'length) := (
80          inquiry_form_str'length,
81          inquiry_form_str'length,
82          inquiry_form_str);
83      -- Text constant from inquiry form's
84      -- pathname string.
85
86      receipts_form_str: constant string :=
87          "/examples/inventory/forms/receipts";
88      -- String constant for receipts form's
89      -- pathname.
90
91      receipts_form_pathname: System_Defs.text(
92          receipts_form_str'length) := (
93          receipts_form_str'length,
94          receipts_form_str'length,
95          receipts_form_str);
96      -- Text constant from receipts form's
97      -- pathname string.
98
99      update_form_str: constant string :=
100         "/examples/inventory/forms/update";
101      -- String constant for update form's
102      -- pathname.
103
104      update_form_pathname: System_Defs.text(
105          update_form_str'length) := (
106          update_form_str'length,
107          update_form_str'length,
108          update_form_str);
109      -- Text constant from update form's
110      -- pathname string.
111
112      report_form_str: constant string :=
113          "/examples/inventory/forms/report";
114      -- String constant for report form's
115      -- pathname.
116
117      report_form_pathname: System_Defs.text(
118          report_form_str'length) := (
119          report_form_str'length,
120          report_form_str'length,
121          report_form_str);
122      -- Text constant from report form's
123      -- pathname string.
124
125
126      -----
127      -- Field and subform names for forms:
128      --
129      part_ID_field:      System_Defs.text( 7) := (
130          7,7, "part_ID");
131      desc_field:        System_Defs.text(11) := (
132          11,11,"description");
133      unit_field:        System_Defs.text( 4) := (
134          4,4, "unit");
135      loc_field:         System_Defs.text( 8) := (
136          8,8, "location");
137      qty_field:         System_Defs.text(11) := (
138          11,11,"qty_on_hand");
139      reorder_pt_field:  System_Defs.text(13) := (
140          13,13,"reorder_point");
141      reorder_qty_field: System_Defs.text(11) := (
142          11,11,"reorder_qty");
143      suppliers_field:   System_Defs.text( 9) := (
144          9,9, "suppliers");
145      usage_tmo_field:   System_Defs.text(16) := (
146          16,16,"usage_this_month");
147      usage_lmo_field:   System_Defs.text(16) := (
148          16,16,"usage_last_month");
149      usage_lyr_field:   System_Defs.text(15) := (
150          15,15,"usage_last_year");
151      avg_cost_field:    System_Defs.text(13) := (

```

PRELIMINARY

```

152     13,13,"avg_unit_cost");
153 last_cost_field: System_Defs.text(14) := (
154     14,14,"last_unit_cost");
155 date_first_field: System_Defs.text(14) := (
156     14,14,"date_first_act");
157 date_last_field: System_Defs.text(13) := (
158     13,13,"date_last_act");
159 status_field: System_Defs.text( 6) := (
160     6,6, "status");
161
162 inq_suppl_ref_field: System_Defs.text(19) := (
163     19,19,"supplier_ref_number");
164 inq_date_field: System_Defs.text( 4) := (
165     4,4, "date");
166 inq_time_field: System_Defs.text( 4) := (
167     4,4, "time");
168
169 rpt_type_field: System_Defs.text(11) := (
170     11,11,"report_type");
171 rpt_opt_field: System_Defs.text(14) := (
172     14,14,"report_options");
173 rpt_dev_field: System_Defs.text(20) := (
174     20,20,"report_output_device");
175
176
177 -- Group and subform names for forms:
178 --
179 inq_part_ID_only: System_Defs.text(16) := (
180     16,16,"inq_part_ID_only");
181 inq_all: System_Defs.text(15) := (
182     15,15,"inq_display_all");
183
184 update_add: System_Defs.text(10) := (
185     10,10,"update_add");
186 update_change: System_Defs.text(13) := (
187     13,13,"update_change");
188 update_delete: System_Defs.text(13) := (
189     13,13,"update_delete");
190
191 --
192 -----
193
194
195 -- Types:
196 --
197 subtype percentage is System.short_ordinal
198     range 0..99;
199
200 type percentage_range_type is
201     -- Type for containing percentage range.
202     record
203         percent_less: percentage;
204         -- Maximum percent of change less than
205         -- reference value.
206         percent_more: percentage;
207         -- Maximum percent of change more than
208         -- reference value.
209     end record;
210
211
212 procedure Process_inquiry_form;
213     -- Function:
214     -- Processes inquiry form: displays form in
215     -- main window, gets valid information
216     -- ("part_ID"), then reads Parts Master File and
217     -- displays parts record.
218
219
220 procedure Process_receipts_form;
221     -- Function:
222     -- Processes receipts form: displays form in
223     -- main window, gets valid information
224     -- ("part_ID", "supplier", "quantity", etc),
225     -- reads Parts Master File to validate, updates
226     -- parts record, then writes log file record.
227
228

```

PRELIMINARY

```

229 procedure Process_update_form(
230     selection: Terminal_Defs.menu_item_ID);
231     -- Selection made in the "Maintenance" menu;
232     -- either *Add*, *Change*, or *Delete*.
233     -- Function:
234     -- Processes update form: displays form in main
235     -- window, gets valid information ("part_ID"),
236     -- reads Parts Master File and displays parts
237     -- record, then updates or deletes part record.
238
239
240 procedure Process_report_form(
241     report_by_part: boolean;
242     -- True if the report is to be "by part",
243     -- false if the report is "by location".
244     report_out_dev: out System_Defs.text);
245     -- Variable that receives output device
246     -- pathname where report is to be sent.
247     --
248     -- Function:
249     -- Processes report form: displays form in main
250     -- window, gets report output device.
251
252
253 -- Form processing & key catcher routines:
254 --
255
256 procedure Validate_cost(
257     old_parts_record:
258         Inventory_Files.parts_record_type;
259     -- Parts record from file.
260     qty_received:
261         Inventory_Files.qty_type;
262     -- Entered quantity received.
263     total_cost: in out
264         Inventory_Files.cost_type;
265     -- Entered or calculated total cost.
266     unit_cost: in out
267         Inventory_Files.cost_type;
268     -- Entered or calculated unit cost.
269     total: boolean := true;
270     -- Whether to calculate the "total_cost" from
271     -- the "unit_cost", or vice versa.
272     --
273     -- If true (default), the "total_cost" is
274     -- calculated from the given "unit_cost" times
275     -- the given "qty_received". If false, the
276     -- "unit_cost" is calculated by dividing the
277     -- given "total_cost" by the given
278     -- "qty_received".
279     percentage_range:
280         percentage_range_type := (5, 5);
281     -- Maximum low and high percentage
282     -- difference between parts record's
283     -- "avg_unit_cost" (also required of
284     -- "last_unit_cost") and the entered or
285     -- calculated "unit_cost" parameter.
286     valid: out boolean);
287     -- Whether the entered or calculated unit cost
288     -- is within the given "percentage_range" of
289     -- cost on file.
290     --
291     -- Function:
292     -- Processing routine called from the Receipts
293     -- form to validate unit cost and to calculate and
294     -- return either total cost or unit cost.
295
296
297 procedure Go_to_inquiry;
298 --
299 -- Function:
300 -- Key catcher called from the "Receipts" or
301 -- "Change" form when the user presses the
302 -- "<Go-to-Inquiry>" key. Calls
303 -- "Process_inquiry_form".
304 --
305 -- When this procedure (key-catcher) is activated,

```

## PRELIMINARY

```
306 -- the enclosing form has been suspended. When
307 -- this procedure returns, the enclosing form
308 -- continues.
309
310
311 procedure Add_supplier_ID(
312     opened_form: Form_Defs.opened_form_AD);
313     -- Opened form to which another "supplier_ID"
314     -- field will be added.
315     --
316     -- Function:
317     -- Key catcher called from the "Add" form when the
318     -- user presses the "<next>" key. Adds another
319     -- "supplier_ID" field to current form, up to a
320     -- total of three.
321
322 end Inventory_Forms;
323
```

## X-A.5.5 Inventory\_Forms Package Body

```

1  with Data_Definition_Mgt,
2     Device_Defs,
3     Directory_Mgt,
4     Form_Defs,
5     Form_Handler,
6     Inventory_Files,
7     Inventory_Menus,
8     Inventory_Windows,
9     Message_Services,
10    Record_AM,
11    System,
12    System_Defs,
13    Terminal_Defs,
14    Text_Mgt,
15    Timed_Requests_Mgt,
16    Timing_Conversions,
17    Unchecked_Conversion,
18    Window_Services;
19
20
21 package body Inventory_Forms is
22
23
24     function Get_form(
25         form_pathname: System_Defs.text)
26         return Form_Defs.opened_form_AD
27     is
28         --
29         -- Logic:
30         -- Gets requested form from directory, opens
31         -- form.
32
33         -- Generic function:
34         --
35         function DDef_from_untyped is new
36             Unchecked_conversion(
37                 source => System.untyped_word,
38                 target => Data_Definition_Mgt.DDef_AD);
39
40         opened_form: Form_Defs.opened_form_AD;
41         -- Returned opened form's AD.
42
43     begin
44         opened_form := Form_Handler.Open_form(
45             DDef => DDef_from_untyped(
46                 Directory_Mgt.Retrieve(
47                     name => form_pathname)));
48
49         return opened_form;
50
51     end Get_form;
52
53
54     procedure Process_inquiry_form
55         --
56         -- Logic:
57         -- 1. Display form in main window
58         -- 2. Get valid information ("part ID")
59         -- 3. Read Parts Master File and display parts
60         -- record
61
62     is
63
64         opened_form: Form_Defs.opened_form_AD;
65         form_status: Form_Defs.status_t;
66
67         opened_record_form: Device_Defs.opened_device;
68         -- For record access to "opened_form".
69
70         part_ID:      Inventory_Files.part_ID_type;
71         parts_record: Inventory_Files.parts_record_type;
72
73         length:      System.ordinal;
74         empty:       boolean;

```

## PRELIMINARY

```

75     error:          boolean;
76
77     first_time:     boolean := true;
78
79     use Form_Defs;  -- import "/"= for type
80                   -- "Form_Defs.status_t"
81
82     use System;    -- for "'size/8" arithmetic
83
84     begin
85
86         opened_form := Get_form(inquiry_form_pathname);
87
88         -- Open form's DDef for record access:
89         --
90         opened_record_form := Record_AM.Open_by_name(
91             name           => inquiry_form_pathname,
92             input_output => Device_Defs.inout);
93
94         -- Set up first rank (group) in "inquiry form"
95         -- pile:
96         --
97         Form_Handler.Create_group_instances(
98             opened_form_a => opened_form,
99             group         => inq_part_ID_only,
100            number_of_instances => 1);
101
102
103         -- Read part ID, display, ask for another:
104         --
105         loop
106
107             -- Get first part ID:
108             --
109             form_status := Form_Handler.Get(
110                 opened_form_a => opened_form,
111                 opened_window_a => Inventory_Windows.
112                     main_window);
113
114             if form_status /= Form_Defs.finished then
115
116                 -- some kind of error processing
117                 null;
118
119             else
120                 Form_Handler.Fetch_value(
121                     opened_form_a => opened_form,
122                     element       => part_ID_field,
123                     subunit       => System_Defs.null_text,
124                     -- added subunit; value correct?
125                     value_buffer_VA => part_ID'address,
126                     value_length  => part_ID'size/8,
127                     value_t       =>
128                         Data_Definition_Mgt.t_string,
129                     element_value_length => length,
130                     empty        => empty);
131
132                 if empty then -- user entered null part ID:
133                     -- exit loop; return to menu
134                     EXIT;
135                 end if;
136
137                 -- Read parts file, handle exceptions:
138                 --
139                 begin
140
141                     Inventory_Files.Read_parts_record(
142                         part_ID      => part_ID,
143                         msg_on_error => true,
144                         parts_record => parts_record);
145
146                     if first_time then
147                         -- set up other rank
148
149                         first_time := false;
150
151                         -- Remove first group (rank):

```

PRELIMINARY

```

152      --
153      Form_Handler.Remove_group_instances(
154          opened_form_a    => opened_form,
155          group            => inq_part_ID_only,
156          number_of_instances => 1);
157
158      -- Add second group (rank):
159      --
160      Form_Handler.Create_group_instances(
161          opened_form_a    => opened_form,
162          group            => inq_all,
163          number_of_instances => 1);
164
165      end if; -- If "first_time" through
166
167      Record_AM.Ops.Update(
168          opened_dev => opened_record_form,
169          buffer_VA => parts_record'address,
170          length    => parts_record'size/8);
171
172      exception
173      when Inventory_Files.not_on_file =>
174          null; -- "Record not found" message
175              -- has been displayed; go
176              -- through loop again
177
178      when Inventory_Files.invalid_part_ID =>
179          null; -- "Invalid part ID entered"
180              -- message has been displayed;
181              -- go through loop again
182
183      end;
184
185      end if; -- if form status = finished
186
187      end loop; -- read part_ID, display loop
188
189      Form_Handler.Clear(
190          opened_form_a => opened_form);
191
192      Form_Handler.Close_form(
193          opened_form_a => opened_form);
194
195      -- Close record access to form:
196      --
197      Record_AM.Ops.Close(
198          opened_dev => opened_record_form);
199
200      end Process_inquiry_form;
201
202
203      procedure Process_receipts_form
204      --
205      -- Logic:
206      -- 1. Display form in main window
207      -- 2. Get receipt information ("part_ID",
208      --    "supplier", etc)
209      -- 3. Read Parts Master File to validate
210      -- 4. If valid, update parts record, then write
211      --    log file record.
212
213      is
214
215          opened_form: Form_Defs.opened_form_AD;
216          form_status: Form_Defs.status_t;
217
218          part_ID:      Inventory_Files.part_ID_type;
219          parts_record: Inventory_Files.parts_record_type;
220
221          length:      System.ordinal;
222          empty, error: boolean;
223
224          now:         Timing_Conversions.numeric_time;
225
226      use Form_Defs; -- import "/"= for type
227                  -- "Form_Defs.status_t"
228

```

```

229 use System;      -- for "'size/8" arithmetic
230
231 begin
232
233   opened_form := Get_form(receipts_form_pathname);
234
235   loop
236
237     form_status := Form_Handler.Get (
238       opened_form_a => opened_form,
239       opened_window_a => Inventory_Windows.
240         main_window);
241
242     if form_status /= Form_Defs.finished then
243
244       -- Some kind of error processing
245       null;
246
247     else
248
249       Form_Handler.Fetch_value(
250         opened_form_a => opened_form,
251         element       => part_ID_field,
252         subunit       => System_Defs.null_text,
253         -- added subunit; value correct?
254         value_buffer_VA => part_ID'address,
255         value_length   => part_ID'size/8,
256         value_t        =>
257           Data_Definition_Mgt.t_string,
258         element_value_length => length,
259         empty          => empty);
260
261     if empty then
262       -- null part_ID; return to menu
263
264       EXIT;
265     end if;
266
267     begin
268
269       Inventory_Files.Read_parts_record(
270         part_ID    => part_ID,
271         msg_on_error => true,
272         parts_record => parts_record);
273
274
275       Form_Handler.Store_value(
276         opened_form_a => opened_form,
277         element       => desc_field,
278         subunit       => System_Defs.null_text,
279         -- added subunit; value correct?
280         value_buffer_VA =>
281           parts_record.desc'address,
282         value_length   =>
283           parts_record.desc'size/8,
284         value_t        =>
285           Data_Definition_Mgt.t_string);
286
287       Form_Handler.Store_value(
288         opened_form_a => opened_form,
289         element       => unit_field,
290         subunit       => System_Defs.null_text,
291         -- added subunit; value correct?
292         value_buffer_VA =>
293           parts_record.unit'address,
294         value_length   =>
295           parts_record.unit'size/8,
296         value_t        =>
297           Data_Definition_Mgt.t_string);
298
299       now := Timing_Conversions.
300         Convert_stu_to_numeric_time(
301           stu => Timed_Requests_Mgt.Get_time);
302
303
304       Form_Handler.Store_value(
305         opened_form_a => opened_form,

```

PRELIMINARY

```

306         element      => inq_date_field,
307         subunit      => System_Defs.null_text,
308         -- added subunit; value correct?
309         value_buffer_VA => now'address,
310         value_length  => now'size/8,
311         value_t       =>
312             Data_Definition_Mgt.t_date);
313
314     Form_Handler.Store_value(
315         opened_form_a => opened_form,
316         element      => inq_time_field,
317         subunit      => System_Defs.null_text,
318         -- added subunit; value correct?
319         value_buffer_VA => now'address,
320         value_length  => now'size/8,
321         value_t       =>
322             Data_Definition_Mgt.t_date);
323
324     exception
325
326         when Inventory_Files.not_on_file =>
327             null; -- "Record not found" message
328             -- has been displayed; go
329             -- through loop again
330
331         when Inventory_Files.invalid_part_ID =>
332             null; -- "Invalid part ID entered"
333             -- message has been displayed;
334             -- go through loop again
335
336     end; -- Read parts record block
337
338 end if; -- if form status = finished
339
340 end loop;
341
342 Form_Handler.Clear(
343     opened_form_a => opened_form);
344
345 Form_Handler.Close_form(
346     opened_form_a => opened_form);
347
348 end Process_receipts_form;
349
350
351 procedure Process_update_form(
352     selection: Terminal_Defs.menu_item_ID)
353 --
354 -- Logic:
355 -- 1. Get update form and create appropriate
356 -- subform
357 -- 2. Get "part_ID"
358 -- 3. Read Parts Master File and display parts
359 -- record
360 -- 4. Add, change, or delete part record
361 -- 5. Write appropriate log record
362
363 is
364
365     opened_form: Form_Defs.opened_form_AD;
366     -- AD to opened "update" form.
367
368     form_status: Form_Defs.status_t;
369
370     part_ID: Inventory_Files.part_ID_type;
371     parts_record:
372         Inventory_Files.parts_record_type;
373     new_parts_record:
374         Inventory_Files.parts_record_type;
375     log_record: Inventory_Files.log_record_type;
376
377     opened_record_form: Device_Defs.opened_device;
378     -- For record access to "opened_form".
379
380     length: System.ordinal;
381     -- Length of a returned record, in bytes.
382     empty: boolean;

```

```

383         -- Whether the entered "part_ID" field was
384         -- empty.
385
386     new_part:    boolean;
387     -- True if this is a new part ID (add only!).
388
389     use Form_Defs;    -- to import "/"= for
390                     -- Form_Defs.status_t
391
392     use System;      -- for "'size/8" arithmetic
393
394     begin
395
396         -- Open "update" form:
397         --
398         opened_form := Get_form(
399             update_form_pathname);
400
401         -- Create appropriate group instance
402         -- (add, change, delete):
403         --
404         case selection is
405
406             when Inventory_Menus.update_add_item =>
407
408                 Form_Handler.Create_group_instances(
409                     opened_form_a => opened_form,
410                     group          => update_add,
411                     number_of_instances => 1);
412
413             when Inventory_Menus.update_change_item =>
414                 Form_Handler.Create_group_instances(
415                     opened_form_a => opened_form,
416                     group          => update_change,
417                     number_of_instances => 1);
418
419             when Inventory_Menus.update_delete_item =>
420                 Form_Handler.Create_group_instances(
421                     opened_form_a => opened_form,
422                     group          => update_delete,
423                     number_of_instances => 1);
424
425             when others =>
426                 null;
427
428         end case;
429
430         -- Open form's DDef for record access:
431         --
432         opened_record_form := Record_AM.Open_by_name(
433             name          => update_form_pathname,
434             input_output => Device_Defs.inout);
435
436     loop
437         -- Get a part ID:
438         --
439         form_status := Form_Handler.Get(
440             opened_form_a => opened_form,
441             opened_window_a =>
442                 Inventory_Windows.main_window);
443
444         if form_status /= Form_Defs.finished then
445
446             -- Some kind of error processing
447             null;
448
449         else
450
451             Form_Handler.Fetch_value(
452                 opened_form_a => opened_form,
453                 element       => part_ID_field,
454                 subunit       => System_Defs.null_text,
455                 -- added subunit; value correct?
456                 value_buffer_VA => part_ID'address,
457                 value_length   => part_ID'size/8,
458                 value_t        =>
459                     Data_Definition_Mgt.t_string,

```

PRELIMINARY

```

460         element_value_length => length,
461         empty                 => empty);
462
463     if empty then
464         EXIT;           -- exit loop
465     else
466         begin
467             -- Get parts record, if possible:
468             --
469             new_part := false;
470
471             Inventory_Files.Read_parts_record(
472                 part_ID => part_ID,
473                 parts_record => parts_record);
474
475             Record_AM.Ops.Update(
476                 opened_dev => opened_record_form,
477                 buffer_VA => parts_record'address,
478                 length     => parts_record'size/8);
479
480         exception
481             when Inventory_Files.not_on_file =>
482                 new_part := true;
483
484             when Inventory_Files.invalid_part_ID =>
485                 null; -- "Invalid part ID
486                     -- entered" message has
487                     -- been displayed; go
488                     -- through loop again
489
490         end;
491
492     case selection is
493     when Inventory_Menus.update_add_item =>
494         if new_part then
495             length := Record_AM.Ops.Read(
496                 opened_dev => opened_record_form,
497                 buffer_VA => parts_record'address,
498                 length     => parts_record'size/8);
499
500             Inventory_Files.Write_parts_record(
501                 parts_record => parts_record);
502
503             -- Create and write log record:
504             --
505             Inventory_Files.Write_log_record(
506                 parts_record => parts_record,
507                 action       =>
508                     Inventory_Files.create);
509
510             end if;
511
512     when Inventory_Menus.update_change_item =>
513         length := Record_AM.Ops.Read(
514             opened_dev => opened_record_form,
515             buffer_VA =>
516                 new_parts_record'address,
517             length     =>
518                 new_parts_record'size/8);
519
520         Inventory_Files.Rewrite_parts_record(
521             parts_record => parts_record);
522
523         -- Create and write log record:
524         --
525         Inventory_Files.Write_log_record(
526             parts_record => parts_record,
527             action       =>
528                 Inventory_Files.update);
529
530     when Inventory_Menus.update_delete_item =>
531
532         Inventory_Files.Delete_parts_record(
533             part_ID => part_ID);
534
535         -- Create and write log record:
536         --

```

```

537         Inventory_Files.Write_log_record(
538             parts_record => parts_record,
539             action      =>
540                 Inventory_Files.delete);
541
542         when others =>
543             null;
544
545         end case;
546
547         end if;      -- if not empty part ID
548
549         end if;      -- if form finished
550
551     end loop;
552
553     Form_Handler.Clear(
554         opened_form_a => opened_form);
555
556     Form_Handler.Close_form(
557         opened_form_a => opened_form);
558
559     -- Close record access to form:
560     --
561     Record_AM.Ops.Close(
562         opened_dev => opened_record_form);
563
564 end Process_update_form;
565
566
567 procedure Process_report_form(
568     report_by_part:      boolean;
569     -- True if by part, false if by location.
570     report_out_dev: out System_Defs.text)
571     -- Returned output device's pathname,
572     -- "System_Defs.null_text" if error.
573     --
574     -- Logic:
575     -- 1. Open report form
576     -- 2. Get report options and output device
577     -- 3. Attempt opening and closing report
578     --    output device
579     -- 4. Clear and close form
580     -- 5. If any error occurred, return
581     --    "report_out_dev" as "null_text"
582 is
583
584     opened_form: Form_Defs.opened_form_AD;
585     form_status: Form_Defs.status_t;
586
587     length:      System.ordinal;
588     empty:       boolean;
589
590     report_options: System.ordinal;
591     -- Report options field value.
592
593     valid: boolean;
594     -- Whether the report information is valid.
595
596     test_out_dev: System_Defs.text(Incident_Defs.txt_length);
597     -- Entered report output device pathname to be
598     -- checked.
599
600     test_opened_dev: Device_Defs.opened_device;
601     -- Opened device returned from
602     -- "Record_AM.Open" (test to see if
603     -- entered device pathname is valid).
604
605 use Form_Defs;      -- import "/"= for type
606                    -- "Form_Defs.status_t"
607
608 use System;        -- for "'size/8" arithmetic
609
610 begin
611
612     opened_form := Get_form(report_form_pathname);
613

```

PRELIMINARY

```

614 form_status := Form_Handler.Get (
615     opened_form_a => opened_form,
616     opened_window_a => Inventory_Windows.
617         main_window);
618
619 if form_status /= Form_Defs.finished then
620
621     -- some kind of error processing
622     null;
623
624 else
625
626     Form_Handler.Fetch_value (
627         opened_form_a      => opened_form,
628         element            => rpt_type_field,
629         subunit            => System_Defs.null_text,
630         -- added subunit; value correct?
631         value_buffer_VA    =>
632             report_by_part'address,
633         value_length       => report_by_part'size/8,
634         value_t            =>
635             Data_Definition_Mgt.t_boolean,
636         element_value_length => length,
637         empty              => empty);
638
639     valid := not empty;
640
641
642     Form_Handler.Fetch_value (
643         opened_form_a      => opened_form,
644         element            => rpt_opt_field,
645         subunit            => System_Defs.null_text,
646         -- added subunit; value correct?
647         value_buffer_VA    =>
648             report_options'address,
649         value_length       => report_options'size/8,
650         value_t            =>
651             Data_Definition_Mgt.t_ord4,
652         element_value_length => length,
653         empty              => empty);
654
655
656     valid := valid and (not empty);
657
658     Form_Handler.Fetch_value (
659         opened_form_a      => opened_form,
660         element            => rpt_dev_field,
661         subunit            => System_Defs.null_text,
662         -- added subunit; value correct?
663         value_buffer_VA    =>
664             test_out_dev'address,
665         value_length       => test_out_dev'size/8,
666         value_t            =>
667             Data_Definition_Mgt.t_string,
668         element_value_length => length,
669         empty              => empty);
670
671     valid := valid and (not empty);
672
673
674     -- Try to open device at the new pathname:
675     --
676     begin
677
678         test_opened_dev := Record_AM.Open_by_name (
679             name          => test_out_dev,
680             input_output => Device_Defs.output);
681
682         Record_AM.Ops.Close (
683             opened_dev => test_opened_dev);
684
685     exception
686
687     when others =>
688         valid := false;
689
690         Message_Services.Write_msg(

```

**PRELIMINARY**

```

691         msg_id => invalid_output_device_code,
692         param1 => Incident_Defs.message_parameter(
693             typ => Incident_Defs.txt,
694             len => test_out_dev.length)' (
695             typ      => Incident_Defs.txt,
696             len      => test_out_dev.length,
697             txt_val => test_out_dev));
698
699     end; -- test open
700
701 end if; -- if form status = finished
702
703 Form_Handler.Clear(
704     opened_form_a => opened_form);
705
706 Form_Handler.Close_form(
707     opened_form_a => opened_form);
708
709 if valid then
710     report_out_dev := test_out_dev;
711 else
712     report_out_dev := System_Defs.null_text;
713 end if;
714
715 end Process_report_form;
716
717
718 -- Form Processing Routine & Key Catchers:
719 --
720
721 procedure Validate_cost(
722     old_parts_record:
723         Inventory_Files.parts_record_type;
724     qty_received:
725         Inventory_Files.qty_type;
726     total_cost:      in out
727         Inventory_Files.cost_type;
728     unit_cost:      in out
729         Inventory_Files.cost_type;
730     total:          boolean := true;
731     percentage_range:
732         percentage_range_type := (5, 5);
733     valid:          out boolean)
734 --
735 -- Logic:
736 -- Called from the Receipts form to validate unit
737 -- cost and to calculate and return either total
738 -- cost or unit cost.
739
740 is
741
742     max_cost, min_cost: float;
743
744 use System; -- to import "*" and "/"
745
746 begin
747
748     -- Calculate total or unit cost:
749     --
750     if total then
751         total_cost := float(unit_cost) *
752             float(qty_received);
753     else
754         unit_cost := float(total_cost) /
755             float(qty_received);
756     end if;
757
758     -- Calculate minimum and maximum acceptable unit
759     -- costs:
760     --
761     min_cost := float(old_parts_record.avg_unit_cost) *
762         (1.0 - float(percentage_range.percent_less)
763             / 100.0);
764
765     max_cost := float(old_parts_record.avg_unit_cost) *
766         (1.0 + float(percentage_range.percent_less)
767             / 100.0);

```

```

768
769     -- Check unit_cost against average cost:
770     --
771     valid := (unit_cost >= min_cost) and
772             (unit_cost <= max_cost);
773
774 end Validate_cost;
775
776
777
778 procedure Go_to_inquiry
779 --
780 -- Logic:
781 --   Called from the "Receipts" or "Change" form.
782 --
783 --   Calls "Process_inquiry_form". Enclosing
784 --   (calling) form is suspended during key-catcher
785 --   call, resumed upon return from this procedure.
786 is
787
788 begin
789     Process_inquiry_form;
790
791 end Go_to_inquiry;
792
793
794
795 procedure Add_supplier_ID(
796     opened_form: Form_Defs.opened_form_AD)
797 --
798 -- Logic:
799 --   Called from the "Add" form.
800 --
801 --   Calls "Process_inquiry_form". Enclosing
802 --   (calling) form is suspended during key-catcher
803 --   call, resumed upon return from this procedure.
804 is
805
806 begin
807
808     begin
809         -- Add another instance of the supplier ID group.
810         Form_Handler.Create_group_instances(
811             opened_form_a => opened_form,
812             group          => suppliers_field,
813             number_of_instances => 1);
814
815     exception
816         when Form_Handler.maximum_number_reached => null;
817
818     end;
819
820 end Add_supplier_ID;
821
822 end Inventory_Forms;
823

```

## X-A.5.6 Inventory\_Menus Package Specification

```

1  with Device_Defs,
2      Incident_Defs,
3      Inventory_Messages,
4      System,
5      System_Defs,
6      Terminal_Defs,
7      Window_Services;
8
9  package Inventory_Menus is
10     --
11     -- Function:
12     --   Contains subprograms to install and process
13     --   Inventory Example Program menus.
14     --
15     --   This package contains the routines which
16     --   perform each menu's selection actions. Some of
17     --   the menu selections require calls to the
18     --   "Inventory_Forms" and "Inventory_Reports"
19     --   packages.
20     --
21     -- History:
22     --   05-18-87, William A. Rohm:  Written.
23     --   10-27-87, WAR:             Revised.
24     --
25     -- End of Header
26
27
28     -- Incident codes for messages:
29     --
30     module: constant := 4;
31     -- Message module index.
32
33     -- *M*   set.language :language = English
34     -- *M*   create.variable module :value = 4
35
36     unable_to_install_code: constant
37         Incident_Defs.incident_code := (
38             message_object =>
39                 Inventory_Messages.message_object,
40             module          => module,
41             number          => 0,
42             severity        =>
43                 Incident_Defs.error);
44
45     -- *M*   store :module = $module :number = 0\
46     -- *M*       :msg_name = unable_install \
47     -- *M*       :short = "Unable to install menus."
48
49
50     no_selection_code: constant
51         Incident_Defs.incident_code := (
52             message_object =>
53                 Inventory_Messages.message_object,
54             module          => module,
55             number          => 1,
56             severity        =>
57                 Incident_Defs.warning);
58
59     -- *M*   store :module = $module :number = 1\
60     -- *M*       :msg_name = no_selection\
61     -- *M*       :short = "Selection $pl<selection
62     -- *M*       number> is not implemented."
63
64
65     menu_group_DDef_path:
66         System_Defs.text(34) := (34,34,
67             "/examples/inventory/DDef/menu_DDef");
68     -- Pathname of stored menu group DDef.
69
70     menu_group_DDef_root_name:
71         System_Defs.text(4) := (4,4,"root");
72     -- Pathname of menu group DDef's root node.
73
74     inv_menu_group_ID: constant

```

PRELIMINARY

```

75     Terminal_Defs.menu_group_ID := 1;
76     -- Inventory menu group's ID.
77
78
79 -- Menu IDs
80     inquiry_menu_ID:      constant
81         Terminal_Defs.menu_ID := 1;
82
83     posting_menu_ID:      constant
84         Terminal_Defs.menu_ID := 2;
85
86     update_menu_ID:       constant
87         Terminal_Defs.menu_ID := 3;
88
89     report_menu_ID:       constant
90         Terminal_Defs.menu_ID := 4;
91
92     housekeeping_menu_ID: constant
93         Terminal_Defs.menu_ID := 5;
94
95     exit_menu_ID:         constant
96         Terminal_Defs.menu_ID := 6;
97
98 -- Inquiry menu items
99     inq_by_part_item:      constant
100         Terminal_Defs.menu_item_ID := 1;
101     inq_by_desc_item:      constant
102         Terminal_Defs.menu_item_ID := 2;
103     inq_exit_item:        constant
104         Terminal_Defs.menu_item_ID := 3;
105
106
107 -- Posting menu items
108     post_receipt_item:     constant
109         Terminal_Defs.menu_item_ID := 1;
110     post_issue_item:       constant
111         Terminal_Defs.menu_item_ID := 2;
112     post_return_item:      constant
113         Terminal_Defs.menu_item_ID := 3;
114     post_spoilage_item:    constant
115         Terminal_Defs.menu_item_ID := 4;
116     post_journal_item:     constant
117         Terminal_Defs.menu_item_ID := 5;
118     post_exit_item:        constant
119         Terminal_Defs.menu_item_ID := 6;
120
121
122 -- Update menu items
123     update_add_item:       constant
124         Terminal_Defs.menu_item_ID := 1;
125     update_change_item:    constant
126         Terminal_Defs.menu_item_ID := 2;
127     update_delete_item:    constant
128         Terminal_Defs.menu_item_ID := 3;
129     update_exit_item:      constant
130         Terminal_Defs.menu_item_ID := 4;
131
132
133 -- Report menu items
134     report_by_part_item:   constant
135         Terminal_Defs.menu_item_ID := 1;
136     report_by_location_item: constant
137         Terminal_Defs.menu_item_ID := 2;
138     report_exit_item:      constant
139         Terminal_Defs.menu_item_ID := 3;
140
141
142 -- Housekeeping menu items
143     hskpg_index_item:      constant
144         Terminal_Defs.menu_item_ID := 1;
145     hskpg_exit_item:       constant
146         Terminal_Defs.menu_item_ID := 2;
147
148
149     procedure Set_up_menu_group;
150     --
151     -- Function:

```

## PRELIMINARY

```
152      -- Retrieve Inventory Example Program's menu
153      -- group description (*a menu DDef*), then
154      -- install and enable the menu group in the main
155      -- window.
156
157
158      -- Menu selection processing procedures:
159      -- Inquiry / Posting / Update / Report / Housekeeping
160      --
161
162      procedure Process_inquiry_menu(
163          selection: Terminal_Defs.menu_item_ID);
164          -- Selection made in this menu.
165      --
166      -- Function:
167      -- Processes selections from the Inquiry menu.
168
169
170
171      procedure Process_posting_menu(
172          selection: Terminal_Defs.menu_item_ID);
173          -- Selection made in this menu.
174      --
175      -- Function:
176      -- Processes selections from the Posting menu.
177
178
179
180      procedure Process_update_menu(
181          selection: Terminal_Defs.menu_item_ID);
182          -- Selection made in this menu.
183      --
184      -- Function:
185      -- Processes selections from the Update menu.
186
187
188
189      procedure Process_report_menu(
190          selection: Terminal_Defs.menu_item_ID);
191          -- Selection made in this menu.
192      --
193      -- Function:
194      -- Processes selections from the Report menu.
195
196
197      procedure Process_housekeeping_menu(
198          selection: Terminal_Defs.menu_item_ID);
199          -- Selection made in this menu.
200      --
201      -- Function:
202      -- Processes selections from the Housekeeping
203      -- menu.
204
205      end Inventory_Menus;
```

## X-A.5.7 Inventory\_Menus Package Body

```

1  with Data_Definition_Mgt,
2      Device_Defs,
3      Directory_Mgt,
4      File_Admin,
5      File_Defs,
6      Incident_Defs,
7      Inventory_Files,
8      Inventory_Forms,
9      Inventory_Messages,
10     Inventory_Reports,
11     Inventory_Windows,
12     Message_Services,
13     Record_AM,
14     System_Defs,
15     Terminal_Defs,
16     Unchecked_Conversion,
17     Window_Services;
18
19  package body Inventory_Menus is
20
21     -- Generic function:
22     --
23     function DDef_from_untyped is new
24         Unchecked_conversion(
25             source => System.untyped_word,
26             target => Data_Definition_Mgt.DDef_AD);
27
28     -- Variables:
29     --
30     menu_group_DDef_AD: Data_Definition_Mgt.DDef_AD;
31     -- AD to stored menu group DDef.
32
33     menu_group_node:
34         Data_Definition_Mgt.node_reference;
35     -- Node reference to stored menu group DDef.
36
37
38     procedure Set_up_menu_group
39
40     is
41
42     begin
43
44         -- Retrieve menu group's DDef:
45         --
46         menu_group_DDef_AD := DDef_from_untyped(
47             Directory_Mgt.Retrieve(
48                 name => menu_group_DDef_path));
49
50
51         -- Retrieve menu group's root node:
52         --
53         menu_group_node := Data_Definition_Mgt.
54             Retrieve_DDef(
55                 DDef => menu_group_DDef_AD,
56                 name => menu_group_DDef_root_name);
57
58
59         -- Install menu group:
60         --
61         Window_Services.Ops.Install_menu_group(
62             window => Inventory_Windows.
63                 main_window,
64             menu_group => menu_group_node,
65             ID => inv_menu_group_ID);
66
67         -- Enable menu group:
68         --
69         Window_Services.Ops.Menu_group_enable(
70             window => Inventory_Windows.
71                 main_window,
72             menu_group => inv_menu_group_ID,
73             enable => true);
74

```

**PRELIMINARY**

```

75  end Set_up_menu_group;
76
77
78
79  procedure Process_inquiry_menu(
80      selection: Terminal_Defs.menu_item_ID)
81      -- Selection made in this menu.
82  is
83      -- Logic:
84      -- Determine item selection, perform actions.
85
86  begin
87
88      case selection is
89
90          when inq_by_part_item => Inventory_Forms.
91              Process_inquiry_form;
92
93          when inq_by_desc_item =>
94
95              Message_Services.Write_msg(
96                  msg_id => no_selection_code,
97                  param1 =>
98                      Incident_Defs.message_parameter(
99                          typ => Incident_Defs.ord,
100                         len => 0)'(
101                             typ  => Incident_Defs.ord,
102                             len  => 0,
103                             o_val => selection));
104
105          when inq_exit_item =>
106              return;
107
108          when others => null;
109
110      end case;
111
112  end Process_inquiry_menu;
113
114
115
116  procedure Process_posting_menu(
117      selection: Terminal_Defs.menu_item_ID)
118      -- Selection made in this menu.
119  is
120      -- Logic:
121      -- Determine item selection, perform actions.
122
123  begin
124      case selection is
125
126          when post_receipt_item => Inventory_Forms.
127              Process_receipts_form;
128
129          when post_issue_item |
130              post_return_item |
131              post_spoilage_item |
132              post_journal_item =>
133
134              Message_Services.Write_msg(
135                  msg_id => no_selection_code,
136                  param1 => Incident_Defs.message_parameter(
137                      typ => Incident_Defs.ord,
138                      len => 0)'(
139                          typ  => Incident_Defs.ord,
140                          len  => 0,
141                          o_val => selection));
142
143          when post_exit_item =>
144              return;
145
146          when others => null;
147
148      end case;
149
150  end Process_posting_menu;
151

```

## PRELIMINARY

```
152 procedure Process_update_menu(  
153     selection: Terminal_Defs.menu_item_ID)  
154     -- Selection made in this menu.  
155 is  
156     -- Logic:  
157     -- Determine item selection, perform actions.  
158  
159 begin  
160  
161     case selection is  
162  
163         when update_add_item |  
164             update_change_item |  
165             update_delete_item =>  
166  
167             Inventory_Forms.Process_update_form(  
168                 selection => selection);  
169  
170         when update_exit_item =>  
171             return;  
172  
173         when others => null;  
174  
175     end case;  
176  
177 end Process_update_menu;  
178  
179  
180  
181 procedure Process_report_menu(  
182     selection: Terminal_Defs.menu_item_ID)  
183     -- Selection made in this menu.  
184 is  
185  
186     report_out_dev: System_Defs.text(256);  
187  
188 begin  
189  
190     case selection is  
191  
192         when report_by_part_item =>  
193  
194             Inventory_Forms.Process_report_form(  
195                 report_by_part => true,  
196                 report_out_dev => report_out_dev);  
197  
198             Inventory_Reports.Print_report_by_part(  
199                 output_dev_pathname => report_out_dev);  
200  
201  
202         when report_by_location_item =>  
203  
204             Inventory_Forms.Process_report_form(  
205                 report_by_part => false,  
206                 report_out_dev => report_out_dev);  
207  
208             Inventory_Reports.Print_report_by_location(  
209                 output_dev_pathname => report_out_dev);  
210  
211         when report_exit_item =>  
212             return;  
213  
214         when others => null;  
215  
216     end case;  
217  
218 end Process_report_menu;  
219  
220  
221  
222 procedure Process_housekeeping_menu(  
223     selection: Terminal_Defs.menu_item_ID)  
224     -- Selection made in this menu.  
225 is  
226  
227 begin  
228
```

PRELIMINARY

```
229 case selection is
230
231   when hskpg_index_item =>
232
233     File_Admin.Reorganize_index(
234       file => File_Defs.Convert_device_to_file(
235         s => Record_AM.Ops.Get_device_object(
236           opened_dev =>
237             Inventory_Files.parts_file)),
238       index =>
239         Inventory_Files.part_ID_index_name);
240
241   when hskpg_exit_item =>
242     return;
243
244   when others => null;
245
246 end case;
247
248 end Process_housekeeping_menu;
249
250
251 end Inventory_Menus;
```

## X-A.5.8 Inventory\_Reports Package Specification

```

1  with Device_Defs,
2      Incident_Defs,
3      Inventory_Messages,
4      System,
5      System_Defs,
6      Terminal_Defs,
7      Window_Services;
8
9  package Inventory_Reports is
10     --
11     -- Function:
12     -- Contains two procedures to process and
13     -- print either of the Inventory Program
14     -- reports (by part ID solely, or by part
15     -- location and then part ID) from the
16     -- Inventory Parts file.
17     --
18     -- One or the other of these procedures is
19     -- called from the Report Menu by the
20     -- appropriate menu selection: "Print
21     -- "Report by Part", or "Print Report by"
22     -- "Location".
23     --
24     -- History:
25     -- 05-21-87, William A. Rohm:  Written.
26     -- 10-27-87, WAR:              Revised.
27     --
28     -- End of Header
29
30     -- Incident codes for messages:
31     --
32     module: constant := 6;
33     -- Message module index.
34
35     --*M*  set.language :language = English
36     --*M*  create.variable module :value = 6
37
38     report_printing_code: constant
39         Incident_Defs.incident_code := (
40             message_object =>
41                 Inventory_Messages.message_object,
42             module          => module,
43             number          => 0,
44             severity        =>
45                 Incident_Defs.information);
46
47     --*M*  store :module = $module :number = 0\
48     --*M*  :msg_name = report_printing \
49     --*M*  :short = "Inventory parts file
50     --*M*  report by $p1<part/location>
51     --*M*  is now printing on device
52     --*M*  $p2<output device name>."
53
54
55     report_by_part_DDef_str: constant string :=
56         "/example/inventory/DDefs/report_by_part";
57     -- String constant for "report by part"
58     -- report DDef's pathname.
59
60     report_by_part_DDef_pathname:
61         System_Defs.text(
62             report_by_part_DDef_str'length) := (
63             report_by_part_DDef_str'length,
64             report_by_part_DDef_str'length,
65             report_by_part_DDef_str);
66     -- Text constant from "report by part"
67     -- DDef's pathname string.
68
69
70     report_by_loc_DDef_str: constant string :=
71         "/example/inventory/DDefs/report_by_location";
72     -- String constant for "report by location"
73     -- report DDef's pathname.
74

```

## PRELIMINARY

```
75 report_by_loc_DDef_pathname:
76   System_Defs.text(
77     report_by_loc_DDef_str'length) := (
78     report_by_loc_DDef_str'length,
79     report_by_loc_DDef_str'length,
80     report_by_loc_DDef_str);
81   -- Text constant from "report by location"
82   -- DDef's pathname string.
83
84
85 sort_by_loc_DDef_str: constant string :=
86 "/example/inventory/DDefs/sort_by_location";
87 -- String constant for "sort by location"
88 -- "(then by part ID)" sort DDef's pathname.
89
90 sort_by_loc_DDef_pathname:
91   System_Defs.text(
92     sort_by_loc_DDef_str'length) := (
93     sort_by_loc_DDef_str'length,
94     sort_by_loc_DDef_str'length,
95     sort_by_loc_DDef_str);
96   -- Text constant from "sort by location"
97   -- DDef's pathname string.
98
99
100 procedure Print_report_by_part(
101   output_dev_pathname: System_Defs.text);
102   -- Pathname of output device for
103   -- printing report. Can be any device
104   -- supporting the byte stream access
105   -- method.
106   --
107   -- Function:
108   -- Prepares report *by part ID* from parts
109   -- file, then prints report to given
110   -- output device.
111
112
113 procedure Print_report_by_location(
114   output_dev_pathname: System_Defs.text);
115   -- Pathname of output device for
116   -- printing report. Can be any device
117   -- supporting the byte stream access
118   -- method.
119   --
120   -- Function:
121   -- Sorts parts file by location (and then
122   -- by part ID) into temporary file, then
123   -- prints report to given output device.
124
125 end Inventory_Reports;
```

## X-A.5.9 Inventory\_Reports Package Body

Note: This example could not be compiled successfully due to the absence of the the Report\_Handler package at the time of this printing.

```

1  with Byte_Stream_AM,
2      Data_Definition_Mgt,
3      Device_Defs,
4      Directory_Mgt,
5      Event_Mgt,
6      File_Admin,
7      File_Defs,
8      Incident_Defs,
9      Inventory_Files,
10     Inventory_Windows,
11     Message_Services,
12     Passive_Store_Mgt,
13     Pipe_Mgt,
14     Process_Mgt,
15     Process_Mgt_Types,
16     Record_AM,
17     Report_Handler,
18     Sort_Merge_Interface,
19     System,
20     System_Defs,
21     Terminal_Defs,
22     Unchecked_conversion,
23     Volume_Set_Defs;
24
25 package body Inventory_Reports is
26     --
27     -- History:
28     --   05-21-87, William A. Rohm:  Written.
29     --   10-27-87, WAR:              Revised.
30     --
31     -- End of Header
32
33     -- Generic function:
34     --
35     function DDef_from_untyped is new
36         Unchecked_conversion(
37             source => System.untyped_word,
38             target => Data_Definition_Mgt.DDef_AD);
39
40     -- Type:
41     --
42     type connection_record is
43         -- Defines sort pipe's input and output, for
44         -- "Sort" and "Print" processes (called by
45         -- "Print_report_by_location").
46         record
47             sort_out:    Device_Defs.opened_device;
48             -- Output from "Sort" to pipe.
49             report_in:   Device_Defs.opened_device;
50             -- Input from pipe to "Print".
51             report_out:  Device_Defs.opened_device;
52             -- Output device for "Print".
53         end record;
54
55
56     procedure Print_report_by_part(
57         output_dev_pathname: System_Defs.text)
58     --
59     -- Logic:
60     --   1. Open parts file for reading
61     --   2. Open report output device
62     --   3. Get report DDef and initialize report
63     --   4. Print report and display message
64
65     is
66
67     opened_output: Device_Defs.opened_device;
68     -- Opened output device for printing report.
69
70     report_DDef: Data_Definition_Mgt.DDef_AD;
71     -- AD to a report data definition.

```

```

72
73 initialized_report: Device_Defs.opened_device;
74 -- Initialized (opened) report object itself.
75
76 local_parts_file: Device_Defs.device :=
77     Record_AM.Ops.Get_device_object(
78         Inventory_Files.parts_file);
79 -- AD to parts file.
80
81 opened_local_parts_file:
82     Device_Defs.opened_device;
83 -- AD to locally opened parts file.
84
85 part: System_Defs.text(4) := (4,4,"part");
86 -- Parameter to "report_printing" message,
87 -- since this report is by "part".
88
89 begin
90
91     -- Open parts file for reading, so no
92     -- concurrent updates will interfere:
93     --
94     opened_local_parts_file := Record_AM.Ops.Open(
95         dev      => local_parts_file,
96         input_output => Device_Defs.input,
97         allow    => Device_Defs.readers);
98
99
100    -- Open output device:
101    --
102    opened_output := Byte_Stream_AM.Open_by_name(
103        name      =>
104            output_dev_pathname,
105        input_output =>
106            Device_Defs.output);
107
108
109    -- Get report definition (DDef):
110    --
111    report_DDef := DDef_from_untyped(
112        Directory_Mgt.Retrieve(
113            name => report_by_part_DDef_pathname));
114    -- Assume "Report_Handler.Is_report".
115
116
117    -- Initialize report:
118    --
119    initialized_report := Report_Handler.Initialize(
120        description => report_DDef,
121        input      => opened_local_parts_file,
122        output     => opened_output);
123
124
125    -- Print report:
126    --
127    Report_Handler.Print(
128        report => initialized_report);
129
130
131    -- Display "report_printing" message:
132    --
133    Message_Services.Write_msg(
134        msg_id => report_printing_code,
135        param1 => Incident_Defs.message_parameter(
136            typ => Incident_Defs.txt,
137            len => part.length)' (
138            typ      => Incident_Defs.txt,
139            len      => part.length,
140            txt_val => part),
141        param2 => Incident_Defs.message_parameter(
142            typ => Incident_Defs.txt,
143            len => output_dev_pathname.length)' (
144            typ      => Incident_Defs.txt,
145            len      => output_dev_pathname.length,
146            txt_val => output_dev_pathname),
147        device => Inventory_Windows.message_window);
148

```

PRELIMINARY

```

149
150 -- Close locally opened parts file:
151 --
152 Record_AM.Ops.Close(
153     opened_dev => opened_local_parts_file);
154
155 end Print_report_by_part;
156
157
158 procedure Sort(
159     param_buffer: System.address;
160     -- Address of connection record.
161     param_length: System.ordinal)
162     -- Not used in this procedure, but required for
163     -- process's initial procedure.
164 --
165 -- Logic:
166 -- 1. Open local copy of parts file (sort input)
167 -- 2. Get sort DDef and perform sort
168 is
169
170     conn_rec: connection_record;
171     -- Record containing pipe input/output devices.
172     FOR conn_rec USE AT param_buffer;
173
174     local_parts_file: Device_Defs.device :=
175         Record_AM.Ops.Get_device_object(
176             Inventory_Files.parts_file);
177     -- AD to parts file.
178
179     opened_local_parts_file: Device_Defs.opened_device;
180     -- AD to locally opened parts file.
181
182     opened_sort_DDef:
183         Device_Defs.opened_device;
184     sort_DDef_reference:
185         Data_Definition_Mgt.node_reference;
186
187 begin
188
189     -- Open parts file for reading, so no
190     -- concurrent updates will interfere:
191     --
192     opened_local_parts_file := Record_AM.Ops.Open(
193         dev      => local_parts_file,
194         input_output => Device_Defs.input,
195         allow     => Device_Defs.readers);
196
197
198     -- Open sort definition (DDef):
199     --
200     opened_sort_DDef := Record_AM.Open_by_name(
201         name      =>
202             sort_by_loc_DDef_pathname,
203         input_output => Device_Defs.input,
204         allow       => Device_Defs.readers,
205         block      => true);
206
207     -- Get sort DDef's node reference:
208     --
209     sort_DDef_reference :=
210         Record_AM.Ops.Get_DDef(
211             opened_dev => opened_sort_DDef);
212
213
214     -- Perform sort, using sort DDef, from parts
215     -- file to pipe:
216     --
217     Sort_Merge_Interface.Sort(
218         input_device =>
219             opened_local_parts_file,
220         DDef         => sort_DDef_reference,
221         output_device => conn_rec.sort_out,
222         stable_sort  => true,
223         tuning_opts  =>
224             Sort_Merge_Interface.no_tuning);
225

```

```

226     -- Close locally opened parts file:
227     --
228     Record_AM.Ops.Close(
229         opened_dev => opened_local_parts_file);
230
231 end Sort;
232 pragma subprogram_value(
233     Process_Mgt.Initial_proc,
234     Sort);
235
236
237 procedure Print(
238     param_buffer: System.address;
239     -- Address of connection record.
240     param_length: System.ordinal)
241     -- Not used in this procedure, but required for
242     -- process's initial procedure.
243     --
244     -- Logic:
245     -- 1. Get report DDef
246     -- 2. Open report output
247     -- 3. Get report DDef and initialize report
248     -- 4. Print report from pipe output.
249
250 is
251
252     report_DDef: Data_Definition_Mgt.DDef_AD;
253     -- AD to a report data definition.
254
255     initialized_report: Device_Defs.opened_device;
256     -- Initialized (opened) report object itself.
257
258     conn_rec: connection_record;
259     -- Record containing pipe input/output devices.
260     FOR conn_rec USE AT param_buffer;
261
262 begin
263
264     -- Get report definition (DDef):
265     --
266     report_DDef := DDef_from_untyped(
267         Directory_Mgt.Retrieve(
268             report_by_loc_DDef_pathname));
269
270     -- Initialize report:
271     --
272     initialized_report := Report_Handler.Initialize(
273         description => report_DDef,
274         input       => conn_rec.report_in,
275         output      => conn_rec.report_out);
276
277     -- Print report:
278     --
279     Report_Handler.Print(
280         report => initialized_report);
281
282     -- Close report output device:
283     --
284     Record_AM.Ops.Close(
285         opened_dev => conn_rec.report_out);
286
287 end Print;
288 pragma subprogram_value(Process_Mgt.Initial_proc,
289     Print);
290
291
292 procedure Print_report_by_location(
293     output_dev_pathname: System_Defs.text)
294     --
295     -- Logic:
296     -- 1. Open pipe input (sort output) and
297     --    output (report input)
298     -- 2. Spawn "Sort" and "Print" processes
299     -- 3. Wait for termination of processes
300     -- 4. Deallocate processes
301     -- 5. Display "report printing" message
302 is

```

PRELIMINARY

```

303
304 conn_rec: connection_record;
305     -- Record referencing all I/O connections used by
306     -- the child processes.
307
308 sort_pipe: Pipe_Mgt.pipe_AD;
309     -- Pipe from sort output to report input.
310
311 this_process_untyped: System.untyped_word;
312     -- Process executing call to
313     -- "Print_report_by_location", as an
314     -- untyped word.
315
316 sort_process: Process_Mgt.Types.process_AD;
317     -- Process executing "Sort".
318
319 print_process: Process_Mgt.Types.process_AD;
320     -- Process executing "Print".
321
322 term_events: Event_Mgt.action_record_list(2);
323     -- Array that receives termination events of the
324     -- two child processes.
325
326 location: System_Defs.text(8) := (8,8,"location");
327     -- Parameter to "report_printing" message, since
328     -- this report is by "location".
329
330 begin
331     -- Create pipe:
332     --
333     sort_pipe := Pipe_Mgt.Create_pipe;
334
335     -- Open sort output, report input, and report
336     -- output devices:
337     --
338     conn_rec := (
339         sort_out   => Record_AM.Ops.Open(
340             Pipe_Mgt.Convert_pipe_to_device(
341                 sort_pipe),
342             Device_Defs.output),
343         report_in  => Record_AM.Ops.Open(
344             Pipe_Mgt.Convert_pipe_to_device(
345                 sort_pipe),
346             Device_Defs.input),
347         report_out => Record_AM.Open_by_name(
348             output_dev_pathname,
349             Device_Defs.output));
350
351     -- Get this process's AD:
352     --
353     this_process_untyped :=
354         Process_Mgt.Get_process_globals_entry(
355             Process_Mgt.Types.process);
356
357     -- Spawn "Sort" process:
358     --
359     sort_process := Process_Mgt.Spawn_process(
360         init_proc   => Sort'subprogram_value,
361         param_buffer => conn_rec'address,
362         term_action => (
363             event     => Event_Mgt.user_1,
364             message   => System.null_address,
365             destination => this_process_untyped));
366
367     -- Spawn "Print" process:
368     --
369     print_process := Process_Mgt.Spawn_process(
370         init_proc   => Print'subprogram_value,
371         param_buffer => conn_rec'address,
372         term_action => (
373             event     => Event_Mgt.user_2,
374             message   => System.null_address,
375             destination => this_process_untyped));
376
377     -- Wait for both processes to finish:
378     --
379

```

PRELIMINARY

```
380     Event_Mgt.Wait_for_all(  
381     events =>  
382         (Event_Mgt.user_1 .. Event_Mgt.user_2 =>  
383             true,  
384             others => false),  
385     action_list => term_events);  
386  
387     -- The two processes must have terminated, so they  
388     -- can be deallocated:  
389     --  
390     Process_Mgt.Deallocate(sort_process);  
391     Process_Mgt.Deallocate(print_process);  
392  
393     -- Display "report printing" message:  
394     --  
395     Message_Services.Write_msg(  
396     msg_id => report_printing_code,  
397     param1 => Incident_Defs.message_parameter(  
398     -- "location"  
399     typ => Incident_Defs.txt,  
400     len => location.length)'(  
401     typ => Incident_Defs.txt,  
402     len => location.length,  
403     txt_val => location),  
404     param2 => Incident_Defs.message_parameter(  
405     -- "output device pathname"  
406     typ => Incident_Defs.txt,  
407     len => output_dev_pathname.length)'(  
408     typ => Incident_Defs.txt,  
409     len => output_dev_pathname.length,  
410     txt_val => output_dev_pathname));  
411  
412     end Print_report_by_location;  
413  
414 end Inventory_Reports;  
415
```

## X-A.5.10 Inventory\_Windows Package Specification

```

1  with Device_Defs,
2      Terminal_Defs;
3
4  package Inventory_Windows is
5      --
6      -- Function:
7      --   Contains procedures to open and close the two
8      --   Inventory Program windows: the main window and
9      --   the message window.
10     --
11     --   The main window is used for menu and form
12     --   display and for user data entry. The message
13     --   window is only used to display status and error
14     --   messages to the user.
15     --
16     -- History:
17     --   06-04-87, William A. Rohm: Written.
18     --
19     -- End of Header
20
21     -- Constants:
22     --
23     module: constant := 2;
24     -- Message module index value, for this
25     -- package's messages. Not currently used.
26
27     main_window_size: Terminal_Defs.point_info := (
28         80,20);
29     -- Size of main window, in columns and rows.
30
31     main_buffer_size: Terminal_Defs.point_info := (
32         80,20);
33     -- Size of main window's buffer.
34
35     main_window_pos: Terminal_Defs.point_info := (
36         1,1);
37     -- Position of main window (upper left corner).
38
39     message_window_size: Terminal_Defs.point_info := (
40         80,3);
41     -- Size of message window, in columns and rows.
42
43     message_buffer_size: Terminal_Defs.point_info := (
44         80,3);
45     -- Size of message window's buffer.
46
47     message_window_pos: Terminal_Defs.point_info := (
48         1, 1 + main_window_pos.vert);
49     -- Position of message window (just below main
50     -- window).
51
52     -- Variables:
53     --
54     main_window: Device_Defs.opened_device;
55     -- Main window, for displaying menus and forms
56     -- and getting user input. Usable by other
57     -- modules after "Open_program_windows" has been
58     -- called.
59
60     message_window: Device_Defs.opened_device;
61     -- Message window, for status and error
62     -- messages. Usable by other modules after
63     -- "Open_program_windows" has been called.
64
65
66
67     procedure Open_program_windows;
68     --
69     -- Function:
70     --   Open both program windows (main and message)
71     --   on the current terminal.
72     --
73     --   The main window is for the Inventory
74     --   Program's menus and forms. The message

```

## PRELIMINARY

```
75      -- window is opened, for message display.
76      --
77      -- The main window is opened at the top of the
78      -- screen. The message window is opened below
79      -- the main window.
80
81
82
83      procedure Close_program_windows;
84      --
85      -- Function:
86      -- Closes both Inventory Program windows: main
87      -- window and message window.
88
89
90      end Inventory_Windows;
91
```

## X-A.5.11 Inventory\_Windows Package Body

```

1  with Byte_Stream_AM,
2      Device_Defs,
3      Process_Mgt,
4      Process_Mgt_Types,
5      System,
6      Terminal_Defs,
7      Window_Services;
8
9  package body Inventory_Windows
10 is
11
12  procedure Open_program_windows
13  --
14  -- Logic:
15  -- 1. Gets device AD to underlying terminal.
16  -- 2. Opens main window, assigning
17  --    "inventory_main".
18  -- 3. Opens message window, assigning
19  --    "inventory_message".
20
21  is
22      old_opened_window: Device_Defs.opened_device;
23      old_window: Device_Defs.device;
24      underlying_terminal: Device_Defs.device;
25
26  begin
27
28  -- Assume standard input, on entry, is from an
29  -- opened window:
30  --
31  old_opened_window :=
32      Process_Mgt.Get_process_globals_entry(
33          Process_Mgt_Types.standard_input);
34
35  -- Get device object of standard input window:
36  --
37  old_window := Byte_Stream_AM.Ops.Get_device_object(
38      old_opened_window);
39
40  -- Get device AD of standard input window's
41  -- terminal:
42  --
43  underlying_terminal :=
44      Window_Services.Ops.Get_terminal(
45          old_window);
46
47  -- Create new main window:
48  --
49  main_window := Window_Services.Ops.Create_window(
50      terminal      => underlying_terminal,
51      pixel_units  => false,
52      -- characters, not pixels
53      fb_size      => main_buffer_size,
54      desired_window_size => main_window_size,
55      window_pos   => main_window_pos,
56      view_pos     =>
57          Terminal_Defs.point_info'(1,1));
58
59
60  -- Create new message window:
61  --
62  message_window := Window_Services.Ops.Create_window(
63      terminal      => underlying_terminal,
64      pixel_units  => false,
65      fb_size      => message_buffer_size,
66      desired_window_size => message_window_size,
67      window_pos   => message_window_pos,
68      view_pos     =>
69          Terminal_Defs.point_info'(1,1));
70
71  end Open_program_windows;
72
73
74

```

```
75 procedure Close_program_windows
76   --
77   -- Logic:
78   --   1. Closes main window.
79   --   2. Closes message window.
80
81   is
82
83   begin
84
85     Window_Services.Ops.Destroy_window(main_window);
86
87     Window_Services.Ops.Destroy_window(message_window);
88
89   end Close_program_windows;
90
91 end Inventory_Windows;
```

## X-A.5.12 Inventory\_Messages Package Specification

```

1  with Incident_Defs,
2      System,
3      System_Defs;
4
5  package Inventory_Messages is
6      --
7      -- Function:
8      --   Defines Inventory Example Program's message
9      --   object, used for all incident code declarations
10     --   in the program.
11     --
12     --   Each package defines its own messages (using
13     --   tagged message definitions) with its unique
14     --   module number.
15     --
16     -- History:
17     --   07-27-87, William A. Rohm:  Written.
18     --   10-27-87, WAR:             Revised.
19     --
20     -- End of Header
21
22     -- Constants:
23     --
24     message_file: constant System_Defs.text_AD :=
25         new System_Defs.text'(
26             31,31,"/example/inventory/message_file");
27         -- AD to message file text name.
28         --
29         -- *This will go away when "pragma bind" changes.*
30
31
32     message_object: constant System.untyped_word :=
33         System.null_word;
34
35     pragma bind (message_object,
36                 "inventory_messages.message_file");
37     -- Message object for Inventory Program Incident
38     -- codes. Bound to "message_file" constant by
39     -- pragma "bind".
40     --
41     -- *When the resident compiler/linker is in place,*
42     -- *this pragma will become:*
43     -- | pragma bind(message_object,
44     -- |               "/example/inventory/message_file");
45
46 end Inventory_Messages;
```

## X-A.6 Program Services

## X-A.6.1 At\_cmd\_ex Procedure

```

1  with At_Support_Ex,
2      Command_Handler,
3      Device_Defs,
4      Long_Integer_Defs,
5      Message_Services,
6      System_Defs,
7      Timed_Requests_Mgt;
8
9  procedure At_cmd_ex
10     --
11     -- Function:
12     --   This procedure will run a command at a specified time.
13     --   It sets defaults for unspecified parameters and
14     --   parses mandatory and specified time parameters
15     --   and calls subprogram that will initial a new session
16     --   and job to run the command. The prompt will
17     --   return after the new job is started. The until
18     --   and count arguments are only effective if period is
19     --   set
20     --
21     -- History:
22     --   04-05-88, Ed Sassone, creation date
23     --   05-20-88, Ed Sassone, working version
24     --
25     -- End of Header
26     --
27     --
28     -- Command Definition:
29     at_cmd_ex      :time=<extended_string_list(1..25(1..11))>
30                   :command=<extended_string(1..80)>
31                   [:period=<extended_string_list(0..25(0..11))>:=("()")]
32                   [:until=<extended_string_list(0..25(0..11))>:=("()")]
33                   [:count=<integer(1..1_000)>:=1_000]
34                   --
35                   --
36     --*D*   manage.commands
37     --*D*   create.invocation_command
38     --*D*
39     --*D*   define.argument time \
40     --*D*     :type = string_list
41     --*D*     set.maximum_length 25 11
42     --*D*     set.mandatory
43     --*D*   end
44     --*D*
45     --*D*   define.argument command \
46     --*D*     :type = string
47     --*D*     set.maximum_length 80
48     --*D*     set.mandatory
49     --*D*   end
50     --*D*
51     --*D*   define.argument period \
52     --*D*     :type = string_list
53     --*D*     set.maximum_length 25 11
54     --*D*     allow.null_values :list :element
55     --*D*     set.value_default "("
56     --*D*   end
57     --*D*
58     --*D*   define.argument until \
59     --*D*     :type = string_list
60     --*D*     set.maximum_length 25 11
61     --*D*     allow.null_values :list :element
62     --*D*     set.value_default "("
63     --*D*   end
64     --*D*
65     --*D*   define.argument count \
66     --*D*     :type=integer
67     --*D*     set.value_default 1000           -- function ($$upper) NYI
68     --*D*     set.bounds 1..1000             -- open bounds NYI
69     --*D*   end
70     --*D*
71     --*D*   end           -- create.invocation_command
72
73 is
74

```

## PRELIMINARY

```

75
76 use Long_Integer_Defs;      -- for time comparison
77
78
79   odo:      Device_Defs.opened_device;
80
81 -- parameters
82 time:      System_Defs.string_list(25) :=
83   (25, 0, 0, (others => ' '));
84
85 command:   System_Defs.text(80) :=
86   (80, 0, (others => ' '));
87
88 period:    System_Defs.string_list(25) :=
89   (25, 0, 0, (others => ' '));
90
91 until:     System_Defs.string_list(25) :=
92   (25, 0, 0, (others => ' '));
93
94 count:     integer;
95
96
97 start_at:  System_Defs.system_time_units :=
98   System_Defs.null_time;
99   -- stu equivalent of time
100
101 next_at:   System_Defs.system_time_units :=
102   System_Defs.null_time;
103   -- stu equivalent of period
104
105 until_at:  System_Defs.system_time_units :=
106   Long_Integer_Defs.max_int;
107   -- stu equivalent of until
108
109 begin
110
111   odo := Command_Handler.
112     Open_invocation_command_processing;
113
114   Command_Handler.Get_string_list(
115     cmd_odo => odo,
116     arg_number => 1,
117     arg_value => time);
118
119   Command_Handler.Get_string(
120     cmd_odo => odo,
121     arg_number => 2,
122     arg_value => command);
123
124   Command_Handler.Get_string_list(
125     cmd_odo => odo,
126     arg_number => 3,
127     arg_value => period);
128
129   Command_Handler.Get_string_list(
130     cmd_odo => odo,
131     arg_number => 4,
132     arg_value => until);
133
134   count :=
135     Command_Handler.Get_integer(
136       cmd_odo => odo,
137       arg_number => 5);
138
139   Command_Handler.Close(odo);
140
141   -- parse timing arguments
142
143   start_at :=
144     At_Support_Ex.Parse_time(
145       time => time,
146       from_when => Timed_Requests_Mgt.system_epoch);
147
148   if period.length > 4 then
149     -- keep defaults if nothing assigned
150     next_at :=
151       At_Support_Ex.Parse_time(

```

PRELIMINARY

```
152         time      => period,
153         from_when => Timed_Requests_Mgt.now);
154     else
155         count := 1; -- if no period do command only once
156     end if;
157
158     if until.length > 4 then
159         -- keep defaults if nothing assigned
160         until_at :=
161             At_Support_Ex.Parse_time(
162                 time      => until,
163                 from_when => Timed_Requests_Mgt.system_epoch);
164     end if;
165
166     if start_at < Timed_Requests_Mgt.get_time then
167
168         Message_Services.Write_msg(
169             msg_id => At_Support_Ex.prior_time_warning_code);
170     end if;
171
172     -- creates new session and job so prompt will return
173     At_Support_Ex.Create_waiting_process(
174         invocation_record => At_Support_Ex.program_record' (
175             command      => command,
176             stu_start    => start_at,
177             stu_period   => next_at,
178             stu_until    => until_at,
179             count        => count));
180
181 end At_cmd_ex;
```

## X-A.6.2 At\_Support\_Ex Package Specification

```

1  with Incident_Defs,
2     Process_Mgt,
3     Timed_Requests_Mgt,
4     System,
5     System_Defs;
6
7  package At_Support_Ex is
8  --
9  -- Function:
10 -- Provides support for At_cmd_ex. Parses time
11 -- arguments and invokes the given command either
12 -- once at the specified time or from the given time
13 -- multiple times based on a specified period until
14 -- a given count or time limit, whichever is first.
15 --
16 -- History:
17 -- 04-05-88, Ed Sassone, creation date
18 -- 05-20-88, Ed Sassone, working version
19 --
20 -- Exception Codes:
21 msg_obj: constant System.untyped_word :=
22     System.null_word; -- use oeo
23
24 time_format_error_code: constant Incident_Defs.
25     incident_code := (
26     module      => 0,
27     number      => 1,
28     severity    => Incident_Defs.error,
29     message_object => msg_obj);
30 day_format_error_code: constant Incident_Defs.
31     incident_code := (
32     module      => 0,
33     number      => 2,
34     severity    => Incident_Defs.error,
35     message_object => msg_obj);
36
37 prior_time_warning_code: constant Incident_Defs.
38     incident_code := (
39     module      => 0,
40     number      => 3,
41     severity    => Incident_Defs.warning,
42     message_object => msg_obj);
43
44 --
45 -- Exceptions:
46 --
47 --*D* manage.messages
48 --
49 time_format_error: exception;
50 -- Occurs when the time was not input in a proper
51 -- format
52 --*D* store 0 1 time_format_error \
53 --*D* :short = "$pl is an improper time specification
54 --*D*The correct format is hh[:mm[:ss[.dd]]]"
55 day_format_error: exception;
56 -- Occurs when the day was not input in a proper
57 -- format
58 --*D* store 0 2 day_format_error \
59 --*D* :short = "$pl is an improper time specification
60 --*D*The correct format is [MM/]DD[/YYYY]"
61
62
63 -- Warning message occurs when the time
64 -- specified has already past
65 --*D* store 0 3 prior_time_warning \
66 --*D* :short = "The specified time has already past.
67 --*D*Command is executed immediately."
68 --
69 -- End of Header
70
71
72 type program_record is record
73 -- times in this record are all in
74 -- system_time_units to be used by Timed_request

```

## PRELIMINARY

```
75     command:      System_Defs.text(80);
76     -- command to be run with arguments
77     stu_start:    System_Defs.system_time_units;
78     -- initial request
79     stu_period:   System_Defs.system_time_units;
80     -- interval between execution (optional argument)
81     stu_until:   System_Defs.system_time_units;
82     -- upper time limit on command run more than once
83     count:       integer;
84     -- number of times job will run
85 end record;
86
87
88
89 function Parse_time(
90     time:      System_Defs.string_list;
91     -- time from command line
92     from_when: Timed_Requests_Mgt.from_when_type)
93     -- specifies time to be relative to now
94     -- or absolute
95     return System_Defs.system_time_units;
96     -- time in form usable for
97     -- Timed_Request.Enter_request
98 --
99 -- Function:
100 -- Parses the time argument on the command line and
101 -- converts to system_time_units. The time
102 -- specification is divided into two strings, the
103 -- first being mandatory specifying hours and
104 -- minutes and optionally seconds and hundredths of
105 -- seconds. The second string is optional and
106 -- specifies the day of month and optionally the
107 -- month and year.
108 --
109 -- Exceptions:
110 -- time_format_error - raised when the hour string list
111 --                     input for the timing
112 --                     parameters is incorrect.
113 --
114 -- day_format_error - raised when the day string list
115 --                     input for the timing parameters
116 --                     is incorrect.
117 --
118 --
119 procedure Create_waiting_process(
120     invocation_record: program_record);
121 --
122 -- Function:
123 -- Creates a new session, job and process to wait
124 -- for specified time to execute.
125 --
126 --
127 --
128 --
129 procedure Wait_program(
130     param_buffer: System.address;
131     param_length: System.ordinal);
132     pragma subprogram_value(Process_Mgt.initial_proc,
133     Wait_program);
134 --
135 -- Function:
136 -- Created in a new session and job. Process issues
137 -- a timed request and waits on the locked semaphore
138 -- for specified time to execute program passed in
139 -- as a parameter. If the command is specified more
140 -- than once it will loop, issue another timing
141 -- request and reset the semaphore and wait.
142 --
143 --
144 end At_Support_Ex;
```

## X-A.6.3 At\_Support\_Ex Package Body

```

1  with Command_Execution,
2      Directory_Mgt,
3      Job_Admin,           -- trusted
4      Job_Mgt,
5      Job_Types,
6      Long_Integer_Defs,
7      Incident_Defs,
8      Message_Services,
9      Message_Stack_Mgt,
10     Semaphore_Mgt,
11     Session_Mgt,
12     Session_Types,
13     String_List_Mgt,
14     System,
15     System_Defs,
16     Text_IO,
17     Text_Mgt,
18     Timed_Requests_Mgt,
19     Timing_String_Conversions,
20     Timing_Conversions;
21
22  package body At_Support_Ex is
23  --
24  -- Logic:
25  -- Supports at command by parsing time specification and creating
26  -- new session, job and process that will wait for timing requests
27  -- to invoke the waiting process.
28  --
29  -- History:
30  -- 04-05-88, Ed Sassone, creation date
31  -- 05-20-88, Ed Sassone, working version
32  --
33  -- End of Header
34
35  -----
36  --                               PARSE_TIME                               --
37  -----
38  function Parse_time(
39      time: System_Defs.string_list;
40      from_when: Timed_Requests_Mgt.from_when_type)
41  return System_Defs.system_time_units
42  --
43  -- Logic:
44  -- This function first parses the mandatory string
45  -- containing hours, minutes, seconds, hundreths and
46  -- then it parses the second optional string
47  -- containing month day and year. For each string
48  -- it counts the number and position of the
49  -- separator. For the first string that is the ':'
50  -- and the '.' if hundreths are specified.
51  -- For the second string it is the '/'. Based on the
52  -- separator positions, substrings representing the
53  -- individual time elements are copied into the
54  -- appropriate fields of string_time.
55  --
56  is
57
58
59  use Timed_Requests_Mgt;
60  -- needed in "if from_when = system_epoch statement"
61
62
63  dum_text: constant System_Defs.text(11) :=
64  (11, 11, (others => ' '));
65  -- used for the following initialization only:
66
67  string_time: Timing_String_Conversions.string_time :=
68  ("0000", " ", " ", "00", "00", "00", "00", "00", dum_text,
69  " ", " ", " ");
70  -- specified time values are copied into fields if
71  -- absolute time is used value is preloaded with
72  -- current time. Fields specified are overwritten
73
74  string_interval: Timing_String_Conversions.string_interval;

```

## PRELIMINARY

```

75         -- used for period (relative time)
76
77     hour_time:      System_Defs.text(Incident_Defs.txt_length);
78         -- used for hh:mm:ss.dd field
79
80     day_time:       System_Defs.text(Incident_Defs.txt_length);
81         -- used for MM/DD/YYYY field
82
83     separators:     array (1 .. 2) of
84         System_Defs.text_length;
85         -- array of positions of separators
86
87     number_separators: integer := 0;
88         -- hold the number of separators in the field
89
90     month:          string (1 .. 2) := "00";
91         -- used in place of string_time.month because
92         -- string_time.month is Jan..Dec and specified
93         -- month is 1..12
94
95     package Int_IO is new Text_IO.Integer_IO(integer);
96         -- needed for conversions from string to numeric
97         -- month
98
99     begin
100
101         -- initialize string_time record
102
103         if from_when = system_epoch then
104             -- absolute time for current day
105             string_time := Timing_String_Conversions.
106                 Convert_numeric_time_to_string(
107                     num_time => Timing_Conversions.
108                         Convert_stu_to_numeric_time(
109                             stu => Timed_Requests_Mgt.
110                                 Get_time)); -- current time
111
112             -- default if not specified
113             string_time.minute := "00";
114             string_time.second := "00";
115             string_time.hundredth := "00";
116         end if;
117
118         -- *** PARSE MANDATORY HOUR STRING ***
119
120         String_List_Mgt.Get_element(
121             from => time,
122             el_pos => 1,
123             element => hour_time);
124
125         -- find positions and number of ":"
126         number_separators := 0;
127         separators := (others => 0);
128         for pos in 1 .. hour_time.length
129             loop
130                 if hour_time.value(pos) = ':' then
131                     number_separators := number_separators + 1;
132                     -- no more than 2 ":" allowed
133                     if number_separators > 2 then
134                         RAISE time_format_error;
135                     end if;
136                     separators(number_separators) := pos;
137                     -- if non-digit or not the other separator
138                     elsif (hour_time.value(pos) < '0' or
139                             hour_time.value(pos) > '9') and
140                             hour_time.value(pos) /= '.' then
141                         RAISE time_format_error;
142                     end if;
143             end loop;
144
145         case number_separators is
146             when 0 =>
147                 if hour_time.length > 2 then
148                     RAISE time_format_error;
149                 end if;
150                 string_time.hour := hour_time.value;
151             when 1 =>

```

**PRELIMINARY**

```

152     if separators(1) /= 3 then
153         RAISE time_format_error;
154     end if;
155     string_time.hour := hour_time.value(1 .. 2);
156     string_time.minute := hour_time.value(4 .. 5);
157 when 2 =>
158     if separators(1) /= 3 or separators(2) /= 6 then
159         RAISE time_format_error;
160     end if;
161     string_time.hour := hour_time.value(1 .. 2);
162     string_time.minute := hour_time.value(4 .. 5);
163     string_time.second := hour_time.value(7 .. 8);
164
165     -- do hundredths if specified
166     declare
167     pos: integer := Text_Mgt.Locate('.', hour_time);
168     begin
169         case pos is
170             when 0 =>
171                 null;
172             when 9 =>
173                 string_time.hundredth := hour_time.value
174                     (pos + 1 .. pos + 2);
175             when others =>
176                 RAISE time_format_error;
177         end case;
178     end; -- declare
179 when others =>
180     RAISE time_format_error;
181 end case;
182
183 -- *** PARSE OPTIONAL DAY STRING ***
184
185 if time.count = 2 then
186     String_List_Mgt.Get_element(
187         from => time,
188         el_pos => 2,
189         element => day_time);
190
191     -- find positions of "/"
192     number_separators := 0;
193     separators := (others => 0);
194     for pos in 1 .. day_time.length
195     loop
196         if day_time.value(pos) = '/' then
197             number_separators := number_separators + 1;
198             -- no more than 2 "/" allowed
199             if number_separators > 2 then
200                 RAISE day_format_error;
201             end if;
202             separators(number_separators) := pos;
203             -- digits only if not a valid separator
204             elsif day_time.value(pos) < '0' or
205                 day_time.value(pos) > '9' then
206                 RAISE day_format_error;
207             end if;
208         end loop;
209
210     case number_separators is
211     when 0 =>
212         -- day of month only
213         string_time.day := day_time.value;
214
215     when 1 =>
216         -- month and day
217         if separators(1) /= 3 then
218             RAISE day_format_error;
219         end if;
220         month := day_time.value(1 .. 2);
221         string_time.day := day_time.value(4 .. 5);
222
223     when 2 =>
224         -- month, day and year
225         if separators(1) /= 3 or separators(2) /= 6 then
226             RAISE day_format_error;
227         end if;
228         month := day_time.value(1 .. 2);

```

PRELIMINARY

```

229     string_time.day := day_time.value(4 .. 5);
230     string_time.year := day_time.value(7 .. 10);
231
232     when others =>
233         RAISE day_format_error;
234     end case;
235
236
237     -- convert 1..12 month to Jan..Dec month
238     declare
239
240         month_tmp:           integer;
241         -- temporary variable for month conversion
242
243         length:             positive;
244         -- dummy variable for month conversion
245
246     begin
247
248         Int_IO.get(           -- convert string to ordinal
249             from => month,
250             item => month_tmp,
251             last => length);
252
253         case month_tmp is
254             when 0 =>
255                 null; --blank initial string
256             when 1 =>
257                 string_time.month := "Jan";
258             when 2 =>
259                 string_time.month := "Feb";
260             when 3 =>
261                 string_time.month := "Mar";
262             when 4 =>
263                 string_time.month := "Apr";
264             when 5 =>
265                 string_time.month := "May";
266             when 6 =>
267                 string_time.month := "Jun";
268             when 7 =>
269                 string_time.month := "Jul";
270             when 8 =>
271                 string_time.month := "Aug";
272             when 9 =>
273                 string_time.month := "Sep";
274             when 10 =>
275                 string_time.month := "Oct";
276             when 11 =>
277                 string_time.month := "Nov";
278             when 12 =>
279                 string_time.month := "Dec";
280             when others =>
281                 RAISE day_format_error;
282         end case;
283     end; -- declare
284
285     end if; -- if time.count = 2
286
287     -- range checking goes here
288
289     if from_when = system_epoch then
290         -- absolute time
291         return Timing_Conversions.Convert_numeric_time_to_stu(
292             num_time => Timing_String_Conversions.
293                 Convert_string_time_to_numeric(
294                     str_time => string_time));
295     else
296         -- relative time
297         -- initialize to zero
298         string_interval := Timing_String_Conversions.
299             Convert_numeric_interval_to_string(
300                 num_interval => Timing_Conversions.
301                     Convert_stu_to_numeric_interval(
302                         stu => System_Defs.null_time));
303
304         string_interval.sign := ' ';
305         string_interval.days(7 .. 8) := string_time.day;

```

**PRELIMINARY**

```

306     string_interval.hours(11 .. 12) := string_time.hour;
307     string_interval.minutes(11 .. 12) := string_time.minute;
308     string_interval.seconds(11 .. 12) := string_time.second;
309     string_interval.hundredths(11 .. 12) := string_time.hundredth;
310
311     return Timing_Conversions.Convert_numeric_interval_to_stu(
312         num_interval => Timing_String_Conversions.
313             Convert_string_interval_to_numeric(
314                 str_interval => string_interval));
315 end if;
316
317 exception
318
319     when time_format_error =>
320         Message_Services.Write_msg(
321             msg_id => time_format_error_code,
322             param1 => Incident_Defs.message_parameter'(
323                 typ      => Incident_Defs.txt,
324                 len      => Incident_Defs.txt_length,
325                 txt_val => hour_time));
326         RAISE;
327
328     when day_format_error =>
329         Message_Services.Write_msg(
330             msg_id => day_format_error_code,
331             param1 => Incident_Defs.message_parameter'(
332                 typ      => Incident_Defs.txt,
333                 len      => Incident_Defs.txt_length,
334                 txt_val => day_time));
335         RAISE;
336
337 end Parse_time;
338
339 -----
340 --          CREATE_WAITING_PROCESS          --
341 -----
342
343 procedure Create_waiting_process(
344     invocation_record: program_record)
345 is
346     --
347     -- Logic:
348     -- Creates a new session, then a job in that session,
349     -- and then the waiting process from that job.
350     --
351
352     new_name:          constant System_Defs.text(13) :=
353         (13, 13, "timed request");
354
355     job_info:          Job_mgt.job_info(80);
356     -- SSO field used for creating new session
357
358     new_job_AD:        Job_Types.job_AD;
359
360     program_length:   System.ordinal := System.ordinal(
361         invocation_record'size / System.storage_unit);
362
363 begin
364
365     -- retrieves SSO for new session
366     Job_Mgt.Get_job_info(
367         info => job_info);
368
369     new_job_AD := Job_Admin.Invoke_job(
370         init_proc  => Wait_program'subprogram_value,
371         param_buffer => invocation_record'address,
372         param_length => program_length,
373         text       => new_name,
374         session    => Session_Mgt.create_session(
375             SSO      => job_info.SSO,
376             session_name => new_name));
377
378 end Create_waiting_process;
379
380 -----
381 --          WAIT_PROGRAM          --
382 -----

```

**PRELIMINARY**

```

383 -----
384 procedure Wait_program(
385     param_buffer: System.Address;
386     param_length: System.ordinal)
387
388 is
389
390
391     use Long_Integer_Defs;
392     -- for system_time_units
393
394
395     program_rec :    program_record;
396     FOR program_rec USE AT param_buffer;
397
398     command_job_AD:  Job_Types.job_Ad;
399
400     req_index:       Timed_Requests_Mgt.request_index;
401
402     wait:            Semaphore_Mgt.semaphore_AD :=
403         Semaphore_Mgt.Create_semaphore(
404             initial_count => 0);
405     -- create semaphore in locked state
406     -- blocks job until time specified
407
408 begin
409
410     -- period must be non-null for
411     -- Timed_Requests_Mgt.Get_next_activation
412     if program_rec.stu_period = System_Defs.null_time then
413         program_rec.stu_period := System_Defs.stu_per_min;
414     end if;
415
416     -- Loop until count is expired or "until" time is
417     -- expired, whichever is first.  Count and until both
418     -- have defaults of max_int.  If period was not specified
419     -- the loop count was set to one by the driver
420     while program_rec.stu_until >= program_rec.stu_start
421         and program_rec.count > 0
422     loop
423         req_index :=
424             Timed_Requests_Mgt.Enter_request(
425                 req_info => Timed_Requests_Mgt.request_info(
426                     Timed_Requests_Mgt.semaphore_signal)'(
427                     kind      => Timed_Requests_Mgt.semaphore_signal,
428                     wakeup_time => program_rec.stu_start,
429                     from_when  => Timed_Requests_Mgt.system_epoch,
430                     semaphore  => wait));
431
432         -- wait until Timed_Requests unlocks semaphore
433         -- NOTE: there is about a 3 second delay before the
434         -- command is actually run
435         Semaphore_Mgt.P(semaphore => wait);
436
437         command_job_AD :=
438             Command_Execution.Run_program_or_script(
439                 command => program_rec.command);
440
441         program_rec.count := program_rec.count - 1;
442
443         -- NOTE1: This is an expensive call that should only be
444         -- used when slippage cannot be tolerated.
445         -- NOTE2: The call should be placed after command invocation.
446         Timed_Requests_Mgt.Get_next_activation(
447             period      => program_rec.stu_period, -- this cannot be null
448             next_activation => program_rec.stu_start);
449
450     end loop;
451
452 end Wait_program;
453
454 end At_Support_Ex;

```

**X-A.6.4 Compiler\_Ex Package Specification**

```
1  with Device_Defs;
2
3  package Compiler_Ex is
4      --
5      -- Function:
6      --   Supplies the procedural interface a Pascal
7      --   compiler.
8      --
9      --   This interface can be used to write the
10     --   compiler invocation script. End of Header
11     --
12     -- History:
13     --   08-10-87, Paul Schwabe: initial revision.
14     --   12-02-87, Paul Schwabe: revision.
15     pragma external;
16
17     procedure Compile_pascal(
18         source_code: Device_Defs.opened_device;
19         -- Opened on source code input file, with read
20         -- rights.
21         machine_code: Device_Defs.opened_device;
22         -- Opened on machine code output file, with read
23         -- and write rights.
24         listing: Device_Defs.opened_device);
25         -- Opened on listing output file, with write
26         -- rights.
27     --
28     -- Function:
29     --   Compiles a Pascal program.
30     --
31     --   Relies on the caller to handle user
32     --   interaction.
33
34 end Compiler_Ex;
```

## X-A.6.5 Compiler\_Ex Package Body

```

1  with Byte_Stream_AM,
2      Device_Defs,
3      Event_Mgt,
4      Pipe_Mgt,
5      Process_Mgt,
6      Process_Mgt_Types,
7      System;
8
9  package body Compiler_Ex is
10     --
11     -- Logic:
12     --   Speeds up a Pascal compiler by dividing parsing
13     --   and code generation between two processes
14     --   connected by a pipe.
15     --
16     --   "Parse" and "Code_gen" are the initial
17     --   procedures of the two child processes.
18     --
19     -- History:
20     --   11-24-87, Paul Schwabe: Initial version.
21     --   11-25-87, Gary Taylor : Added tagged comment lines.
22     --
23     -- End of Header
24
25
26  type connection_record is record
27     -- A "connection_record" contains the I/O
28     -- connections used by the two child processes.
29     -- The entire record is passed to both children.
30
31     source_code: Device_Defs.opened_device;
32     -- input file
33     machine_code: Device_Defs.opened_device;
34     -- output file
35     listing: Device_Defs.opened_device;
36     -- output file
37     parse_out: Device_Defs.opened_device;
38     -- output to pipe
39     code_gen_in: Device_Defs.opened_device;
40     -- input from pipe
41  end record;
42
43
44  procedure Parse(
45     param_buffer: System.address;
46     -- Address of connection record.
47     param_length: System.ordinal)
48     -- Not used in this procedure, but required for
49     -- process's initial procedure.
50  --
51  -- Logic:
52  --   Do Pascal parsing using the I/O connections
53  --   specified in the "conn_rec" parameter record.
54  is
55     conn_rec: connection_record; -- Record containing
56     -- parameters.
57     FOR conn_rec USE AT param_buffer;
58  begin
59     -- Code to parse "conn_rec.source_code" and write
60     -- parsed stream to "conn_rec.parse_out" and listing
61     -- to "conn_rec.listing" goes here.
62     null;
63  end Parse;
64  pragma subprogram_value(Process_Mgt.Initial_proc, Parse);
65
66
67  procedure Code_gen(
68     param_buffer: System.address;
69     -- Address of connection record.
70     param_length: System.ordinal)
71     -- Not used but required for process's initial
72     -- procedure.
73  --
74  -- Logic:

```

## PRELIMINARY

```

75  -- Do Pascal code generation using the I/O
76  -- connections specified in the "conn_rec"
77  -- parameter record.
78  is
79  conn_rec: connection_record;
80  -- Record containing parameters.
81  FOR conn_rec USE AT param_buffer;
82  begin
83  -- Code to read "conn_rec.code_gen_in", write
84  -- compiled code to "conn_rec.machine_code", and add
85  -- any needed messages to "cr.listing" goes here.
86  null;
87  end Code_gen;
88  pragma subprogram_value(
89  Process_Mgt.Initial_proc,
90  Code_gen);
91
92
93  procedure Compile_pascal(
94  source_code: Device_Defs.opened_device;
95  machine_code: Device_Defs.opened_device;
96  listing: Device_Defs.opened_device)
97  --
98  -- Logic:
99  --
100 -- 1. Create a pipe.
101 --
102 -- 2. Create a record specifying all I/O
103 -- connections for child processes. Open both
104 -- ends of the pipe to create the pipe
105 -- connections needed.
106 --
107 -- 3. Get an AD for this process from process
108 -- globals.
109 --
110 -- 4. Spawn the parsing process. The parameter
111 -- buffer address is the connection record's
112 -- address. The termination action signals the
113 -- "user_1" event to this process.
114 --
115 -- 5. Spawn the code generation process. The
116 -- parameter buffer address is the connection
117 -- record's address. The termination action
118 -- signals the "user_2" event to this process.
119 --
120 -- 6. Wait for both the "user_1" and "user_2"
121 -- events to be signalled indicating that both
122 -- child processes have terminated.
123 --
124 -- 7. Deallocate both child processes.
125 --
126 -- Notes:
127 -- No check is made for abnormal termination of
128 -- the child processes.
129 --
130 -- Would like to deallocate pipe when done with it
131 -- but "Pipe_Mgt" does not provide a "Deallocate"
132 -- call.
133  is
134  compiler_pipe: Pipe_Mgt.pipe_AD;
135  -- Pipe that connects "Parse" and "Code_gen"
136  -- processes.
137
138  conn_rec: connection_record;
139  -- Record referencing all I/O connections used by
140  -- the child processes.
141
142  this_process_untyped: System.untyped_word;
143  -- Process executing call to "Compile_pascal",
144  -- as an "untyped_word".
145
146  parse_process: Process_Mgt.Types.process_AD;
147  -- Process executing "Parse".
148
149  code_gen_process: Process_Mgt.Types.process_AD;
150  -- Process executing "Code_gen".
151

```

## PRELIMINARY

```
152 term_events: Event_Mgt.action_record_list(2);
153 -- Array that receives termination events of the
154 -- two child processes.
155
156 begin
157   compiler_pipe := Pipe_Mgt.Create_pipe;
158
159   conn_rec := (
160     source_code => source_code,
161     machine_code => machine_code,
162     listing => listing,
163     parse_out => Byte_Stream_AM.Ops.Open(
164       Pipe_Mgt.Convert_pipe_to_device(
165         compiler_pipe),
166       Device_Defs.output),
167     code_gen_in => Byte_Stream_AM.Ops.Open(
168       Pipe_Mgt.Convert_pipe_to_device(
169         compiler_pipe),
170       Device_Defs.input));
171
172   this_process_untyped :=
173     Process_Mgt.Get_process_globals_entry(
174       Process_Mgt.Types.process);
175
176   parse_process := Process_Mgt.Spawn_process(
177     init_proc => Parse'subprogram_value,
178     param_buffer => conn_rec'address,
179     term_action => (
180       event => Event_Mgt.user_1,
181       message => System.null_address,
182       destination => this_process_untyped));
183
184   code_gen_process := Process_Mgt.Spawn_process(
185     init_proc => Code_gen'subprogram_value,
186     param_buffer => conn_rec'address,
187     term_action => (
188       event => Event_Mgt.user_2,
189       message => System.null_address,
190       destination => this_process_untyped));
191
192   Event_Mgt.Wait_for_all(
193     events =>
194       (Event_Mgt.user_1 .. Event_Mgt.user_2 =>
195         true,
196         others => false),
197     action_list => term_events);
198
199   -- These process are terminated so
200   -- "Deallocate" should work.
201   Process_Mgt.Deallocate(parse_process);
202   Process_Mgt.Deallocate(code_gen_process);
203
204 end Compile_pascal;
205
206
207 end Compiler_Ex;
```

## X-A.6.6 Conversion\_Support\_Ex Package Specification

```

1  with Attribute_Mgt,
2     Authority_List_Mgt,
3     Data_Definition_Mgt,
4     Device_Defs,
5     Directory_Mgt,
6     Event_Mgt,
7     File_Defs,
8     Identification_Mgt,
9     Identification_Mgt,
10    Job_Types,
11    Name_Space_Mgt,
12    Object_Mgt,
13    Object_Mgt,
14    Passive_Store_Mgt,
15    Pipe_Mgt,
16    Process_Mgt_Types,
17    Session_Types,
18    System_Defs,
19    System,
20    Unchecked_conversion;
21
22 package Conversion_Support_Ex is
23 --
24 -- Function:
25 -- Provides commonly needed compile-time type
26 -- conversions for OS access types.
27 --
28 -- Some OS calls can operate on many different
29 -- object types. Such calls require or return
30 -- values of type "System.untyped_word", used to
31 -- hold any AD. If your application uses ADs with
32 -- more specific types, you must convert those
33 -- types to and from "System.untyped_word". For
34 -- example, to store an AD to a Type Definition
35 -- Object in a directory, you must convert from
36 -- the type "Object_Mgt.TDO_AD" to
37 -- "System.untyped_word".
38 --
39 -- All the conversion routines in this package are
40 -- instantiations of the "Unchecked_conversion"
41 -- generic Ada function. Calls to the conversion
42 -- routines are processed at compile-time, and
43 -- have no runtime cost.
44 --
45 -- There are a few conversions that don't require
46 -- using a conversion routine. For example,
47 -- "Device_Defs.device" is a subtype of
48 -- "System.untyped_word". This package still
49 -- provides the expected conversion routines--they
50 -- have no runtime cost, and by using them you do
51 -- not have to remember which types don't require
52 -- conversion.
53 --
54 -- The conversion function names have the form
55 -- "X_from_Y" where "X" indicates the result type
56 -- and "Y" indicates the source type.
57 --
58 -- History:
59 -- 06-03-87, Martin L. Buchanan: Initial version.
60 -- 06-09-87, Paul Schwabe: Added full set of
61 -- unchecked_conversions.
62 -- 11-23-87, Paul Schwabe: Fixed line sizes.
63 --
64 -- End of Header
65 pragma external;
66
67 function Attribute_ID_from_untyped is new
68   Unchecked_conversion(
69     source => System.untyped_word,
70     target => Attribute_Mgt.attribute_ID_AD);
71
72
73 function Untyped_from_attribute_ID is new
74   Unchecked_conversion(

```

## PRELIMINARY

```
75         source => Attribute_Mgt.attribute_ID_AD,
76         target => System.untyped_word);
77
78
79 function Authority_list_from_untyped is new
80     Unchecked_conversion(
81         source => System.untyped_word,
82         target => Authority_List_Mgt.
83             authority_list_AD);
84
85
86 function Untyped_from_authority_list is new
87     Unchecked_conversion(
88         source => Authority_List_Mgt.
89             authority_list_AD,
90         target => System.untyped_word);
91
92
93 function DDef_from_untyped is new
94     Unchecked_conversion(
95         source => System.untyped_word,
96         target => Data_Definition_Mgt.DDef_AD);
97
98
99 function Untyped_from_DDef is new
100     Unchecked_conversion(
101         source => Data_Definition_Mgt.DDef_AD,
102         target => System.untyped_word);
103
104
105 function Device_from_untyped is new
106     Unchecked_conversion(
107         source => Device_Defs.device,
108         target => Authority_List_Mgt.
109             authority_list_AD);
110
111
112 function Untyped_from_device is new
113     Unchecked_conversion(
114         source => Authority_List_Mgt.
115             authority_list_AD,
116         target => Device_Defs.device);
117
118
119 function Opened_device_from_untyped is new
120     Unchecked_conversion(
121         source => System.untyped_word,
122         target => Device_Defs.opened_device);
123
124
125 function Untyped_from_opened_device is new
126     Unchecked_conversion(
127         source => Device_Defs.opened_device,
128         target => System.untyped_word);
129
130
131 function Directory_from_untyped is new
132     Unchecked_conversion(
133         source => System.untyped_word,
134         target => Directory_Mgt.directory_AD);
135
136
137 function Untyped_from_directory is new
138     Unchecked_conversion(
139         source => Directory_Mgt.directory_AD,
140         target => System.untyped_word);
141
142
143 function Event_cluster_from_untyped is new
144     Unchecked_conversion(
145         source => System.untyped_word,
146         target => Event_Mgt.event_cluster_AD);
147
148
149 function Untyped_from_event_cluster is new
150     Unchecked_conversion(
151         source => Event_Mgt.event_cluster_AD,
```

## PRELIMINARY

```
152         target => System.untyped_word);
153
154
155 function File_from_untyped is new
156   Unchecked_conversion(
157     source => System.untyped_word,
158     target => File_Defs.file_AD);
159
160
161 function Untyped_from_file is new
162   Unchecked_conversion(
163     source => File_Defs.file_AD,
164     target => System.untyped_word);
165
166
167 function ID_from_untyped is new
168   Unchecked_conversion(
169     source => System.untyped_word,
170     target => Identification_Mgt.ID_AD);
171
172
173 function Untyped_from_ID is new
174   Unchecked_conversion(
175     source => Identification_Mgt.ID_AD,
176     target => System.untyped_word);
177
178
179 function ID_list_from_untyped is new
180   Unchecked_conversion(
181     source => System.untyped_word,
182     target => Identification_Mgt.ID_list_AD);
183
184
185 function Untyped_from_ID_list is new
186   Unchecked_conversion(
187     source => Identification_Mgt.ID_list_AD,
188     target => System.untyped_word);
189
190
191 function Job_from_untyped is new
192   Unchecked_conversion(
193     source => System.untyped_word,
194     target => Job_Types.job_AD);
195
196
197 function Untyped_from_job is new
198   Unchecked_conversion(
199     source => Job_Types.job_AD,
200     target => System.untyped_word);
201
202
203 function Name_space_from_untyped is new
204   Unchecked_conversion(
205     source => System.untyped_word,
206     target => Name_Space_Mgt.name_space_AD);
207
208
209 function Untyped_from_name_space is new
210   Unchecked_conversion(
211     source => Name_Space_Mgt.name_space_AD,
212     target => System.untyped_word);
213
214
215 function SRO_from_untyped is new
216   Unchecked_conversion(
217     source => System.untyped_word,
218     target => Object_Mgt.SRO_AD);
219
220
221 function Untyped_from_SRO is new
222   Unchecked_conversion(
223     source => Object_Mgt.SRO_AD,
224     target => System.untyped_word);
225
226
227 function TDO_from_untyped is new
228   Unchecked_conversion(
```

## PRELIMINARY

```
229         source => System.untyped_word,
230         target => Object_Mgt.TDO_AD);
231
232
233     function Untyped_from_TDO is new
234         Unchecked_conversion(
235             source => Object_Mgt.TDO_AD,
236             target => System.untyped_word);
237
238
239     function PSM_attributes_from_untyped is new
240         Unchecked_conversion(
241             source => System.untyped_word,
242             target => Passive_Store_Mgt.
243                 PSM_attributes_AD);
244
245
246     function Untyped_from_PSM_attributes is new
247         Unchecked_conversion(
248             source => Passive_Store_Mgt.
249                 PSM_attributes_AD,
250             target => System.untyped_word);
251
252
253     function Pipe_from_untyped is new
254         Unchecked_conversion(
255             source => System.untyped_word,
256             target => Pipe_Mgt.pipe_AD);
257
258
259     function Untyped_from_pipe is new
260         Unchecked_conversion(
261             source => Pipe_Mgt.pipe_AD,
262             target => System.untyped_word);
263
264
265     function Process_from_untyped is new
266         Unchecked_conversion(
267             source => System.untyped_word,
268             target => Process_Mgt_Types.process_AD);
269
270
271     function Untyped_from_process is new
272         Unchecked_conversion(
273             source => Process_Mgt_Types.process_AD,
274             target => System.untyped_word);
275
276
277     function Session_from_untyped is new
278         Unchecked_conversion(
279             source => System.untyped_word,
280             target => Session_Types.session_AD);
281
282
283     function Untyped_from_session is new
284         Unchecked_conversion(
285             source => Session_Types.session_AD,
286             target => System.untyped_word);
287
288
289     function Text_from_untyped is new
290         Unchecked_conversion(
291             source => System.untyped_word,
292             target => System_Defs.text_AD);
293
294
295     function Untyped_from_text is new
296         Unchecked_conversion(
297             source => System_Defs.text_AD,
298             target => System.untyped_word);
299
300
301 end Conversion_Support_Ex;
```

## X-A.6.7 Memory\_ex Procedure

```

1  with Object_Mgt,
2      Long_Integer_Defs,
3      SRO_Mgt,
4      System_Defs;
5
6  procedure Memory_ex
7      --
8      -- Function:
9      --   Provide examples of several memory management
10     --   programming techniques.
11
12  is
13     -- Declare a record for a job's memory
14     -- information:
15     --
16     job_memory_info: SRO_Mgt.SRO_information;
17  begin
18
19     -- Get current memory information for the calling
20     -- job:
21     --
22     job_memory_info := SRO_Mgt.Read_SRO_information;
23
24
25     -- Shrink the calling process's stack to the
26     -- size currently used. The stack can still
27     -- grow and will be expanded as needed.
28     --
29     Object_Mgt.Trim_stack;
30
31
32     -- Force a local garbage collection run to start
33     -- immediately in the calling job:
34     --
35     SRO_Mgt.Start_GCOL;
36
37
38     -- Configure a local garbage collection daemon
39     -- to run in the calling job when it has used
40     -- 50% of its storage claim OR 50% of its object
41     -- table page claim, AND at least 5 minutes
42     -- has elapsed since a previous local GCOL run
43     -- in the job.
44     --
45     SRO_Mgt.Start_GCOL(
46         storage_claim_percent => 50,
47         OTP_claim_percent     => 50,
48         minimum_delay         =>
49             Long_Integer_Defs."*(
50                 Long_Integer_Defs.long_integer'(0, 5),
51                 System_Defs.stu_per_min));
52
53
54     -- Kill any local garbage collection daemon in
55     -- the calling job. (Does nothing if there
56     -- is no daemon.)
57     --
58     SRO_Mgt.Start_GCOL(0, 0, Long_Integer_Defs.max_int);
59
60  end Memory_ex;

```

## X-A.6.8 Process\_Globals\_Support\_Ex Package Specification

```

1  with Authority_List_Mgt,
2     Device_Defs,
3     Directory_Mgt,
4     Identification_Mgt,
5     Job_Types,
6     Name_Space_Mgt,
7     Process_Mgt_Types,
8     Session_Types,
9     System_Defs;
10
11 package Process_Globals_Support_Ex is
12
13     -- Function:
14     -- Provide calls to get and set commonly used
15     -- process globals entries, for the calling
16     -- process.
17     --
18     -- See "Process_Mgt_Types" for descriptions of all
19     -- process globals entries.
20     --
21     --
22     -- << What You Get with This Package >>
23     --
24     -- There are three advantages to using this
25     -- package, as compared to using the "Process_Mgt"
26     -- calls to get and set process globals:
27     --
28     -- 1. The underlying calls require or return
29     -- untyped words. You must instantiate
30     -- "Unchecked_conversion" to convert to and from
31     -- the types you actually need, such as
32     -- "Device_Defs.opened_device".
33     --
34     -- 2. You don't have to supply a value of type
35     -- "Process_Mgt_Types.process_globals_entry" that
36     -- specifies the process globals *slot* you are
37     -- manipulating.
38     --
39     -- 3. The underlying calls can be used to stuff
40     -- garbage into process globals entries and later
41     -- return that garbage. The calls in this
42     -- package do reasonable checks on type, rights,
43     -- and object state for the modifiable process
44     -- globals entries. Such checks aren't needed for
45     -- the non-modifiable entries, assigned by
46     -- the OS.
47     --
48     --
49     -- << What You Don't Get with This Package >>
50     --
51     -- This package does not support assigning or
52     -- retrieving null values for the modifiable
53     -- process globals entries. You can assign and
54     -- retrieve null values for these entries using
55     -- "Process_Mgt" calls.
56     --
57     -- This package does not support getting or
58     -- setting another process's globals. You can
59     -- access another process's globals by using
60     -- "Process_Mgt" or "Process_Admin" calls.
61     --
62     -- This package does not support setting any
63     -- process globals entries that can only be set by
64     -- an administrative interface, such as
65     -- "Process_Admin".
66     --
67     -- This package is selective, and does not provide
68     -- calls to get or set every publicly accessible
69     -- entry.
70     --
71     -- Exceptions:
72     -- user_dialog_not_interactive
73     --
74     --

```

## PRELIMINARY

```
75 -- History:
76 --   06-03-87, Martin L. Buchanan: Initial version.
77 --   11-23-87, Paul Schwabe: Updated spec.
78 --
79 -- End of Header
80 pragma external;
81
82 function Get_standard_input
83   return Device_Defs.opened_device;
84   -- The calling process's standard
85   -- input opened device,
86   -- open and with read rights.
87   --
88   -- Function:
89   -- Returns the calling process's standard input.
90   --
91   -- Exceptions:
92   -- Device_Defs.device_not_open -
93   -- The opened device has been closed.
94
95
96 procedure Set_standard_input(
97   opened_dev: Device_Defs.opened_device);
98   -- Opened device, open and with read rights.
99   --
100  -- Function:
101  -- Assigns the calling process's standard input.
102  --
103  -- Exceptions:
104  -- Device_Defs.device_not_open -
105  -- The opened device has been closed.
106
107
108 function Get_standard_output
109   return Device_Defs.opened_device;
110   -- The calling process's standard
111   -- output opened device,
112   -- open and with write rights.
113   --
114   -- Function:
115   -- Returns the calling process's standard output.
116   --
117   -- Exceptions:
118   -- Device_Defs.device_not_open -
119   -- The opened device has been closed.
120
121
122 procedure Set_standard_output(
123   opened_dev: Device_Defs.opened_device);
124   -- Opened device, open and with write rights.
125   --
126   -- Function:
127   -- Assigns the calling process's standard output.
128   --
129   -- Exceptions:
130   -- Device_Defs.device_not_open -
131   -- The opened device has been closed.
132
133
134 function Get_standard_message
135   return Device_Defs.opened_device;
136   -- The calling process's standard
137   -- message opened device,
138   -- open and with write rights.
139   --
140   -- Function:
141   -- Returns the calling process's standard message
142   -- opened device.
143   --
144   -- Exceptions:
145   -- Device_Defs.device_not_open -
146   -- The opened device has been closed.
147
148
149 procedure Set_standard_message(
150   opened_dev: Device_Defs.opened_device);
151   -- Opened device, open and with write rights.
```

## PRELIMINARY

```
152      --
153      -- Function:
154      --   Assigns the calling process's standard
155      --   message opened device.
156      --
157      -- Exceptions:
158      --   Device_Defs.device_not_open -
159      --   The opened device has been closed.
160
161
162 function Get_user_dialog
163   return Device_Defs.opened_device;
164   -- The calling process's user
165   -- dialog opened device, open, with the
166   -- "is_interactive" flag set in the
167   -- underlying device's information record,
168   -- and with both read and write rights.
169   --
170   -- Function:
171   --   Returns the calling process's
172   --   user dialog opened device.
173   --
174   -- Exceptions:
175   --   Device_Defs.device_not_open -
176   --   The opened device has been closed.
177
178
179 procedure Set_user_dialog(
180   opened_dev: Device_Defs.opened_device);
181   -- An opened device that is open, with the
182   -- "is_interactive" flag set in the underlying
183   -- device's information record, and with both
184   -- read and write rights.
185   --
186   -- Function:
187   --   Assigns the calling process's user dialog
188   --   opened device.
189   --
190   -- Exceptions:
191   --   Device_Defs.device_not_open -
192   --   The opened device has been closed.
193
194
195 function Get_home_directory
196   return Directory_Mgt.directory_AD;
197   -- The calling process's home directory.
198   --
199   -- Function:
200   --   Returns the calling process's home directory.
201   --
202   -- Notes:
203   --   Setting a process's home directory is an
204   --   administrative operation.
205
206
207 function Get_current_directory
208   return Directory_Mgt.directory_AD;
209   -- The calling process's current directory.
210   --
211   -- Function:
212   --   Returns the calling process's current
213   --   directory.
214
215
216 procedure Set_current_directory(
217   dir: Directory_Mgt.directory_AD);
218   -- Any directory.
219   --
220   -- Function:
221   --   Assigns the calling process's current
222   --   directory.
223
224
225 function Get_authority_list
226   return Authority_List_Mgt.authority_list_AD;
227   -- The calling process's authority list.
228   --
```

## PRELIMINARY

```
229     -- Function:
230     -- Returns the calling process's authority list.
231
232
233 procedure Set_authority_list(
234   auth: Authority_List_Mgt.authority_list_AD);
235     -- Any authority list.
236     --
237     -- Function:
238     -- Assigns the calling process's default
239     -- authority list.
240
241
242 function Get_ID_list
243   return Identification_Mgt.ID_list_AD;
244     -- The calling process's ID list.
245     --
246     -- Function:
247     -- Returns the calling process's ID list.
248     --
249     -- Notes:
250     -- Setting a process's ID list is an
251     -- administrative operation.
252
253
254 function Get_command_name_space
255   return Name_Space_Mgt.name_space_AD;
256     -- The calling process's command name space.
257     --
258     -- Function:
259     -- Returns the calling process's command name
260     -- space.
261
262
263 procedure Set_command_name_space(
264   ns: Name_Space_Mgt.name_space_AD);
265     -- Any name space.
266     --
267     -- Function:
268     -- Assigns the calling process's command name
269     -- space.
270
271
272 function This_process
273   return Process_Mgt_Types.process_AD;
274     -- The calling process, with control rights.
275     --
276     -- Function:
277     -- Returns the calling process.
278
279
280 function Get_parent_process
281   return Process_Mgt_Types.process_AD;
282     -- Parent process of the calling process, with
283     -- control rights. Null if the calling
284     -- process is the initial process of its job.
285     --
286     -- Function:
287     -- Returns the calling process's parent process,
288     -- if any.
289
290
291 function This_job
292   return Job_Types.job_AD;
293     -- Job that contains the calling process, with
294     -- list and control rights.
295     --
296     -- Function:
297     -- Returns the calling job.
298
299
300 function This_session
301   return Session_Types.session_AD;
302     -- Session that contains the calling job, with
303     -- list and control rights.
304     --
305     -- Function:
```

PRELIMINARY

```
306      -- Returns the caller's session.
307
308
309  function Get_process_name
310      return System_Defs.text_AD;
311      -- AD to text record containing the calling
312      -- process's name.
313      --
314      -- Function:
315      -- Returns the calling process's symbolic name.
316      --
317      -- The symbolic name may be a null text record.
318
319
320  procedure Set_process_name(
321      name: System_Defs.text);
322      -- A text record containing a name for the
323      -- process. The text record must be valid,
324      -- with a "length" field less than or equal
325      -- to its "max_length" field.
326      --
327      -- Function:
328      -- Assigns the calling process's symbolic name.
329      --
330      -- Exceptions:
331      -- System_Exceptions.bad_parameter
332      --
333
334  end Process_Globals_Support_Ex;
```

## X-A.6.9 Process\_Globals\_Support\_Ex Package Body

```

1  with Access_Mgt,
2     Authority_List_Mgt,
3     Byte_Stream_AM,
4     Device_Defs,
5     Directory_Mgt,
6     Identification_Mgt,
7     Job_Mgt,
8     Job_Types,
9     Name_Space_Mgt,
10    Process_Mgt,
11    Process_Mgt_Types,
12    Session_Mgt,
13    Session_Types,
14    System_Defs,
15    System_Exceptions,
16    System;
17
18  package body Process_Globals_Support_Ex is
19
20    -- Function:
21    --   Provide calls to get and set commonly used
22    --   process globals entries, for the calling
23    --   process.
24    --
25    -- History:
26    --   06-10-87, Paul Schwabe:  Initial version.
27    --   11-24-87, Paul Schwabe:  Updated version.
28    --   11-25-87, Gary Taylor :  Added tagged comment lines.
29    --
30    -- End of Header
31
32
33  function Get_standard_input
34    return Device_Defs.opened_device
35    --
36    -- Logic:
37    --   1. Get the process globals entry.
38    --   2. Check that the standard input is open,
39    --      which implicitly checks that its an opened
40    --      device.
41    --   3. Check that the standard input has
42    --      read rights.
43    --   4. Return the standard input.
44  is
45    stdin:          Device_Defs.opened_device;
46    stdin_untyped: System.untyped_word;
47    FOR stdin_untyped USE AT stdin'address;
48  begin
49    stdin_untyped := Process_Mgt.
50      Get_process_globals_entry(
51        Process_Mgt_Types.standard_input);
52
53    if not Byte_Stream_AM.Ops.Is_open(stdin) then
54      RAISE Device_Defs.device_not_open;
55
56    elsif not Access_Mgt.Permits(
57      AD => stdin_untyped,
58      rights => Device_Defs.read_rights) then
59      RAISE System_Exceptions.insufficient_type_rights;
60
61    else
62      RETURN stdin;
63
64    end if;
65  end Get_standard_input;
66
67
68  procedure Set_standard_input(
69    opened_dev: Device_Defs.opened_device)
70    --
71    -- Logic:
72    --   1. Check that the new standard input is open,
73    --      which implicitly checks that its an opened
74    --      device.

```

PRELIMINARY

```

75     -- 2. Check that that the new standard
76     --   input has read rights.
77     -- 3. Set the new standard input.
78   is
79     stdin_untyped: System.untyped_word;
80     FOR stdin_untyped USE AT opened_dev'address;
81   begin
82     if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
83       RAISE Device_Defs.device_not_open;
84
85     elsif not Access_Mgt.Permits(
86       AD => stdin_untyped,
87       rights => Device_Defs.read_rights) then
88       RAISE System_Exceptions.insufficient_type_rights;
89
90     else Process_Mgt.Set_process_globals_entry(
91       slot => Process_Mgt_Types.standard_input,
92       value => stdin_untyped);
93     end if;
94
95   end Set_standard_input;
96
97
98   function Get_standard_output
99     return Device_Defs.opened_device
100    --
101    -- Logic:
102    -- 1. Get the process globals entry.
103    -- 2. Check that the new standard output is open,
104    --   which implicitly checks that its an opened
105    --   device.
106    -- 3. Check that the standard output has
107    --   read rights.
108    -- 4. Return the new standard output.
109  is
110    stdout: Device_Defs.opened_device;
111    stdout_untyped: System.untyped_word;
112    FOR stdout_untyped USE AT stdout'address;
113  begin
114    stdout_untyped := Process_Mgt.
115      Get_process_globals_entry(
116        Process_Mgt_Types.standard_output);
117
118    if not Byte_Stream_AM.Ops.Is_open(stdout) then
119      RAISE Device_Defs.device_not_open;
120
121    elsif not Access_Mgt.Permits(
122      AD => stdout_untyped,
123      rights => Device_Defs.write_rights) then
124      RAISE System_Exceptions.insufficient_type_rights;
125
126    else
127      RETURN stdout;
128
129    end if;
130
131  end Get_standard_output;
132
133
134  procedure Set_standard_output(
135    opened_dev: Device_Defs.opened_device)
136  --
137  -- Logic:
138  -- 1. Check that the new standard output is
139  --   open, which implicitly checks that its an
140  --   opened device.
141  -- 2. Check that that the new standard output
142  --   has write rights.
143  -- 3. Set the new standard output.
144  is
145    stdout_untyped: System.untyped_word;
146    FOR stdout_untyped USE AT
147      opened_dev'address;
148  begin
149    if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
150      RAISE Device_Defs.device_not_open;
151

```

**PRELIMINARY**

```

152     elsif not Access_Mgt.Permits(
153         AD      => stdout_untyped,
154         rights => Device_Defs.write_rights) then
155         RAISE System_Exceptions.insufficient_type_rights;
156
157     else Process_Mgt.Set_process_globals_entry(
158         slot => Process_Mgt_Types.standard_output,
159         value => stdout_untyped);
160     end if;
161
162 end Set_standard_output;
163
164
165 function Get_standard_message
166     return Device_Defs.opened_device
167     --
168     -- Logic:
169     -- 1. Get the process globals entry.
170     -- 2. Check that the standard message
171     --    output is open, which implicitly
172     --    checks that its an opened device.
173     -- 3. Check that the standard message
174     --    output has write rights.
175     -- 4. Return the standard message output.
176 is
177     stdmsg: Device_Defs.opened_device;
178     stdmsg_untyped: System.untyped_word;
179     FOR stdmsg_untyped USE AT
180         stdmsg'address;
181 begin
182     stdmsg_untyped := Process_Mgt.
183         Get_process_globals_entry(
184             Process_Mgt_Types.standard_message);
185
186     if not Byte_Stream_AM.Ops.Is_open(stdmsg) then
187         RAISE Device_Defs.device_not_open;
188
189     elsif not Access_Mgt.Permits(
190         AD      => stdmsg_untyped,
191         rights => Device_Defs.write_rights)
192     then
193         RAISE System_Exceptions.insufficient_type_rights;
194
195     else
196         RETURN stdmsg;
197
198     end if;
199 end Get_standard_message;
200
201
202 procedure Set_standard_message(
203     opened_dev: Device_Defs.opened_device)
204     --
205     -- Logic:
206     -- 1. Check that the new standard message
207     --    output is open, which implicitly checks
208     --    that its an opened device.
209     -- 2. Check that that the new standard
210     --    message has write rights.
211     -- 3. Set the new standard message output.
212 is
213     stdmsg_untyped: System.untyped_word;
214     FOR stdmsg_untyped USE AT
215         opened_dev'address;
216 begin
217     if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
218         RAISE Device_Defs.device_not_open;
219
220     elsif not Access_Mgt.Permits(
221         AD      => stdmsg_untyped,
222         rights => Device_Defs.write_rights) then
223         RAISE System_Exceptions.insufficient_type_rights;
224
225     else Process_Mgt.Set_process_globals_entry(
226         slot => Process_Mgt_Types.standard_message,
227         value => stdmsg_untyped);
228     end if;

```

**PRELIMINARY**

```

229
230 end Set_standard_message;
231
232
233 function Get_user_dialog
234   return Device_Defs.opened_device
235   --
236   -- Logic:
237   --   1. Get the process globals entry.
238   --   2. Check that the user dialog is open,
239   --      which implicitly checks that its an
240   --      opened device.
241   --   3. Check that the user dialog has
242   --      read and write rights.
243   --   4. Return the user dialog.
244 is
245   user_dialog:      Device_Defs.opened_device;
246   user_dialog_untyped: System.untyped_word;
247   FOR user_dialog_untyped USE AT
248     user_dialog'address;
249 begin
250   user_dialog_untyped := Process_Mgt.
251     Get_process_globals_entry(
252       Process_Mgt_Types.user_dialog);
253
254   if not Byte_Stream_AM.Ops.Is_open(user_dialog) then
255     RAISE Device_Defs.device_not_open;
256
257   elsif not Access_Mgt.Permits(
258     AD => user_dialog_untyped,
259     rights => Device_Defs.read_write_rights)
260   then
261     RAISE System_Exceptions.insufficient_type_rights;
262
263   else
264     RETURN user_dialog;
265
266   end if;
267
268 end Get_user_dialog;
269
270
271 procedure Set_user_dialog(
272   opened_dev: Device_Defs.opened_device)
273   --
274   -- Logic:
275   --   1. Check that the new user_dialog is open,
276   --      which implicitly checks that its an opened
277   --      device.
278   --   2. Check that that the new user dialog has
279   --      read and write rights.
280   --   3. Set the new standard message.
281 is
282   user_dialog_untyped: System.untyped_word;
283   FOR user_dialog_untyped USE AT
284     opened_dev'address;
285 begin
286   if not Byte_Stream_AM.Ops.Is_open(opened_dev) then
287     RAISE Device_Defs.device_not_open;
288
289   elsif not Access_Mgt.Permits(
290     AD => user_dialog_untyped,
291     rights => Device_Defs.read_write_rights)
292   then
293     RAISE System_Exceptions.insufficient_type_rights;
294
295   else Process_Mgt.Set_process_globals_entry(
296     slot => Process_Mgt_Types.user_dialog,
297     value => user_dialog_untyped);
298   end if;
299
300 end Set_user_dialog;
301
302
303 function Get_home_directory
304   return Directory_Mgt.directory_AD
305   --

```

```

306     -- Logic:
307     --     1. Get the process globals entry for
308     --     the "home directory."
309     --     2. Check that the entry is a
310     --     directory.
311     --     3. Check that directory has read rights.
312     --     4. Return the directory.
313 is
314     dir:          Directory_Mgt.directory_AD;
315     dir_untyped: System.untyped_word;
316     FOR dir_untyped USE AT
317     dir'address;
318 begin
319     dir_untyped := Process_Mgt.
320     Get_process_globals_entry(
321     Process_Mgt.Types.home_dir);
322
323     if not Directory_Mgt.Is_directory(dir_untyped) then
324     RAISE System_Exceptions.type_mismatch;
325
326     else
327     RETURN dir;
328
329     end if;
330
331 end Get_home_directory;
332
333
334 function Get_current_directory
335     return Directory_Mgt.directory_AD
336     --
337     -- Logic:
338     --     1. Get the process globals entry.
339     --     2. Check that the "current directory"
340     --     is a directory.
341     --     3. Return the current directory.
342 is
343     dir:          Directory_Mgt.directory_AD;
344     dir_untyped: System.untyped_word;
345     FOR dir_untyped USE AT dir'address;
346 begin
347     dir_untyped := Process_Mgt.
348     Get_process_globals_entry(
349     Process_Mgt.Types.current_dir);
350
351     if not Directory_Mgt.Is_directory(dir_untyped) then
352     RAISE System_Exceptions.type_mismatch;
353
354     else
355     RETURN dir;
356
357     end if;
358
359 end Get_current_directory;
360
361
362 procedure Set_current_directory(
363     dir: Directory_Mgt.directory_AD)
364     --
365     -- Logic:
366     --     1. Check that the "current directory" is
367     --     a directory.
368     --     2. Set the new current directory.
369 is
370     dir_untyped: System.untyped_word;
371     FOR dir_untyped USE AT dir'address;
372 begin
373     if not Directory_Mgt.Is_directory(dir_untyped) then
374     RAISE System_Exceptions.type_mismatch;
375
376     else Process_Mgt.Set_process_globals_entry(
377     slot => Process_Mgt.Types.current_dir,
378     value => dir_untyped);
379     end if;
380
381 end Set_current_directory;
382

```

**PRELIMINARY**

```

383
384 function Get_authority_list
385     return Authority_List_Mgt.authority_list_AD
386     --
387     -- Logic:
388     --     1. Get the process globals entry.
389     --     2. Check that the entry is an authority list.
390     --     3. Return the authority list.
391 is
392     auth_list: Authority_List_Mgt.authority_list_AD;
393     auth_list_untyped: System.untyped_word;
394     FOR auth_list_untyped USE AT auth_list'address;
395 begin
396     auth_list_untyped := Process_Mgt.
397         Get_process_globals_entry(
398             Process_Mgt_Types.authority_list);
399
400     if not Authority_List_Mgt.
401         Is_authority_list(auth_list_untyped) then
402         RAISE System_Exceptions.type_mismatch;
403
404     else
405         RETURN auth_list;
406
407     end if;
408
409 end Get_authority_list;
410
411
412 procedure Set_authority_list(
413     auth: Authority_List_Mgt.authority_list_AD)
414     --
415     -- Logic:
416     --     1. Check that "auth" is an authority list.
417     --     2. Set the new authority list.
418 is
419     auth_untyped: System.untyped_word;
420     FOR auth_untyped USE AT auth'address;
421 begin
422     if not Authority_List_Mgt.Is_authority_list(
423         auth_untyped) then
424         RAISE System_Exceptions.Type_mismatch;
425
426     else Process_Mgt.Set_process_globals_entry(
427         slot => Process_Mgt_Types.authority_list,
428         value => auth_untyped);
429     end if;
430
431 end Set_authority_list;
432
433
434 function Get_ID_list
435     return Identification_Mgt.ID_list_AD
436     --
437     -- Logic:
438     --     1. Get the process globals entry.
439     --     2. Check that the entry is an ID list.
440     --     3. Return the ID list entry.
441 is
442     ID_list: Identification_Mgt.ID_list_AD;
443     ID_list_untyped: System.untyped_word;
444     FOR ID_list_untyped USE AT ID_list'address;
445 begin
446     ID_list_untyped := Process_Mgt.
447         Get_process_globals_entry(
448             Process_Mgt_Types.ID_list);
449
450     if not Identification_Mgt.
451         Is_ID_list(ID_list_untyped) then
452         RAISE System_Exceptions.type_mismatch;
453
454     else
455         RETURN ID_list;
456
457     end if;
458
459 end Get_ID_list;

```

PRELIMINARY

```

460
461
462 function Get_command_name_space
463     return Name_Space_Mgt.name_space_AD
464     --
465     -- Logic:
466     --     1. Get the process globals entry.
467     --     2. Check that the entry is a name space.
468     --     3. Return the name space entry.
469 is
470     cmd_name_space :      Name_Space_Mgt.
471                          name_space_AD;
472     cmd_name_space_untyped: System.untyped_word;
473     FOR cmd_name_space_untyped USE AT
474         cmd_name_space'address;
475 begin
476     cmd_name_space_untyped := Process_Mgt.
477         Get_process_globals_entry(
478             Process_Mgt_Types.cmd_name_space);
479
480     if not Name_Space_Mgt.
481         Is_name_space(cmd_name_space_untyped) then
482         RAISE System_Exceptions.type_mismatch;
483
484     else
485         RETURN cmd_name_space;
486
487     end if;
488
489 end Get_command_name_space;
490
491
492 procedure Set_command_name_space(
493     ns: Name_Space_Mgt.name_space_AD)
494     --
495     -- Logic:
496     --     1. Check that "ns" is a name space.
497     --     2. Set the new command name space.
498 is
499     ns_untyped: System.untyped_word;
500     FOR ns_untyped USE AT
501         ns'address;
502 begin
503     if not Name_Space_Mgt.
504         Is_name_space(ns_untyped) then
505         RAISE System_Exceptions.type_mismatch;
506
507     else Process_Mgt.Set_process_globals_entry(
508         slot => Process_Mgt_Types.cmd_name_space,
509         value => ns_untyped);
510     end if;
511
512 end Set_command_name_space;
513
514 function This_process
515     return Process_Mgt_Types.process_AD
516     --
517     -- Logic:
518     --     1. Get the process globals entry
519     --         for the current process.
520     --     2. Return the process.
521 is
522     current_process: Process_Mgt_Types.process_AD;
523     current_process_untyped: System.untyped_word;
524     FOR current_process_untyped USE AT
525         current_process'address;
526 begin
527     current_process_untyped := Process_Mgt.
528         Get_process_globals_entry(
529             Process_Mgt_Types.process);
530
531     RETURN current_process;
532
533 end This_process;
534
535 function Get_parent_process
536     return Process_Mgt_Types.process_AD

```

PRELIMINARY

```

537  --
538  -- Logic:
539  --   1. Get the process globals entry
540  --   for the parent process.
541  --   2. Return the parent process.
542  is
543  parent_process:      Process_Mgt_Types.
544                      process_AD;
545  parent_process_untyped: System.untyped_word;
546  FOR parent_process_untyped USE AT
547  parent_process'address;
548  begin
549  parent_process_untyped := Process_Mgt.
550  Get_process_globals_entry(
551  Process_Mgt_Types.creator);
552
553  RETURN parent_process;
554
555  end Get_parent_process;
556
557  function This_job
558  return Job_Types.job_AD
559  --
560  -- Logic:
561  --   1. Get the process globals
562  --   entry for the current job.
563  --   2. Return the current job.
564  is
565  current_job:         Job_Types.job_AD;
566  current_job_untyped: System.untyped_word;
567  FOR current_job_untyped USE AT
568  current_job'address;
569  begin
570  current_job_untyped := Process_Mgt.
571  Get_process_globals_entry(
572  Process_Mgt_Types.job);
573
574  RETURN current_job;
575
576  end This_job;
577
578
579  function This_session
580  return Session_Types.session_AD
581  --
582  -- Logic:
583  --   1. Get process globals entry
584  --   for the current session.
585  --   2. Return the current session.
586  is
587  current_session:     Session_Types.session_AD;
588  current_session_untyped: System.untyped_word;
589  FOR current_session_untyped USE AT
590  current_session'address;
591  begin
592  current_session_untyped := Process_Mgt.
593  Get_process_globals_entry(
594  Process_Mgt_Types.session);
595
596  RETURN current_session;
597
598  end This_session;
599
600
601  function Get_process_name
602  return System_Defs.text_AD
603  --
604  -- Logic:
605  --   1. Return the name of the current process.
606  is
607  name:                System_Defs.text_AD;
608  name_untyped:        System.untyped_word;
609  FOR name_untyped USE AT name'address;
610  begin
611  name_untyped := Process_Mgt.
612  Get_process_globals_entry(
613  Process_Mgt_Types.name);

```

```
614     RETURN name;
615
616 end Get_process_name;
617
618
619
620 procedure Set_process_name(
621     name: System_Defs.text)
622     --
623     -- Logic:
624     --     1. Check that "name" is a valid text.
625     --     2. Set the new process name.
626 is
627     name_untyped: System.untyped_word;
628     FOR name_untyped USE AT
629         name'address;
630 begin
631     if name.length > name.max_length then
632         RAISE System_Exceptions.bad_parameter;
633     else
634         Process_Mgt.Set_process_globals_entry(
635             slot => Process_Mgt.Types.name,
636             value => name_untyped);
637     end if;
638 end Set_process_name;
639
640 end Set_process_name;
641
642
643 end Process_Globals_Support_Ex;
```

## X-A.6.10 Symbol\_Table\_Ex Package Specification

```

1  package Symbol_Table_Ex is
2  --
3  -- Function:
4  --   Manages a symbol table for use by a compiler or
5  --   other application.
6  --
7  --   Synchronizes concurrent access to the symbol
8  --   table.
9  --
10 --   Symbol names can be no longer than
11 --   "max_symbol_length" characters.
12 --
13 --   There is no limit on the number of symbols in
14 --   the table; it is expanded as needed.
15 --
16 --   The symbol table is created empty at package
17 --   initialization.
18 --
19 -- Notes:
20 --   Nested blocks and symbols local to blocks are
21 --   not supported.
22 --
23 -- Exceptions:
24 --
25 symbol_exists:  exception;
26 -- "Add_symbol" was called with a symbol that is
27 -- already in the table.
28 --
29 no_such_symbol:  exception;
30 -- "Read_symbol_data" was called with a symbol
31 -- that is not in the table.
32 --
33 name_too_long:  exception;
34 -- "Add_symbol" or "Read_symbol_data" was called
35 -- with a symbol name longer than
36 -- "max_symbol_length".
37 --
38 max_symbol_length:  constant positive := 32;
39 -- Maximum symbol length allowed.
40 --
41 -- History:
42 --   11-24-87, Paul Schwabe: updated spec.
43 --
44 -- End of Header
45 pragma external;
46
47 type symbol_data is record
48 -- This type defines the characteristics recorded
49 -- for each symbol in the table. No fields are
50 -- defined for this example package.
51 null;
52 end record;
53
54
55 procedure Add_symbol(
56   name: string;
57   -- Name cannot be in use in the table. Name
58   -- cannot be longer than "max_symbol_length".
59   data: symbol_data);
60 --
61 -- Function:
62 --   Adds a symbol and its data to the symbol
63 --   table.
64 --
65 -- Exceptions:
66 --   symbol_exists
67 --   name_too_long
68
69
70 function Read_symbol_data(
71   name: string)
72 -- Must name a symbol in the table. Name
73 -- cannot be longer than "max_symbol_length".
74 return symbol_data;
```

## PRELIMINARY

```
75      --  
76      -- Function:  
77      -- Reads a symbol's data from the symbol table.  
78      --  
79      -- Exceptions:  
80      --   no_such_symbol  
81      --   name_too_long  
82  
83  
84 end Symbol_Table_Ex;
```

## X-A.6.11 Symbol\_Table\_Ex Package Body

```

1  with Object_Mgt,
2      Semaphore_Mgt,
3      System;
4
5  package body Symbol_Table_Ex is
6      --
7      -- Logic:
8      -- The symbol table is implemented as an object
9      -- containing an array. Because the table is
10     -- dynamically allocated, it can be expanded as
11     -- needed.
12     --
13     -- The "symbol_table.lock" semaphore is used to
14     -- exclude other processes while a process is
15     -- accessing the table. All symbol table
16     -- operations lock ("P") the semaphore before
17     -- accessing the table, and unlock ("V") the
18     -- semaphore before returning or propagating an
19     -- exception.
20     --
21     -- Notes:
22     -- A realistic implementation could be optimized
23     -- for keyed retrieval using a hash table. Such
24     -- an implementation could use the same locking
25     -- code.
26     -- History:
27     -- 11-24-87, Paul Schwabe: updated code.
28     -- 11-25-87, Gary Taylor : Added tagged comment lines.
29     --
30     -- End of Header
31
32     use System; -- Import arithmetic on type "ordinal".
33
34     table_size: constant System.ordinal := 100;
35
36     type symbol_name is array(
37         1 .. max_symbol_length) of character;
38
39     type symbol_entry is record
40         name: symbol_name;
41         data: symbol_data;
42     end record;
43
44     FOR symbol_entry USE
45         record at mod 32;
46     end record;
47
48     type symbol_entry_array is array(
49         System.ordinal range <>) of symbol_entry;
50
51     type symbol_table object (
52         max_length: System.ordinal) is record
53         -- "max_length" is maximum number of entries in a
54         -- full table. Table can still grow by calling
55         -- "Expand_symbol_table".
56         length: System.ordinal;
57         -- Number of entries in use.
58         lock: Semaphore_Mgt.semaphore_AD;
59         -- Used to lock symbol table while a process
60         -- is accessing it.
61         value: symbol_entry_array(1 .. max_length);
62         -- Entries 1 .. "length" contain symbol
63         -- entries.
64     end record;
65
66     type symbol_table_AD is access symbol_table_object;
67     pragma access_kind(symbol_table_AD, AD);
68
69     symbol_table: symbol_table AD;
70     procedure Expand_symbol_table is
71         --
72         -- Operation:
73         -- Doubles the symbol table size.
74         --

```

**PRELIMINARY**

```

75  -- "Expand_symbol_table" is normally called only
76  -- when the symbol table is full.
77  --
78  -- Performs these steps:
79  -- 1. Resizes the symbol table object with space
80  --    for twice as many entries.
81  -- 2. Changes the maximum length of the
82  --    symbol table entry.
83  --
84  -- Notes:
85  -- "Expand_symbol_table" is an internal
86  -- procedure that must be called with the symbol
87  -- table already locked via the associated
88  -- semaphore!
89
90  symbol_table_untyped: System.untyped_word;
91  FOR symbol_table_untyped USE AT
92  symbol_table'address;
93
94  max_length_access: System.ordinal;
95  FOR max_length_access USE AT
96  symbol_table.max_length'address;
97  begin
98  Object_Mgt.Resize(
99  obj => symbol_table_untyped,
100  size => 3 + (2 * symbol_table.max_length * (
101  symbol_entry'size/32)));
102
103  max_length_access := 2 * symbol_table.max_length;
104
105
106  end Expand_symbol_table;
107
108
109  procedure Add_symbol(
110  name: string;
111  data: symbol_data)
112  --
113  -- Logic:
114  -- 1. Surround everything else with a lock on
115  --    "symbol_table.lock". Release the lock
116  --    on all return paths and exception paths.
117  -- 2. Check for "name" too long.
118  -- 3. Convert "name" to "fixed_width_name",
119  --    padding with blanks.
120  -- 4. Search the table and raise an exception if
121  --    the symbol is in the table.
122  -- 5. Otherwise, add the symbol to the end of
123  --    the table, expanding the symbol table if
124  --    it is full.
125  is
126  fixed_width_name: symbol_name := (others => ' ');
127  begin
128  Semaphore_Mgt.P(symbol_table.lock);
129  begin
130  if name'length > max_symbol_length then
131  RAISE name_too_long;
132
133  else
134  fixed_width_name(1 .. name'length) :=
135  symbol_name(name);
136  for i in 1 .. symbol_table.length loop
137  if symbol_table.value(i).name =
138  fixed_width_name then
139  RAISE symbol_exists;
140  end if;
141  end loop;
142  if symbol_table.length =
143  symbol_table.max_length then
144  Expand_symbol_table;
145  end if;
146  symbol_table.length := symbol_table.length + 1;
147  symbol_table.value(symbol_table.length) :=
148  symbol_entry'(fixed_width_name, data);
149  end if;
150
151  exception

```

```

152         when others =>
153             Semaphore_Mgt.V(symbol_table.lock);
154             RAISE;
155             -- Reraise exception that entered handler.
156         end;
157
158     Semaphore_Mgt.V(symbol_table.lock);
159
160
161 end Add_symbol;
162
163
164 function Read_symbol_data(
165     name: string)
166     return symbol_data
167     --
168     -- Logic:
169
170     -- 1. Surround everything else with a lock on
171     --    "symbol_table.lock". Release the lock
172     --    on all return paths and exception paths.
173
174     -- 2. Check for "name" too long.
175
176     -- 3. Convert "name" to "fixed_width_name",
177     --    padding with blanks.
178
179     -- 4. Search the table. If the symbol is found,
180     --    return the symbol data. Otherwise raise
181     --    "no_such_symbol".
182 is
183     fixed_width_name: symbol_name := (others => ' ');
184 begin
185
186     Semaphore_Mgt.P(symbol_table.lock);
187
188     if name'length > max_symbol_length then
189         RAISE name_too_long;
190
191     else
192         fixed_width_name(1 .. name'length) :=
193             symbol_name(name);
194         for i in 1 .. symbol_table.length loop
195             if symbol_table.value(i).name =
196                 fixed_width_name then
197                 Semaphore_Mgt.V(symbol_table.lock);
198                 RETURN symbol_table.value(i).data;
199
200             end if;
201         end loop;
202         RAISE no_such_symbol;
203
204     end if;
205
206     -- This call to "V" is never reached in the
207     -- current implementation. The call is included
208     -- as a safeguard in case code changes make it
209     -- reachable.
210     Semaphore_Mgt.V(symbol_table.lock);
211
212     exception
213     when others =>
214         Semaphore_Mgt.V(symbol_table.lock);
215         RAISE; -- Reraise exception
216             -- that entered handler.
217
218 end Read_symbol_data;
219
220
221 -- PACKAGE INITIALIZATION
222 begin
223     symbol_table := new symbol_table_object(
224         table size);
225     symbol_table.length := 0;
226     -- Symbol table initially has space for 100
227     -- entries with 0 in use.
228

```

**PRELIMINARY**

```
229     symbol_table.lock := Semaphore_Mgt.  
230         Create_semaphore;  
231         -- Lock initially indicates table is available.  
232         -- First "P" on lock will succeed.  
233  
234 end Symbol_Table_Ex;  
235
```

**X-A.6.12 Word\_Processor\_Ex Package Specification**

```
1 package Word_Processor_Ex is
2   --
3   -- Function:
4   --   This example shows how a word processor with a
5   --   spelling checker can use processes and events.
6   --
7   -- End of Header
8   pragma external;
9
10
11 procedure Word_processor;
12   --
13   -- Function:
14   --   Does word processing.
15   --
16   --   Gets its arguments from the command line.
17   --
18   --   Includes a concurrent spelling checker.
19
20
21 end Word_Processor_Ex;
```

## X-A.6.13 Word\_Processor\_Ex Package Body

```

1  with Conversion_Support_Ex,
2      Event_Mgt,
3      Process_Globals_Support_Ex,
4      Process_Mgt,
5      Process_Mgt_Types,
6      System;
7
8  package body Word_Processor_Ex is
9      --
10     -- Logic:
11     -- This example shows how a word processor with a
12     -- concurrent spelling checker uses processes
13     -- and events.
14     --
15     -- The "Word_processor" procedure spawns a
16     -- separate process to execute the
17     -- "Spelling_checker" procedure. Communication
18     -- between the two processes is entirely via
19     -- events.
20     --
21     -- When a word is entered by the word processor
22     -- user, the word processor signals a 'word' event
23     -- to the spelling checker process. That event
24     -- has these
25     -- fields:
26     --
27     --     "event"          - "word_event_value".
28     --
29     --     "message.offset" - Location of word to check,
30     --                       encoded as a 32-bit
31     --                       "word_record".
32     --
33     --     "message.AD"    - AD to word processor
34     --                       process that is signalling
35     --                       the event.
36     --
37     --     "destination"  - AD to spelling checker
38     --                       process that receives
39     --                       the event.
40     --
41     -- Inclusion of an AD to the process that signals
42     -- the event allows a future implementation to use
43     -- the spelling checker process as a server for
44     -- multiple client processes.
45     --
46     -- If a word is misspelled, the spelling checker
47     -- signals a 'spelling error' event to the process
48     -- that requested the spelling check.
49     -- That event has these fields:
50     --
51     --     "event"          - spelling_error_event_value.
52     --
53     --     "message.offset" - Location of word that was
54     --                       checked, encoded as a 32-bit
55     --                       "word_record".
56     --
57     --     "message.AD"    - Not used. In this
58     --                       implementation,
59     --                       is "System.null_word".
60     --
61     --     "destination"  - AD to the word processor
62     --                       process that signalled the word to the spelling
63     --                       checker.
64     --
65     -- The word processor handles spelling error
66     -- events with the "Spelling_error_handler"
67     -- procedure.
68     --
69     -- Notes:
70     -- The "word_record" scheme of communicating words
71     -- to be checked is probably inadequate for an
72     -- implementation of the spelling checker as a
73     -- general server that can be used by multiple
74     -- concurrent applications.

```

PRELIMINARY

```

75  --
76  -- History:
77  --   11-24-87, Paul Schwabe: updated code.
78  --   11-25-87, Gary Taylor : Added tagged comment lines.
79  --
80  -- End of Header
81
82  use System; -- Import operations on ordinal types.
83
84  type word_record is record
85  -- This type encodes a word location into 32 bits,
86  -- allowing a word location to be transmitted
87  -- using the "message.offset" field when an event
88  -- is signalled. The word processor and spelling
89  -- checker are presumed to share a two-dimensional
90  -- array containing the text being edited. Words
91  -- are presumed to not break across lines of the
92  -- array. A word location can thus be specified
93  -- as a line number, a starting column number, and
94  -- an ending column number. The encoding limits
95  -- line numbers to the range 0 .. 65 535 and
96  -- column numbers to the range 0 .. 255.
97  line:      System.short_ordinal;
98  start_col: System.byte_ordinal;
99  end_col:   System.byte_ordinal;
100 end record;
101
102 FOR word_record USE
103   record at mod 32;
104     line      at 0 range 0 .. 15;
105     start_col at 0 range 16 .. 23;
106     end_col   at 0 range 24 .. 31;
107   end record;
108
109
110  -- << Event Values Used >>
111  --
112  -- The following local events can use the same event
113  -- value without conflict because they are always
114  -- signalled to different processes.
115
116  word_event_value:
117    constant Event_Mgt.event_value := Event_Mgt.user_1;
118    -- Local event signalled to spelling checker for
119    -- each word to be checked.
120
121  spelling_error_event_value:
122    constant Event_Mgt.event_value :=
123      Event_Mgt.user_1;
124    -- Local event signalled to client process for
125    -- each misspelled word.
126
127
128  procedure Spelling_checker(
129    param_buffer: System.address;
130    -- Not used but required for process's initial
131    -- procedure.
132    param_length: System.ordinal)
133    -- Not used but required for process's initial
134    -- procedure.
135    -- Operation:
136    --   Loops doing these steps:
137    --   1. Wait for a word event.
138    --   2. Check the word's spelling.
139    --   3. If the word is misspelled, signal a
140    --      spelling error event to whatever
141    --      process requested the check.
142  is
143    word_event:      Event_Mgt.action_record;
144    -- Receives each word to be checked.
145    current_word:    word_record;
146    FOR current_word USE AT word_event.
147      message.offset'address;
148    -- Overlay used to extract word location.,
149    word_misspelled: boolean;
150  begin
151    loop

```

## PRELIMINARY

```
152     Event_Mgt.Wait_for_any(  
153         events => (word_event_value => true,  
154             others => false),  
155         action => word_event);  
156  
157     -- Code to check spelling of current word goes  
158     -- here. The "word_misspelled" flag is a stand-in  
159     -- for whatever conditional expression indicates  
160     -- a misspelled word.  
161  
162     if word_misspelled then  
163         Event_Mgt.Signal(Event_Mgt.action_record' (  
164             event      => spelling_error_event_value,  
165             message    => (  
166                 offset => word_event.message.offset,  
167                 AD     => System.null_word),  
168             destination => word_event.message.AD));  
169     end if;  
170  
171     end loop;  
172  
173 end Spelling_checker;  
174 pragma subprogram_value(Process_Mgt.Initial_proc,  
175     Spelling_checker);  
176  
177  
178 procedure Spelling_error_handler(  
179     action: Event_Mgt.action_record)  
180     --  
181     -- Operation:  
182     -- Handler invoked for each 'spelling error'  
183     -- event.  
184 is  
185     misspelled_word: word_record;  
186     FOR misspelled_word  
187         USE AT action.message.offset'address;  
188         -- Overlay used to extract word location.  
189 begin  
190     -- Code to handle misspelled word goes here. For  
191     -- example, this code could highlight the  
192     -- misspelled word on the display and ring the  
193     -- terminal's bell.  
194  
195     null;  
196 end Spelling_error_handler;  
197 pragma subprogram_value(  
198     Event_Mgt.Event_handler,  
199     Spelling_error_handler);  
200  
201  
202 procedure Word_processor  
203     --  
204     -- Logic:  
205     -- 1. Retrieve an AD for this process, to be  
206     --    passed to the spelling checker so it will  
207     --    know what process to signal if a word is  
208     --    misspelled.  
209     --  
210     -- 2. Create the spelling checker process.  
211     --  
212     -- 3. Establish a handler for the spelling error  
213     --    local event and enable the event. Save the  
214     --    previous event status.  
215     --  
216     -- 4. Loop, doing word processing. For each  
217     --    word that is entered, signal the word event  
218     --    to the spelling checker process.  
219     --  
220     -- 5. When word processing is done, terminate and  
221     --    deallocate the spelling checker process and  
222     --    restore the previous event status for the  
223     --    spelling error local event.  
224 is  
225     spelling_checker_process:  
226         Process_Mgt_Types.process AD;  
227         -- Process executing "Spelling_checker".  
228
```

PRELIMINARY

```

229 child_termination_event_value:
230     constant Event_Mgt.event_value :=
231         Event_Mgt.user_2;
232     -- Local event signalled when spelling checker
233     -- process terminates.
234
235 child_termination_event: Event_Mgt.action_record;
236     -- Action record used to receive spelling checker
237     -- process's termination event.
238
239 this_process_untyped: System.untyped_word;
240     -- Process executing "Word_processor",
241     -- as an "untyped_word".
242
243 word_event: Event_Mgt.action_record;
244     -- Used to signal each word to be checked.
245 current_word: word_record;
246 FOR current_word
247     USE AT word_event.message.offset'address;
248     -- Overlay used for word location.
249
250 old_event_status: Event_Mgt.event_status;
251     -- Saves previous event status for the
252     -- spelling_error local event, so the previous
253     -- status can be restored before exit.
254
255 begin
256     this_process_untyped :=
257         Process_Mgt.Get_process_globals_entry(
258             Process_Mgt.Types.process);
259
260     spelling_checker_process := Process_Mgt.
261         Spawn_process(
262             init_proc =>
263                 Spelling_checker'subprogram_value,
264             term_action => (
265                 event =>
266                     child_termination_event_value,
267                 message => System.null_address,
268                 -- Not used.
269                 destination => this_process_untyped));
270
271     old_event_status := Event_Mgt.
272         Establish_event_handler(
273             event => spelling_error_event_value,
274             status => (
275                 handler =>
276                     Spelling_error_handler'
277                         subprogram_value,
278                 state => Event_Mgt.enabled,
279                 interrupt_system_call => false));
280
281     loop
282         -- Presume that control exits the loop when a
283         -- user quits the word processor.
284
285         -- Code to do word processing goes here. For
286         -- each word entered by the user,
287         -- the following code is executed:
288
289         word_event.event := word_event_value;
290         word_event.message.AD := this_process_untyped;
291
292         -- Code goes here to assign "current_word" which
293         -- overlays "word_event.message.offset".
294
295         word_event.destination :=
296             Conversion_Support_Ex.Untyped_from_process(
297                 spelling_checker_process);
298         Event_Mgt.Signal(word_event);
299
300     end loop;
301
302 << QUIT >> -- Presume control reaches this point
303             -- when a user exits the word
304             -- processor.
305

```

## PRELIMINARY

```
306     Event_Mgt.Signal(Event_Mgt.action_record'(
307         event      => Event_Mgt.termination,
308         message     => System.null_address,
309         -- No message.
310         destination => Conversion_Support_Ex.
311             Untyped_from_process(
312                 spelling_checker_process));
313     Event_Mgt.Wait_for_any(
314         events => (
315             child_termination_event_value => true,
316             others => false),
317         action => child_termination_event);
318     Process_Mgt.Deallocate(spelling_checker_process);
319
320     old_event_status := Event_Mgt.
321         Establish_event_handler(
322             event => spelling_error_event_value,
323             status => old_event_status);
324     -- Reestablish previous event status.
325     -- Value returned is never used.
326
327     end Word_processor;
328
329
330 end Word_Processor_Ex;
```

## X-A.6.14 View\_device\_main Procedure

```

1  with CL_Defs,
2      Command_Handler,
3      Device_Defs,
4      Environment_Mgt,
5      System,
6      System_Defs,
7      VD_Commands,
8      VD_Devices,
9      VD_Defs;
10
11 procedure View_device_main
12 --
13 -- Function:
14 --   Main program for "view.device" utility
15 --   (Command-Oriented Program Example).
16 --
17 --   The procedure "View_device_main" is
18 --   called from CLEX. "View_device_main"
19 --   performs the top-level processing for the
20 --   "view.device" example utility.
21 --
22 -- History:
23 --   10-08-87, William A. Rohm:  Written.
24 --   11-17-87, WAR:              Revised.
25 --
26 is
27
28 -- Variables:
29 --
30 command: System.short_ordinal;
31 -- Index of current command (in current
32 -- command set).
33
34 command_name: System_Defs.text(CL_Defs.max_name_sz);
35 -- Name of current command (in current
36 -- command set).
37
38 current_cmd_odo: Device_Defs.opened_device :=
39   Command_Handler.Open_invocation_command_processing;
40 -- Current opened command input device,
41 -- initially the invocation command.
42
43 device_name: System_Defs.text(256);
44 -- Pathname of viewed device.
45
46 device_opened: boolean;
47 -- Returned true from
48 -- "VD_Devices.Open_device" if device
49 -- successfully opened.
50
51 processing_runtime: boolean := false;
52 -- True if currently processing runtime
53 -- commands, false if processing startup
54 -- commands.
55
56 use System; -- to import = for
57             -- System.short_ordinal
58
59 begin
60
61   VD_Devices.Open_program_window;
62
63   -- Get ":device" pathname:
64   --
65   Command_Handler.Get_string(
66     cmd_odo => current_cmd_odo,
67     arg_number => 1,
68     arg_value => device_name);
69
70   -- Close invocation command processing:
71   --
72   Command_Handler.Close(current_cmd_odo);
73
74

```

## PRELIMINARY

```
75 -- Open startup command input:
76 --
77 current_cmd_odo :=
78   Command_Handler.Open_startup_command_processing(
79     cmd_set => VD_Defs.main_cmd_set);
80
81
82 -- Main processing loop:
83 --
84 loop
85
86   Command_Handler.Get_command(
87     cmd_odo => current_cmd_odo,
88     prompt => VD_Defs.main_prompt,
89     cmd_id => command,
90     cmd_name => command_name);
91
92   case command is
93   when VD_Defs.main_change_ID =>
94     Command_Handler.Get_string(
95       cmd_odo => current_cmd_odo,
96       arg_number => 1,
97       arg_value => device_name);
98
99     VD_Devices.device_info_valid := false;
100
101   when VD_Defs.main_list_ID =>
102
103     declare
104       ops: boolean;
105       -- Returned ":operations" parameter.
106
107     begin
108       -- Get ":operations" parameter:
109       --
110       ops := Command_Handler.Get_boolean(
111         cmd_odo => current_cmd_odo,
112         arg_number => 1);
113
114
115       -- Display device information:
116       --
117       VD_Commands.Display_device_info(
118         device_name => device_name,
119         operations => ops);
120
121     end;
122
123
124   when VD_Defs.main_access_ID =>
125
126     declare
127       open_mode: System.short_ordinal;
128       -- Enumeration index value of "access.device" method.
129
130     begin
131       -- Get desired open mode:
132       --
133       open_mode :=
134         Command_Handler.Get_enumeration_index(
135           cmd_odo => current_cmd_odo,
136           arg_number => 1);
137
138       -- Open device:
139       --
140       device_opened := VD_Devices.Open_device(
141         device_name => device_name,
142         open_mode => Device_Defs.
143           open_mode'val(open_mode));
144     end;
145
146   if device_opened then
147     -- Change to "access" command set:
148     --
149     Command_Handler.Change_cmd_set(
150       cmd_odo => current_cmd_odo,
151       cmd_set_name => VD_Defs.access_cmd_set);
```

PRELIMINARY

```
152
153     VD_Commands.Process_access_commands(
154         cmd_odo => current_cmd_odo);
155
156     -- Return to "main" command set:
157     --
158     Command_Handler.Change_cmd_set(
159         cmd_odo      => current_cmd_odo,
160         cmd_set_name => VD_Defs.main_cmd_set);
161
162     end if;      -- if device_opened
163
164     when VD_Defs.main_exit_ID =>
165
166         if processing_runtime then
167             EXIT;
168         else
169
170             -- Close invocation command input
171             -- device:
172             --
173             Command_Handler.Close(current_cmd_odo);
174
175             -- Open runtime command processing:
176             --
177             current_cmd_odo :=
178                 Command_Handler.Open_runtime_command_processing(
179                     cmd_set => VD_Defs.main_cmd_set);
180
181             processing_runtime := true;
182
183         end if;
184
185         when others =>
186             null;
187
188     end case;
189
190 end loop;
191
192 if device_opened then
193     VD_Devices.Close_device;
194 end if;
195
196 -- Close runtime command input device:
197 --
198 Command_Handler.Close(current_cmd_odo);
199
200 -- Close program window:
201 --
202 VD_Devices.Close_program_window;
203
204 end View_device_main;
```

## X-A.6.15 VD\_Defs Package Specification

```

1  with System,
2      System_Defs,
3      Terminal_Defs;
4
5  package VD_Defs is
6      --
7      -- Function:
8      --   Contains definitions for the constants in
9      --   the Example Utility.
10     --
11     -- History:
12     --   10-08-87, William A. Rohm:  Written.
13     --   11-16-87, WAR:              Revised.
14     --
15     -- End of Header
16
17
18     -- Constants:
19     --
20     program_window_size:
21         Terminal_Defs.point_info := (80,20);
22         -- Size of program's window, in columns
23         -- and rows.
24
25     program_buffer_size:
26         Terminal_Defs.point_info := (80,20);
27         -- Size of program window's buffer.
28
29     program_window_pos:
30         Terminal_Defs.point_info := (1,1);
31         -- Position of program's window on
32         -- terminal (upper left corner).
33
34     main_cmd_set_str: constant string := "$OEO/main";
35         -- String value of main command set's
36         -- pathname.
37
38     main_cmd_set: System_Defs.text(
39         main_cmd_set_str'length) := (
40         main_cmd_set_str'length,
41         main_cmd_set_str'length,
42         main_cmd_set_str);
43         -- Pathname of main command set.
44
45
46     access_cmd_set_str: constant string := "$OEO/access";
47         -- String value of "device access" command
48         -- set's pathname.
49
50     access_cmd_set: System_Defs.text(
51         access_cmd_set_str'length) := (
52         access_cmd_set_str'length,
53         access_cmd_set_str'length,
54         access_cmd_set_str);
55         -- Pathname of "device access" command set.
56
57
58     main_prompt_str: constant string := "view.device> ";
59         -- String value of prompt for "main" command
60         -- set.
61
62     main_prompt: System_Defs.text(
63         main_prompt_str'length) := (
64         main_prompt_str'length,
65         main_prompt_str'length,
66         main_prompt_str);
67         -- "main" prompt's text.
68
69
70     access_prompt_str: constant string :=
71         "access.device> ";
72         -- String value of prompt for "access"
73         -- command set.
74

```

```

75     access_prompt: System Defs.text (
76         access_prompt_str'length) := (
77             access_prompt_str'length,
78             access_prompt_str'length,
79             access_prompt_str);
80     -- "access" prompt's text.
81
82
83     -- Command and Argument Indexes:
84     --
85     main_change_ID: constant System.short_ordinal := 1;
86     main_list_ID:   constant System.short_ordinal := 2;
87     main_access_ID: constant System.short_ordinal := 3;
88     main_exit_ID:  constant System.short_ordinal := 4;
89     -- *Main* command set command index values.
90
91     input_index:   constant
92         System.short_ordinal := 1;
93     output_index:  constant
94         System.short_ordinal := 2;
95     input_partial_index: constant
96         System.short_ordinal := 3;
97     input_output_index: constant
98         System.short_ordinal := 4;
99     -- For "access.device :open_mode"; the
100    -- argument's enumeration index values.
101
102
103     access_read_ID: constant System.short_ordinal := 1;
104     access_write_ID: constant System.short_ordinal := 2;
105     access_exit_ID: constant System.short_ordinal := 3;
106     -- *access* command set's
107     -- command index values.
108
109     read_length_arg: constant
110         System.short_ordinal := 1;
111     read_position_arg: constant
112         System.short_ordinal := 2;
113     read_offset_arg: constant
114         System.short_ordinal := 3;
115     -- Argument index values for "read".
116
117     write_position_arg: constant
118         System.short_ordinal := 1;
119     write_offset_arg: constant
120         System.short_ordinal := 2;
121     -- Argument index values for "write".
122
123 end VD_Defs;
124

```

## X-A.6.16 VD\_Commands Package Specification

```
1  with Device_Defs,
2      System_Defs;
3
4  package VD_Commands is
5      --
6      -- Function:
7      -- Contains operations related to processing
8      -- "view.device" "access" command set's
9      -- commands.
10     --
11     -- History:
12     -- 10-08-87, William A. Rohm: Written.
13     -- 11-17-87, WAR: Revised.
14     --
15     -- End of Header
16
17
18     procedure Display_device_info(
19         device_name: System_Defs.text;
20         -- Pathname of device.
21         operations: boolean);
22         -- If true, displays "Byte_Stream_AM.Ops"
23         -- operations supported by "device_name".
24     --
25     -- Function:
26     -- Calls "VD_Devices.Get_device_info",
27     -- then displays the returned device
28     -- information record.
29
30
31     procedure Process_access_commands(
32         cmd_odo: Device_Defs.opened_device);
33         -- Opened command input device.
34     --
35     -- Function:
36     -- Processes the "access" command set.
37
38
39 end VD_Commands;
```

## X-A.6.17 VD\_Commands Package Body

```

1  with Byte_Stream_AM,
2      CL_Defs,
3      Command_Handler,
4      Device_Defs,
5      System,
6      System_Defs,
7      Text_Mgt,
8      VD_Defs,
9      VD_Devices;
10
11 package body VD_Commands is
12     --
13     -- Function:
14     --   Contains operations related to processing
15     --   "view.device" "access" command set.
16     --
17     -- History:
18     --   10-08-87, William A. Rohm:   Written.
19     --   11-17-87, WAR:              Revised.
20     --
21     -- End of Header
22
23
24     procedure Display_device_info(
25         device_name: System_Defs.text;
26         operations: boolean)
27     --
28     -- Logic:
29     --   1. Check for valid device info record; get it
30     --   if not valid
31     --   2. Display common device info values
32     --   3. Display BSAM device info values
33     --   4. If "operations" is true, display supported
34     --   ops
35
36     is
37
38         procedure Write_info(
39             info_string: string)
40             -- String value to be written.
41         is
42             --
43             -- Function:
44             --   Display string value, followed by a linefeed.
45
46             info_text: System_Defs.text(32);
47             -- Text value of various values' "image"s.
48         begin
49
50             -- Make a text value of "info_string":
51             --
52             Text_Mgt.Set(
53                 dest => info_text,
54                 source => info_string);
55
56             -- Add a linefeed:
57             --
58             Text_Mgt.Append(
59                 dest => info_text,
60                 source => Standard.ASCII.LF);
61
62             -- Write text to the program's window:
63             --
64             Byte_Stream_AM.Ops.Write(
65                 opened_dev => VD_Devices.program_window,
66                 buffer_VA => info_text'address,
67                 length => System.ordinal(info_text.length));
68
69         end Write_info;
70
71     begin
72
73         -- Check for valid "device_info":

```

PRELIMINARY

```

75  --
76  if not VD_Devices.device_info_valid then
77    VD_Devices.Get_device_info(
78      device_name => device_name);
79  end if;
80
81
82  -- Display node id:
83  --
84  Write_info(
85    info_string => "  Node ID:");
86
87  Write_info(
88    info_string => System_Defs.node_ID'image(
89      VD_Devices.device_info.common_info.node));
90
91
92  -- Display access methods supported:
93  --
94  Write_info(
95    info_string => "  Access Methods Supported:");
96
97  for i in Device_Defs.access_method'first ..
98    Device_Defs.access_method'last loop
99
100    if VD_Devices.device_info.
101      common_info.acc_methods_supp(i) then
102
103      Write_info(
104        info_string => Device_Defs.
105          access_method'image(i));
106    end if;
107
108  end loop;
109
110
111  -- Display open modes supported:
112  --
113  Write_info(
114    info_string => "  Supported Open Modes:");
115
116  for i in Device_Defs.open_mode'first ..
117    Device_Defs.open_mode'last loop
118
119    if VD_Devices.device_info.
120      common_info.open_modes_supp(i) then
121
122      Write_info(
123        info_string => Device_Defs.
124          open_mode'image(i));
125    end if;
126
127  end loop;
128
129
130  -- Display "store supported" boolean:
131  --
132  Write_info(
133    info_string => "  Data written to device can be read back:");
134
135  Write_info(
136    info_string => boolean'image(
137      VD_Devices.device_info.
138        common_info.store_supp));
139
140
141  -- Display "is interactive" boolean:
142  --
143  Write_info(
144    info_string => "  Device is interactive is:");
145
146  Write_info(
147    info_string => boolean'image(
148      VD_Devices.device_info.
149        common_info.is_interactive));
150
151

```

```

152     -- Display byte-stream operations supported;
153     --
154     if operations then
155         Write_info(
156             info_string => "    Supported Byte Stream Operations:");
157
158         for i in Byte_Stream_AM.bsam_operation'first ..
159             Byte_Stream_AM.bsam_operation'last loop
160
161             if VD_Devices.device_info.bsam_ops_supp(i) then
162
163                 Write_info(
164                     info_string => Byte_Stream_AM.
165                         bsam_operation'image(i));
166                 end if;
167             end loop;
168         end if;
169     end if;
170
171 end Display_device_info;
172
173
174
175 procedure Process_access_commands(
176     cmd_odo: Device_Defs.opened_device)
177
178 is
179     command: System.short_ordinal;
180     -- Index of current command (in current
181     -- command set).
182
183     command_name: System_Defs.text(CL_Defs.max_name_sz);
184     -- Name of current command (in current
185     -- command set).
186
187     length: CL_Defs.CL_range;
188     -- Length of displayed bytes for "read :length".
189
190     position: System.short_ordinal;
191     -- Index of "read/write :position" argument's value.
192
193     offset: integer;
194     -- Value of "read/write :offset" argument.
195
196 begin
197
198     -- Command processing loop:
199     --
200     loop
201
202         Command_Handler.Get_command(
203             cmd_odo => cmd_odo,
204             prompt => VD_Defs.access_prompt,
205             cmd_id => command,
206             cmd_name => command_name);
207
208         case command is
209
210             when VD_Defs.access_read_ID =>
211
212                 -- Get ":length" argument:
213                 --
214                 length := Command_Handler.Get_range(
215                     cmd_odo => cmd_odo,
216                     arg_number => VD_Defs.read_length_arg);
217
218                 -- Get ":position" argument:
219                 --
220                 position := Command_Handler.
221                     Get_enumeration_index(
222                         cmd_odo => cmd_odo,
223                         arg_number => VD_Defs.read_position_arg);
224
225                 -- Get ":offset" argument:
226                 --
227                 offset := Command_Handler.Get_integer(

```

```
229         cmd_odo => cmd_odo,
230         arg_number => VD_Defs.read_offset_arg);
231
232     -- Read and display bytes:
233     --
234     -- TBD
235
236     when VD_Defs.access_write_ID =>
237
238         -- Get ":position" argument:
239         --
240         position := Command_Handler.
241             Get_enumeration_index(
242                 cmd_odo => cmd_odo,
243                 arg_number => VD_Defs.write_position_arg);
244
245         -- Get ":offset" argument:
246         --
247         offset := Command_Handler.Get_integer(
248             cmd_odo => cmd_odo,
249             arg_number => VD_Defs.write_offset_arg);
250
251         -- Get bytes and write to device:
252         --
253         -- TBD
254
255     when VD_Defs.access_exit_ID =>
256         EXIT;
257
258     when others =>
259         null;
260
261     end case;
262
263 end loop;
264
265 end Process_access_commands;
266
267 end VD_Commands;
```

## X-A.6.18 VD\_Devices Package Specification

```
1  with Byte_Stream_AM,
2      Device_Defs,
3      Long_Integer_Defs,
4      System,
5      System_Defs;
6
7  package VD_Devices is
8      --
9      -- Function:
10     -- Contains all operations related to the
11     -- viewed device and the windows.
12     --
13     -- This package contains calls to open and
14     -- close the program's windows, and calls to
15     -- read and write bytes to and from the
16     -- viewed device.
17     --
18     -- History:
19     -- 10-08-87, William A. Rohm:  Written.
20     -- 11-17-87, WAR:             Revised.
21     --
22     -- End of Header
23
24
25     -- Variables:
26     --
27     program_window: Device_Defs.opened_device;
28     -- Utility's window, for accepting commands
29     -- and displaying data.
30
31
32     opened_device: Device_Defs.opened_device :=
33         System.null_word;
34     -- Opened viewed device.
35
36     device_info: Byte_Stream_AM.device_info;
37     -- Device information record for
38     -- "Byte_Stream_AM".
39
40     device_info_valid: boolean := false;
41     -- Whether the device information record is valid.
42
43
44     procedure Open_program_window;
45     --
46     -- Function:
47     -- Open the program's window on the
48     -- current terminal.
49
50
51     procedure Close_program_window;
52     --
53     -- Function:
54     -- Closes the program's main window, and
55     -- any opened "::window" windows.
56
57
58     procedure Get_device_info(
59         device_name: System_Defs.text);
60     --
61     -- Function:
62     -- Calls "Byte_Stream_AM.Get_device_info" to set
63     -- "VD_Devices.device_info" information record.
64
65
66     function Open_device(
67         device_name: System_Defs.text;
68         -- Pathname of device to be opened.
69         open_mode: Device_Defs.open_mode)
70         -- Open mode for device.
71         return boolean;
72         -- True if device successfully opened.
73     --
74     -- Function:
```

```

75  -- Opens given device with
76  -- "Byte_Stream_AM.Open_by_name",
77  -- returning true if successful.
78  --
79  -- Sets this package's "opened device"
80  -- variable; "System.null_word" if
81  -- inaccessible.
82
83
84  procedure Read_bytes(
85      length:      System.ordinal;
86      -- Number of bytes to be read and
87      -- displayed.
88      position:    Byte_Stream_AM.position_mode;
89      -- Position from which "offset" is measured.
90      offset:      integer;
91      -- Offset of first byte to be read and
92      -- displayed.
93      bytes:       out System_Defs.text);
94      -- Bytes read from device.
95  --
96  -- Function:
97  -- Reads and displays bytes from the opened
98  -- device.
99
100
101  procedure Write_bytes(
102      position:    Byte_Stream_AM.position_mode;
103      -- Position from which "offset" is measured.
104      offset:      System.ordinal;
105      -- Offset of first byte to be written to
106      -- device.
107      bytes:       System_Defs.text);
108      -- Bytes to be written to device.
109  --
110  -- Function:
111  -- Reads and displays bytes from the opened
112  -- device.
113
114
115  procedure Close_device;
116  --
117  -- Function:
118  -- Closes opened device with
119  -- "Byte_Stream_AM.Close".
120
121  end VD_Devices;

```

## X-A.6.19 VD\_Devices Package Body

```

1  with Access_Mgt,
2      Byte_Stream_AM,
3      Device_Defs,
4      Directory_Mgt,
5  --   Example_Messages,
6      Object_Mgt,
7      Process_Mgt,
8      Process_Mgt_Types,
9      System,
10     System_Defs,
11     System_Exceptions,
12     Terminal_Defs,
13     Unchecked_Conversion,
14     VD_Defs,
15     Window_Services;
16
17
18 package body VD_Devices is
19     --
20     -- History:
21     --   10-08-87, William A. Rohm:   Written.
22     --   11-17-87, WAR:              Revised.
23     --
24     -- End of Header
25
26
27
28 procedure Open_program_window
29     --
30     -- Logic:
31     --   1. Gets device AD to underlying terminal.
32     --   2. Opens and assigns "program_window".
33 is
34     old_opened_window: Device_Defs.opened_device;
35     old_window:        Device_Defs.device;
36     underlying_terminal: Device_Defs.device;
37
38 begin
39
40     -- Assume standard input, on entry, is from
41     -- an opened window:
42     --
43     old_opened_window :=
44         Process_Mgt.Get_process_globals_entry(
45             Process_Mgt_Types.standard_input);
46
47
48     -- Get device object of standard input
49     -- window:
50     --
51     old_window :=
52         Byte_Stream_AM.Ops.Get_device_object(
53             old_opened_window);
54
55
56     -- Get device AD of standard input window's
57     -- terminal:
58     --
59     underlying_terminal :=
60         Window_Services.Ops.Get_terminal(
61             old_window);
62
63
64     -- Create program window:
65     --
66     program_window := Window_Services.Ops.Create_window(
67         terminal           => underlying_terminal,
68         pixel_units       => false, -- characters, not pixels
69         fb_size           => VD_Defs.program_buffer_size,
70         desired_window_size => VD_Defs.program_window_size,
71         window_pos        => VD_Defs.program_window_pos,
72         view_pos           => Terminal_Defs.point_info'(1,1));
73
74 end Open_program_window;

```

PRELIMINARY

```

75
76
77 procedure Close_program_window
78     --
79     -- Logic:
80     -- 1. Close the program window.
81 is
82 begin
83
84     Window_Services.Ops.Destroy_window(program_window);
85
86 end Close_program_window;
87
88
89 procedure Get_device_info(
90     device_name: System_Defs.text)
91
92 is
93     device: Device_Defs.device;
94     -- Device.
95
96     device_untyped: System.untyped_word;
97     FOR device_untyped USE AT device'address;
98     -- Device as an untyped word.
99
100 begin
101
102     begin
103
104         device_untyped := Directory_Mgt.Retrieve(
105             name => device_name);
106
107         device_info :=
108             Byte_Stream_AM.Ops.Get_device_info(
109                 dev => device);
110
111         device_info_valid := true;
112
113     exception
114         when Directory_Mgt.no_access =>
115             RAISE; -- msg no_access
116
117         when others => RAISE;
118
119     end;
120
121 end Get_device_info;
122
123
124 function Open_device(
125     device_name: System_Defs.text;
126     open_mode: Device_Defs.open_mode)
127     return boolean
128     --
129     -- Logic:
130     -- 1. Check for allowed open mode
131     -- 2. Attempt "BSAM_AM.Open_by_name"
132     -- 3. If successful, assign
133     --     "opened_device", return true;
134     --     otherwise, assign "opened_device"
135     --     null, return false
136 is
137     successful: boolean := false;
138     -- Returned true if successfully opens
139     -- device.
140 begin
141
142     if not device_info_valid then
143         Get_device_info(device_name);
144     end if;
145
146     if device_info_valid and
147         device_info.common_info.
148         open_modes_supp(open_mode) then
149
150         -- Try to open device:
151         --

```

```

152     begin
153         opened_device := Byte_Stream_AM.Open_by_name(
154             name      => device_name,
155             input_output => open_mode,
156             allow      => Device_Defs.anything,
157             block      => true);
158
159         successful := true;
160
161     exception
162     when others =>
163         opened_device := System.null_word;
164     end;
165
166 end if;      -- if valid and open_mode
167             -- supported
168
169     return successful;
170
171 end Open_device;
172
173
174
175 procedure Read_bytes(
176     length:      System.ordinal;
177     position:    Byte_Stream_AM.position_mode;
178     offset:      integer;
179     bytes:      out System_Defs.text)
180 is
181     byte_position: Long_Integer_Defs.long_integer;
182     -- Byte pointer position, returned from
183     -- "Byte_Stream_AM.Ops.Set_position".
184
185     bytes_read: System.ordinal;
186     -- Number of bytes actually read.
187
188 use System;  -- to import "/" for System.ordinal
189
190 begin
191
192     byte_position := Byte_Stream_AM.Ops.Set_position(
193         opened_dev => opened_device,
194         pos        => Long_Integer_Defs.
195             Convert_to_long_integer(
196                 number => offset),
197         mode       => position);
198
199     bytes_read := Byte_Stream_AM.Ops.Read(
200         opened_dev => opened_device,
201         buffer_VA  => bytes'address,
202         length     => System.ordinal(offset),
203         block      => false);
204
205     if integer(bytes_read) = offset then
206
207         bytes_read := Byte_Stream_AM.Ops.Read(
208             opened_dev => opened_device,
209             buffer_VA  => bytes'address,
210             length     => bytes'size/8,
211             block      => false);
212
213         if bytes_read /= length then
214             bytes.length := System_Defs.text_length(bytes_read);
215         end if;
216
217     end if;
218
219 end Read_bytes;
220
221
222 procedure Write_bytes(
223     position:    Byte_Stream_AM.position_mode;
224     offset:      System.ordinal;
225     bytes:      System_Defs.text)
226 is
227     bytes_read: System.ordinal;
228     -- Number of bytes actually read.

```

```
229
230 use System; -- import "=" for System.ordinal;
231
232 begin
233
234     bytes_read := Byte_Stream_AM.Ops.Read(
235         opened_dev => opened_device,
236         buffer_VA  => bytes'address,
237         length     => offset,
238         block      => false);
239
240     if bytes_read = offset then
241
242         bytes_read := Byte_Stream_AM.Ops.Read(
243             opened_dev => opened_device,
244             buffer_VA  => bytes'address,
245             length     => bytes'size/8,
246             block      => false);
247
248     end if;
249
250 end Write_bytes;
251
252
253 procedure Close_device
254 is
255 begin
256     Byte_Stream_AM.Ops.Close(opened_device);
257 end Close_device;
258
259 end VD_Devices;
```

## X-A.7 Type Manager Services

## X-A.7.1 Acct\_main\_ex Procedure

Main procedure of the account manager test driver.

```
1 -----
2 -----
3 --
4 --          COMMAND DEFINITIONS          --
5 --
6 -----
7 -----
8
9
10 ---D* manage.commands
11 ---D*
12 ---D*
13 ---D*
14 ---D*   create.invocation_command
15 ---D*   end
16 ---D*
17 ---D*
18 ---D*
19 ---D*   create.runtime_command_set :cmd_def = acct_cmds \
20 ---D*                               :prompt = "ACCT_MGT> "
21 ---D*
22 ---D*   define.command :cmd_name = create
23 ---D*   set.description :text = "
24 ---D*       -- Create a new account with an initial balance.
25 ---D*       "
26 ---D*
27 ---D*   define.argument :arg_name = init_balance \
28 ---D*                   :type      = integer
29 ---D*   set.description :text = "
30 ---D*       -- Initial balance of an account.
31 ---D*       -- Must be between 0 and 100000.
32 ---D*       "
33 ---D*   set.bounds      :value = 0..100000
34 ---D*   set.mandatory
35 ---D*   end
36 ---D*   end
37 ---D*
38 ---D*
39 ---D*   define.command :cmd_name = cstore
40 ---D*   set.description :text = "
41 ---D*       -- Create and store a new account in one step.
42 ---D*       "
43 ---D*
44 ---D*   define.argument :arg_name = pathname \
45 ---D*                   :type      = string
46 ---D*   set.description :text = "
47 ---D*       -- Pathname to store the account. Must be
48 ---D*       -- a valid pathname that is not already in use.
49 ---D*       -- Caller must have store rights in the referenced
50 ---D*       -- directory.
51 ---D*       "
52 ---D*   set.maximum_length 43
53 ---D*   set.mandatory
54 ---D*   end
55 ---D*
56 ---D*   define.argument :arg_name = init_balance \
57 ---D*                   :type      = integer
58 ---D*   set.description :text = "
59 ---D*       -- Initial balance of the account. Must be
60 ---D*       -- greater or equal to zero and less than or equal
61 ---D*       -- to 100000.
62 ---D*       "
63 ---D*   set.bounds      :value = 0..100000
64 ---D*   set.mandatory
65 ---D*   end
66 ---D*
67 ---D*   define.argument :arg_name = authority \
68 ---D*                   :type      = string
69 ---D*   set.description :text = "
70 ---D*       -- Specifies an authority list to be stored
71 ---D*       -- with an account. Has to be created separately
72 ---D*       -- invoking the manage.authority runtime command.
73 ---D*       -- Default value is none.
```

## PRELIMINARY

```

74 ---D*      "
75 ---D*      set.maximum_length 43
76 ---D*      set.value_default :value = "none"
77 ---D*      end
78 ---D*      end
79 ---D*
80 ---D*
81 ---D*      define.command :cmd_name = store
82 ---D*      set.description :text = "
83 ---D*          -- Store an existing active account.
84 ---D*          -- Causes separate command set acct_cmd_store
85 ---D*          -- to be invoked.
86 ---D*      "
87 ---D*
88 ---D*      define.argument :arg_name = ref_number \
89 ---D*          :type      = integer
90 ---D*      set.description :text = "
91 ---D*          -- Reference to an account. Has to be
92 ---D*          -- between 1 and 100.
93 ---D*      "
94 ---D*      set.bounds :value = 1..100
95 ---D*      set.mandatory
96 ---D*      end
97 ---D*
98 ---D*      define.argument :arg_name = pathname \
99 ---D*          :type      = string
100 ---D*      set.description :text = "
101 ---D*          -- Pathname to store the account. Must be
102 ---D*          -- a valid pathname that is not already in use.
103 ---D*          -- Caller must have store rights in the referenced
104 ---D*          -- directory.
105 ---D*      "
106 ---D*      set.maximum_length 43
107 ---D*      set.mandatory
108 ---D*      end
109 ---D*
110 ---D*      define.argument :arg_name = authority \
111 ---D*          :type      = string
112 ---D*      set.description :text = "
113 ---D*          -- Specifies an authority list to be stored
114 ---D*          -- with an account. Has to be created separately
115 ---D*          -- invoking the manage.authority runtime command.
116 ---D*          -- Default value is none.
117 ---D*      "
118 ---D*      set.maximum_length 43
119 ---D*      set.value_default :value = "none"
120 ---D*      end
121 ---D*      end
122 ---D*
123 ---D*
124 ---D*      define.command :cmd_name = retrieve
125 ---D*      set.description :text = "
126 ---D*          -- Retrieve a stored account from a pathname
127 ---D*          -- and make it available for online processing.
128 ---D*      "
129 ---D*
130 ---D*      define.argument :arg_name = pathname \
131 ---D*          :type      = string
132 ---D*      set.description :text = "
133 ---D*          -- Pathname of a account to be retrieved. Can
134 ---D*          -- be relative, absolute, or network pathname.
135 ---D*          -- Must be a valid pathname and pathname must
136 ---D*          -- reference an account.
137 ---D*      "
138 ---D*      set.maximum_length :value = 43
139 ---D*      set.mandatory
140 ---D*      end
141 ---D*      end
142 ---D*
143 ---D*
144 ---D*      define.command :cmd_name = "list"
145 ---D*      set.description :text = "
146 ---D*          -- List all accounts currently available for
147 ---D*          -- online processing by ordinal reference number.
148 ---D*      "
149 ---D*      end
150 ---D*

```

PRELIMINARY

```

151 ---D*
152 ---D*   define.command :cmd_name = display
153 ---D*   set.description :text = "
154 ---D*     -- Display all relevant information about an account.
155 ---D*     "
156 ---D*
157 ---D*   define.argument :arg_name = ref_number \
158 ---D*     :type           = integer
159 ---D*   set.description :text = "
160 ---D*     -- Ordinal number referencing the account
161 ---D*     "
162 ---D*   set.bounds      :value = 0..100
163 ---D*   set.value_default :value = 0
164 ---D* end
165 ---D* end
166 ---D*
167 ---D*
168 ---D*   define.command :cmd_name = withdraw
169 ---D*   set.description :text = "
170 ---D*     -- Withdraw a given amount from an account
171 ---D*     "
172 ---D*
173 ---D*   define.argument :arg_name = ref_number \
174 ---D*     :type           = integer
175 ---D*   set.bounds      :value = 1..100
176 ---D*   set.mandatory
177 ---D* end
178 ---D*
179 ---D*   define.argument :arg_name = amount \
180 ---D*     :type           = integer
181 ---D*   set.bounds      :value = 0..100000
182 ---D*   set.mandatory
183 ---D* end
184 ---D* end
185 ---D*
186 ---D*
187 ---D*   define.command :cmd_name = deposit
188 ---D*   set.description :text = "
189 ---D*     -- Deposit a given amount to an account
190 ---D*     "
191 ---D*
192 ---D*   define.argument :arg_name = ref_number \
193 ---D*     :type           = integer
194 ---D*   set.bounds      :value = 1..100
195 ---D*   set.mandatory
196 ---D* end
197 ---D*
198 ---D*   define.argument :arg_name = amount \
199 ---D*     :type           = integer
200 ---D*   set.bounds      :value = 0..100000
201 ---D*   set.mandatory
202 ---D* end
203 ---D* end
204 ---D*
205 ---D*
206 ---D*   define.command :cmd_name = transfer
207 ---D*   set.description :text = "
208 ---D*     -- Transfer amount from source to destination.
209 ---D*     "
210 ---D*
211 ---D*   define.argument :arg_name = source \
212 ---D*     :type           = integer
213 ---D*   set.bounds      :value = 1..100
214 ---D*   set.mandatory
215 ---D* end
216 ---D*
217 ---D*   define.argument :arg_name = destination \
218 ---D*     :type           = integer
219 ---D*   set.bounds      :value = 1..100
220 ---D*   set.mandatory
221 ---D* end
222 ---D*
223 ---D*   define.argument :arg_name = amount \
224 ---D*     :type           = integer
225 ---D*   set.bounds      :value = 0..100000
226 ---D*   set.mandatory
227 ---D* end

```

PRELIMINARY

```

228 --*D*      end
229 --*D*
230 --*D*
231 --*D*      define.command :cmd_name = remove
232 --*D*          set.description :text = "
233 --*D*              -- Remove an account from online processing
234 --*D*              -- Does not affect an accounts passive version.
235 --*D*              "
236 --*D*
237 --*D*          define.argument :arg_name = ref_number \
238 --*D*              :type      = integer
239 --*D*          set.bounds      :value = 1..100
240 --*D*          set.mandatory
241 --*D*      end
242 --*D*      end
243 --*D*
244 --*D*
245 --*D*      define.command :cmd_name = destroy
246 --*D*          set.description :text = "
247 --*D*              -- Destroy an account's passive version.
248 --*D*              -- Does not affect an account's online representation.
249 --*D*              -- Fails for account's that have not been passivated.
250 --*D*              "
251 --*D*
252 --*D*          define.argument :arg_name = ref_number \
253 --*D*              :type      = integer
254 --*D*          set.bounds      :value = 1..100
255 --*D*          set.mandatory
256 --*D*      end
257 --*D*      end
258 --*D*
259 --*D*
260 --*D*      define.command :cmd_name = manage.authority
261 --*D*          set.description :text = "
262 --*D*              -- Invokes the manage.authority utility to
263 --*D*              -- create authority list from within this
264 --*D*              -- program.
265 --*D*              "
266 --*D*      end
267 --*D*
268 --*D*
269 --*D*      define.command :cmd_name = save
270 --*D*          --
271 --*D*          -- Invokes the screensaver utility.
272 --*D*          --
273 --*D*
274 --*D*          define.argument :arg_name = "args" \
275 --*D*              :type      = string
276 --*D*          set.value_default :value = ""
277 --*D*          set.description :text = "
278 --*D*              -- Arguments to pass on to screensaver
279 --*D*              -- Type csh command line in quotes.
280 --*D*              "
281 --*D*      end
282 --*D*      end
283 --*D*
284 --*D*
285 --*D*      define.command :cmd_name = "exit"
286 --*D*          set.description :text = "
287 --*D*              -- Exit accounting program
288 --*D*              "
289 --*D*      end
290 --*D*      end
291 --*D*      exit
292
293 -----
294 -----
295 --
296 --          MESSAGE DEFINITIONS          --
297 --
298 -----
299 -----
300
301 --*D*
302 --*D*      manage.messages
303 --*D*
304 --*D*

```

```

305 ---D*
306 ---D*   set.language :language = english
307 ---D*
308 ---D*
309 ---D*   store :module   = 1 \
310 ---D*       :number   = 1 \
311 ---D*       :msg_name  = welcome \
312 ---D*       :short    = "Welcome to the Account Manager"
313 ---D*
314 ---D*
315 ---D*   store :module   = 1 \
316 ---D*       :number   = 2 \
317 ---D*       :msg_name  = local_created \
318 ---D*       :short    = "Local account number $p1<ref_number> has \
319 ---D* initial balance $p2<initial_balance>."
320 ---D*
321 ---D*   store :module   = 1 \
322 ---D*       :number   = 3 \
323 ---D*       :msg_name  = list_limits_exceeded \
324 ---D*       :short    = \
325 ---D*       "You can no longer create accounts.
326 ---D*       Your limit of $p1<list_length_limit> has been exceeded."
327 ---D*
328 ---D*
329 ---D*   store :module   = 1 \
330 ---D*       :number   = 4 \
331 ---D*       :msg_name  = unrecognized_problem \
332 ---D*       :short    = "An unrecognized exception has been found."
333 ---D*
334 ---D*
335 ---D*   store :module   = 1 \
336 ---D*       :number   = 5 \
337 ---D*       :msg_name  = no_access \
338 ---D*       :short    = "You specified an invalid pathname."
339 ---D*
340 ---D*
341 ---D*   store :module   = 1 \
342 ---D*       :number   = 6 \
343 ---D*       :msg_name  = invalid_account \
344 ---D*       :short    = "You have specified an invalid account."
345 ---D*
346 ---D*
347 ---D*   store :module   = 1 \
348 ---D*       :number   = 7 \
349 ---D*       :msg_name  = directory_entry_exists \
350 ---D*       :short    = "You have specified an existing directory entry"
351 ---D*
352 ---D*
353 ---D*   store :module   = 1 \
354 ---D*       :number   = 8 \
355 ---D*       :msg_name  = no_default_authority \
356 ---D*       :short    = "There is no default authority list."
357 ---D*
358 ---D*
359 ---D*   store :module   = 1 \
360 ---D*       :number   = 9 \
361 ---D*       :msg_name  = not_implemented \
362 ---D*       :short    = "Operation not currently implemented."
363 ---D*
364 ---D*
365 ---D*   store :module   = 1 \
366 ---D*       :number   = 10 \
367 ---D*       :msg_name  = not_supported \
368 ---D*       :short    = "Operation not supported."
369 ---D*
370 ---D*
371 ---D*   store :module   = 1 \
372 ---D*       :number   = 11 \
373 ---D*       :msg_name  = new_balance \
374 ---D*       :short    = "The new balance in the account
375 ---D* is $p1<new_balance>"
376 ---D*
377 ---D*
378 ---D*   store :module   = 1 \
379 ---D*       :number   = 12 \
380 ---D*       :msg_name  = acct_removed \
381 ---D*       :short    = \

```

```

382 ---D*      "Account with local number $pl<ref_number> has been removed."
383 ---D*
384 ---D*
385 ---D*      store :module = 1 \
386 ---D*           :number = 13 \
387 ---D*           :msg_name = acct_destroyed \
388 ---D*           :short = \
389 ---D*      "Account with pathname $pl<pathname> has been destroyed."
390 ---D*
391 ---D*
392 ---D*      store :module = 1 \
393 ---D*           :number = 14 \
394 ---D*           :msg_name = not_account \
395 ---D*           :short = \
396 ---D*      "$pl<pathname> is not an account."
397 ---D*
398 ---D*
399 ---D*      store :module = 1 \
400 ---D*           :number = 15 \
401 ---D*           :msg_name = not_type_rights \
402 ---D*           :short = \
403 ---D*      "You have insufficient rights for this account."
404 ---D*
405 ---D*
406 ---D*      store :module = 1 \
407 ---D*           :number = 16 \
408 ---D*           :msg_name = no_master_AD \
409 ---D*           :short = "This operation requires that the account \
410 ---D*                   be stored."
411 ---D*
412 ---D*
413 ---D* exit
414
415 with
416   Acct_Mgt_Ex,
417   Acct_visual,
418   Acct_Types,
419   Authority_List_Mgt,
420   Character_Display_AM,
421   Command_Execution,
422   Command_Handler,
423   Conversion_Support_Ex,
424   Device_Defs,
425   Directory_Mgt,
426   Incident_Defs,
427   Long_Integer_Defs,
428   Message_Services,
429   Passive_Store_Mgt,
430   Process_Mgt,
431   Process_Mgt_Types,
432   System,
433   System_Defs,
434   System_Exceptions,
435   Terminal_Defs,
436   Text_Mgt,
437   Transaction_Mgt,
438   Unchecked_conversion,
439   Window_Services;
440
441
442 procedure Acct_main_loop is
443   --
444   -- Function:
445   --   Main event loop for account managing program.
446   --
447
448   -- Variables for creating and storing accounts:
449   --
450   local_list:      Acct_Types.list;
451   -- List of local accounts.
452   list_pointer:    Acct_Types.acct_enum := Acct_Types.list_pointer_init;
453   -- Pointer to first free element in "local_list".
454   ref_number:      Acct_Types.acct_enum;
455   -- Pointer to current element in "local_list".
456   source_number:   Acct_Types.acct_enum;
457   dest_number:     Acct_Types.acct_enum;
458   list_exceeded:   boolean := false;

```

```

459     -- True if "list" is full.
460     pathname:      System_Defs.text(Acct_Types.name_length_limit);
461     -- Container for pathnames.
462     initial_balance: integer;
463     -- Container for initial balances.
464     long_initial_balance: Long_Integer_Defs.long_integer;
465     -- Container for long integers.
466     amount:        Long_Integer_Defs.long_integer;
467     -- Container for long integers.
468     new_balance:   Long_Integer_Defs.long_integer;
469     -- Container for long integers.
470
471
472     -- Variables for Command processing:
473     --
474     input:         Device_Defs.opened_device;
475     -- Opened device for top level command processing.
476     cmd_id:        System.short_ordinal;
477     -- Ordinal identifier for commands.
478     cmd_name:      System_Defs.text(Acct_Types.name_length_limit);
479     -- Textual identifier for commands.
480
481     -- Variables for Window output:
482     --
483     old_opened_window: Device_Defs.opened_device;
484     -- Standard input.
485     old_window:       Device_Defs.device;
486     -- Standard input .. underlying device.
487     new_opened_window: Device_Defs.opened_device;
488     -- Window for display output.
489     new_window:       Device_Defs.device;
490     -- Window for display output -- underlying device.
491     underlying_terminal: Device_Defs.device;
492     -- User terminal.
493     curr_pos:         Terminal_Defs.point_info;
494     -- Current position in the opened window.
495     new_window_info:  Window_Services.window_style_info;
496     -- Style info for new window.
497
498     -- Constants defining Window output:
499     --
500     frame_buffer: constant Terminal_Defs.point_info :=
501     Terminal_Defs.point_info'(80, 20);
502     window_size:  constant Terminal_Defs.point_info :=
503     Terminal_Defs.point_info'(80, 10);
504     window_pos:   constant Terminal_Defs.point_info :=
505     Terminal_Defs.point_info'(1, 1);
506     view_pos:     constant Terminal_Defs.point_info :=
507     Terminal_Defs.point_info'(1, 1);
508     title_string: constant string := "ACCOUNTS";
509
510
511     -- Variables for authority lists:
512     --
513     auth_list:      Authority_list_Mgt.authority_list_AD;
514     -- Authority list for storing accounts.
515     authority_name: System_Defs.text(Acct_Types.name_length_limit);
516     -- Pathname of authority list.
517
518     -- Auxiliary variables:
519     --
520     i:              integer;
521     exit_status:    Incident_Defs.severity_value;
522     aux_text:       System_Defs.text(Window_Services.max_title);
523     untyped_AD:     System.untyped_word;
524     args:           System_Defs.text(Acct_Types.name_length_limit);
525     cmd_line:       System_Defs.text(Acct_Types.name_length_limit);
526
527     -- Exceptions:
528     --
529     list_exceeded_exc: exception;
530     mission_accomplished: exception;
531     invalid_account:    exception;
532     not_implemented:    exception;
533     new_balance_exc:    exception;
534     account_removed:    exception;
535     account_destroyed:  exception;

```

PRELIMINARY

```

536     not_account:           exception;
537
538     -- Conversions:
539     --
540     function Untyped_from_account is new
541         Unchecked_conversion(
542             source => Account_Mgt_Ex.account_AD,
543             target => System.untyped_word);
544
545     function Account_from_untyped is new
546         Unchecked_conversion(
547             source => System.untyped_word,
548             target => Account_Mgt_Ex.account_AD);
549
550
551     use Incident_Defs;           -- Import some frequently used defs.
552     use Long_Integer_Defs;      -- Import long integer arithmetic.
553
554
555     begin
556         -- Initialize account list
557         --
558         for i in Acct_Types.list_pointer_init .. Acct_Types.list_length_limit
559             loop
560             Text_Mgt.Set(
561                 dest => local_list(i).name,
562                 source => Acct_Types.empty_text);
563
564             end loop;
565
566         -- Open runtime command processing:
567         --
568         input := Command_Handler.Open_runtime_command_processing(
569             cmd_set => System_Defs.text'(14, 14, "$OEO/acct_cmds"));
570
571         -- Open window for display output:
572         --
573         old_opened_window := Process_Mgt.Get_process_globals_entry(
574             slot => Process_Mgt_Types.standard_input);
575         -- Retrieve standard input.
576
577         old_window := Character_Display_AM.Ops.Get_device_object(
578             opened_dev => old_opened_window);
579         -- Retrieve the window underlying standard input.
580
581         underlying_terminal := Window_Services.Ops.Get_terminal(
582             window => old_window);
583         -- Retrieve underlying terminal.
584
585         new_window := Window_Services.Ops.Create_window(
586             terminal => underlying_terminal,
587             pixel_units => false,
588             fb_size => frame_buffer,
589             desired_window_size => window_size,
590             window_pos => window_pos,
591             view_pos => view_pos);
592
593         Text_Mgt.Set(
594             dest => new_window_info.title,
595             source => title_string);
596
597
598         -- Window_Services.Ops.Set_window_style(
599         --     window => new_opened_window,
600         --     new_info => new_window_info,
601         --     style_list => Window_Services.window_style_mask'
602         --         (Window_Services.set_title => true,
603         --         others => false));
604
605         new_opened_window := Character_Display_AM.Ops.Open(
606             device => new_window,
607             input_output => Device_Defs.output);
608
609         Character_Display_AM.Ops.Clear(new_opened_window);
610
611         Character_Display_AM.Ops.Move_cursor_absolute(
612             opened_dev => new_opened_window,

```

```

613         new_pos    => Terminal_Defs.point_info' (15,2));
614
615     curr_pos := Terminal_Defs.point_info' (3, 5);
616
617     Message_Services.Write_msg(
618         msg_id    => Incident_Defs.incident_code'
619             (1, 1, information, System.null_word),
620         no_header => true,
621         device   => new_opened_window);
622
623     curr_pos := Terminal_Defs.point_info' (3, 2);
624
625
626
627     loop
628     begin
629         -- Program block to handle exceptions.
630
631         Command_Handler.Get_command(
632             cmd_odo => input,
633             cmd_id  => cmd_id,
634             cmd_name => cmd_name);
635
636         case cmd_id is
637
638             -- CREATE:
639             --
640             when 0 =>
641                 -- A. Get argument from command line:
642                 --
643                 initial_balance := Command_Handler.Get_integer(
644                     cmd_odo    => input,
645                     name       => System_Defs.text' (12, 12,"init_balance"));
646
647                 -- B. Check whether there is space available:
648                 --
649                 if list_exceeded then
650                     -- Out of space.
651                     RAISE list_exceeded_exc;
652
653                 else
654                     -- Space available
655                     long_initial_balance :=
656                         Long_Integer_Defs.long_integer' (0,
657                             System.ordinal(initial_balance));
658                     -- Convert integer to long integer.
659
660                     -- C. Create account and add to local list:
661                     --
662                     local_list(list_pointer).AD :=
663                         Account_Mgt_Ex.Create_account(
664                             starting_balance => long_initial_balance);
665                     local_list(list_pointer).number := list_pointer;
666                     Text_Mgt.Set(
667                         dest    => local_list(list_pointer).name,
668                         source  => Acct_Types.local_text);
669                     local_list(list_pointer).stored := false;
670
671                     if list_pointer = Acct_Types.list_length_limit then
672                         list_exceeded := true;
673                         RAISE list_exceeded_exc;
674
675                     end if;
676                     list_pointer := list_pointer+1;
677                     RAISE mission_accomplished;
678                 end if;
679
680
681
682
683             -- CSTORE:
684             --
685             when 1 =>
686                 -- A. Get arguments from command line:
687                 --
688                 initial_balance := Command_Handler.Get_integer(
689                     cmd_odo    => input,

```

PRELIMINARY

```

690         name      => System_Defs.text'(12, 12, "init_balance"));
691
692 Command_Handler.Get_string(
693     cmd_odo  => input,
694     name     => System_Defs.text'(8, 8, "pathname"),
695     arg_value => pathname);
696
697 Command_Handler.Get_string(
698     cmd_odo  => input,
699     name     => System_Defs.text'(9, 9, "authority"),
700     arg_value => authority_name);
701
702 if list_exceeded then
703     -- Out of space.
704     RAISE list_exceeded_exc;
705
706 else
707     -- Space available
708     long_initial_balance :=
709         Long_Integer_Defs.long_integer'(0,
710         System.ordinal(initial_balance));
711     -- Convert integer to long integer.
712
713     if Text_Mgt.Equal(authority_name, Acct_Types.none_text) then
714         -- No authority list was specified. Use default.
715         auth_list := null;
716
717     else
718         auth_list := Conversion_Support_Ex.
719             Authority_list_from_untyped(
720             Directory_Mgt.Retrieve(authority_name));
721         -- Retrieve authority list;
722
723     end if;
724
725     -- C. Create account and add to local list:
726     --
727     local_list(list_pointer).AD      :=
728         Account_Mgt_Ex.Create_stored_account(
729         starting_balance => long_initial_balance,
730         master           => pathname,
731         authority        => auth_list);
732
733     local_list(list_pointer).number := list_pointer;
734     local_list(list_pointer).name   := pathname;
735     local_list(list_pointer).stored := true;
736
737
738     if list_pointer = Acct_Types.list_length_limit then
739         list_exceeded := true;
740         RAISE list_exceeded_exc;
741
742     end if;
743
744 end if;
745 list_pointer := list_pointer+1;
746 RAISE mission_accomplished;
747
748
749 -- STORE:
750 --
751 when 2 =>
752     -- A. Get arguments from command line:
753     --
754     ref_number := Command_Handler.Get_integer(
755         cmd_odo  => input,
756         name     => System_Defs.text'(10, 10, "ref_number"));
757
758     Command_Handler.Get_string(
759         cmd_odo  => input,
760         name     => System_Defs.text'(8, 8, "pathname"),
761         arg_value => pathname);
762
763     Command_Handler.Get_string(
764         cmd_odo  => input,
765         name     => System_Defs.text'(8, 8, "pathname"),
766         arg_value => authority_name);

```

```

767
768     if Text_Mgt.Equal(local_list(ref_number).name,
769                       Acct_Types.empty_text)
770     then
771         -- Unassigned account.
772         RAISE invalid_account;
773
774     end if;
775
776     if Text_Mgt.Equal(authority_name, Acct_Types.none_text) then
777         -- No authority list was specified. Use default.
778         auth_list := null;
779
780     end if;
781
782     -- Enclose passive store operations in a transaction:
783     --
784     Transaction_Mgt.Start_transaction;
785     begin
786         Directory_Mgt.Store(
787             name     => pathname,
788             object   => Untyped_from_account(
789                 local_list(ref_number).AD),
790             aut      => auth_list);
791
792         Passive_Store_Mgt.Request_update(
793             obj     => Untyped_from_account(local_list(ref_number).AD));
794
795         Transaction_Mgt.Commit_transaction;
796     exception
797     when others =>
798         Transaction_Mgt.Abort_transaction;
799         RAISE;
800
801     end;
802
803
804     local_list(ref_number).name := pathname;
805     local_list(ref_number).stored := true;
806
807
808
809     -- RETRIEVE:
810     --
811     when 3 =>
812         -- A. Get arguments from command line:
813         --
814         Command_Handler.Get_string(
815             cmd_odo  => input,
816             name     => System_Defs.text'(8, 8, "pathname"),
817             arg_value => pathname);
818
819         if list_exceeded then
820             RAISE list_exceeded_exc;
821
822         else
823             -- B. Retrieve account and add to local list:
824             --
825             untyped_AD := Directory_Mgt.Retrieve(pathname);
826             if not Account_Mgt_Ex.Is_account(untyped_AD) then
827                 RAISE not_account;
828
829             end if;
830
831             local_list(list_pointer).AD :=
832                 Account_from_untyped(untyped_AD);
833             local_list(list_pointer).number := list_pointer;
834             local_list(list_pointer).name := pathname;
835             local_list(list_pointer).stored := true;
836
837             long_initial_balance := Account_Mgt_Ex.Get_balance(
838                 local_list(list_pointer).AD);
839
840             initial_balance := integer(long_initial_balance.l);
841
842             if list_pointer = Acct_Types.list_length_limit then
843                 list_exceeded := true;

```

## PRELIMINARY

```

844         RAISE list_exceeded_exc;
845
846     end if;
847     list_pointer := list_pointer+1;
848     RAISE mission_accomplished;
849
850 end if;
851
852 -- LIST:
853 --
854 when 4 =>
855     Character_Display_AM.Ops.Clear(new_opened_window);
856     Acct_visual.Display_list(
857         list           => local_list,
858         output         => new_opened_window,
859         pixel_units    => false,
860         location       => curr_pos);
861
862
863
864 -- DISPLAY:
865 --
866 when 5 =>
867     -- A. Get arguments from command line:
868     --
869     ref_number := Command_Handler.Get_integer(
870         cmd_odo  => input,
871         name     => System_Defs.text'(10, 10, "ref_number"));
872
873     if ref_number = 0 then
874         ref_number := list_pointer-1;
875     end if;
876
877     if Text_Mgt.Equal(local_list(ref_number).name,
878                     Acct_Types.empty_text)
879     then
880         -- Unassigned account.
881         RAISE invalid_account;
882
883     end if;
884
885     Character_Display_AM.Ops.Clear(new_opened_window);
886     Acct_visual.Display_account(
887         account      => local_list(ref_number).AD,
888         output       => new_opened_window,
889         pixel_units  => false,
890         location     => curr_pos);
891
892
893
894
895 -- WITHDRAW:
896 --
897 when 6 =>
898     -- A. Get arguments from command line:
899     --
900     ref_number := Command_Handler.Get_integer(
901         cmd_odo  => input,
902         name     => System_Defs.text'(10, 10, "ref_number"));
903
904     initial_balance := Command_Handler.Get_integer(
905         cmd_odo  => input,
906         name     => System_Defs.text'(6, 6, "amount"));
907
908     if Text_Mgt.Equal(local_list(ref_number).name,
909                     Acct_Types.empty_text)
910     then
911         -- Unassigned account.
912         RAISE invalid_account;
913
914     end if;
915
916     amount :=
917         Long_Integer_Defs.long_integer'(0,
918         System.ordinal(initial_balance));
919     -- Convert integer to long integer.
920

```

```

921         new_balance := Account_Mgt_Ex.Change_balance(
922             account => local_list(ref_number).AD,
923             amount => - amount);
924
925         RAISE new_balance_exc;
926
927
928     -- DEPOSIT:
929     --
930     when 7 =>
931         -- A. Get arguments from command line:
932         --
933         ref_number := Command_Handler.Get_integer(
934             cmd_odo => input,
935             name    => System_Defs.text'(10, 10, "ref_number"));
936
937         initial_balance := Command_Handler.Get_integer(
938             cmd_odo => input,
939             name    => System_Defs.text'(6, 6, "amount"));
940
941         if Text_Mgt.Equal(local_list(ref_number).name,
942             Acct_Types.empty_text)
943         then
944             -- Unassigned account.
945             RAISE invalid_account;
946
947         end if;
948
949         amount :=
950             Long_Integer_Defs.long_integer'(0,
951             System.ordinal(initial_balance));
952         -- Convert integer to long integer.
953
954         new_balance := Account_Mgt_Ex.Change_balance(
955             account => local_list(ref_number).AD,
956             amount => amount);
957
958         RAISE new_balance_exc;
959
960
961     -- TRANSFER:
962     --
963     when 8 =>
964         -- A. Get arguments from command line:
965         --
966         source_number := Command_Handler.Get_integer(
967             cmd_odo => input,
968             name    => System_Defs.text'(6, 6, "source"));
969
970         dest_number := Command_Handler.Get_integer(
971             cmd_odo => input,
972             name    => System_Defs.text'(11, 11, "destination"));
973
974         initial_balance := Command_Handler.Get_integer(
975             cmd_odo => input,
976             name    => System_Defs.text'(6, 6, "amount"));
977
978         if Text_Mgt.Equal(local_list(source_number).name,
979             Acct_Types.empty_text) or
980             Text_Mgt.Equal(local_list(dest_number).name,
981             Acct_Types.empty_text)
982         then
983             then
984             -- Unassigned account.
985             RAISE invalid_account;
986
987         end if;
988
989         amount :=
990             Long_Integer_Defs.long_integer'(0,
991             System.ordinal(initial_balance));
992         -- Convert integer to long integer.
993
994         Account_Mgt_Ex.Transfer(
995             source_account => local_list(source_number).AD,
996             dest_account   => local_list(dest_number).AD,
997             amount         => amount);

```

PRELIMINARY

```

998
999
1000
1001 -- REMOVE:
1002 --
1003 when 9 =>
1004 -- A. Get arguments from command line:
1005 --
1006 ref_number := Command_Handler.Get_integer(
1007   cmd_odo => input,
1008   name    => System_Defs.text'(10, 10, "ref_number"));
1009
1010 if Text_Mgt.Equal(local_list(ref_number).name,
1011   Acct_Types.empty_text)
1012 then
1013   -- Unassigned account.
1014   RAISE invalid_account;
1015
1016 end if;
1017
1018 Text_Mgt.Set(
1019   dest => local_list(ref_number).name,
1020   source => Acct_Types.empty_text);
1021
1022 RAISE account_removed;
1023
1024
1025 -- DESTROY:
1026 --
1027 when 10 =>
1028 -- A. Get arguments from command line:
1029 --
1030 ref_number := Command_Handler.Get_integer(
1031   cmd_odo => input,
1032   name    => System_Defs.text'(10, 10, "ref_number"));
1033
1034 if Text_Mgt.Equal(local_list(ref_number).name,
1035   Acct_Types.empty_text)
1036 then
1037   -- Unassigned account.
1038   RAISE invalid_account;
1039
1040 end if;
1041
1042 Account_Mgt_Ex.Destroy_account(local_list(ref_number).AD);
1043
1044 Text_Mgt.Set(
1045   dest => local_list(ref_number).name,
1046   source => Acct_Types.empty_text);
1047
1048 RAISE account_destroyed;
1049
1050
1051 -- MANAGE.AUTHORITY:
1052 --
1053 when 11 =>
1054   exit_status := Command_Execution.Execute_command(
1055     command => System_Defs.text'(16, 16, "manage.authority"));
1056
1057 -- SAVE:
1058 --
1059 when 12 =>
1060 -- A. Get arguments from command line:
1061 --
1062 Command_Handler.Get_string(
1063   cmd_odo => input,
1064   name    => System_Defs.text'(4, 4, "args"),
1065   arg_value => args);
1066
1067 Text_mgt.Set(
1068   dest => cmd_line,
1069   source => "ss ");
1070
1071 Text_Mgt.Append(
1072   dest => cmd_line,
1073   source => args);
1074

```

```

1075         exit_status := Command_Execution.Execute_command(
1076             command => cmd_line);
1077
1078
1079
1080         -- EXIT:
1081         --
1082         when 13 =>
1083             EXIT;
1084
1085         when others =>
1086             RAISE not_implemented;
1087
1088     end case;
1089
1090
1091 exception
1092     -- Main exception handler:
1093
1094     -- LOCAL:
1095     --
1096     when account_destroyed =>
1097         Text_Mgt.Set(
1098             dest => aux_text,
1099             source => pathname);
1100         Message_Services.Write_msg(
1101             msg_id => Incident_Defs.incident_code'
1102                 (1, 13, information, System.null_word),
1103             param1 => Incident_Defs.message_parameter'
1104                 (txt, pathname.length, aux_text),
1105             no_header => true);
1106
1107     when account_removed =>
1108         Message_Services.Write_msg(
1109             msg_id => Incident_Defs.incident_code'
1110                 (1, 12, information, System.null_word),
1111             param1 => Incident_Defs.message_parameter'
1112                 (int, 4, integer(ref_number)),
1113             no_header => true);
1114
1115     when invalid_account =>
1116         Message_Services.Write_msg(
1117             msg_id => Incident_Defs.incident_code'
1118                 (1, 6, error, System.null_word),
1119             no_header => true);
1120
1121     when list_exceeded_exc =>
1122         Message_Services.Write_msg(
1123             msg_id => Incident_Defs.incident_code'
1124                 (1, 3, warning, System.null_word),
1125             param1 => Incident_Defs.message_parameter'
1126                 (int, 4, Acct_Types.list_length_limit),
1127             no_header => true);
1128
1129     when mission_accomplished =>
1130         Message_Services.Write_msg(
1131             msg_id => Incident_Defs.incident_code'
1132                 (1, 2, information, System.null_word),
1133             param1 => Incident_Defs.message_parameter'
1134                 (int, 4, list_pointer-1),
1135             param2 => Incident_Defs.message_parameter'
1136                 (int, 4, initial_balance),
1137             no_header => true);
1138
1139     when new_balance_exc =>
1140         Message_Services.Write_msg(
1141             msg_id => Incident_Defs.incident_code'
1142                 (1, 11, warning, System.null_word),
1143             param1 => Incident_Defs.message_parameter'
1144                 (int, 4, integer(new_balance.l)),
1145             no_header => true);
1146
1147     when not_account =>
1148         Text_Mgt.Set(
1149             dest => aux_text,
1150             source => pathname);
1151         Message_Services.Write_msg(

```

PRELIMINARY

```

1152         msg_id    => Incident_Defs.incident_code'
1153         (1, 14, error, System.null_word),
1154         param1    => Incident_Defs.message_parameter'
1155         (txt, pathname.length, aux_text),
1156         no_header => true);
1157
1158     when not_implemented =>
1159         Message_Services.Write_msg(
1160             msg_id    => Incident_Defs.incident_code'
1161             (1, 9, warning, System.null_word),
1162             no_header => true);
1163
1164
1165     -- ACCOUNT_MGT_EX:
1166     --
1167     when Account_mgt_Ex.balance_not_zero =>
1168         Message_Services.Write_msg(
1169             msg_id    => Incident_Defs.incident_code'
1170             (0, 2, error, System.null_word),
1171             no_header => true);
1172
1173     when Account_Mgt_Ex.insufficient_balance =>
1174         Message_Services.Write_msg(
1175             msg_id    => Incident_Defs.incident_code'
1176             (0, 1, error, System.null_word),
1177             no_header => true);
1178
1179     -- DIRECTORY_MGT:
1180     --
1181     when Directory_Mgt.entry_exists =>
1182         Message_Services.Write_msg(
1183             msg_id    => Incident_Defs.incident_code'
1184             (1, 7, error, System.null_word),
1185             no_header => true);
1186
1187     when Directory_Mgt.no_access =>
1188         Message_Services.Write_msg(
1189             msg_id    => Incident_Defs.incident_code'
1190             (1, 5, error, System.null_word),
1191             no_header => true);
1192
1193     when Directory_Mgt.no_default_authority_list =>
1194         Message_Services.Write_msg(
1195             msg_id    => Incident_Defs.incident_code'
1196             (1, 8, error, System.null_word),
1197             no_header => true);
1198
1199     -- PASSIVE_STORE_MGT:
1200     --
1201     when Passive_Store_Mgt.no_master_AD =>
1202         Message_Services.Write_msg(
1203             msg_id    => Incident_Defs.incident_code'
1204             (1, 16, error, System.null_word),
1205             no_header => true);
1206
1207     -- SYSTEM_EXCEPTIONS:
1208     --
1209     when System_Exceptions.insufficient_type_rights =>
1210         Message_Services.Write_msg(
1211             msg_id    => Incident_Defs.incident_code'
1212             (1, 15, warning, System.null_word),
1213             no_header => true);
1214
1215     when System_Exceptions.operation_not_supported =>
1216         Message_Services.Write_msg(
1217             msg_id    => Incident_Defs.incident_code'
1218             (1, 10, warning, System.null_word),
1219             no_header => true);
1220
1221     --
1222     when others =>
1223         Message_Services.Write_msg(
1224             msg_id    => Incident_Defs.incident_code'
1225             (1, 4, error, System.null_word),
1226             no_header => true);
1227         RAISE;
1228 end;
```

```
1229     end loop;  
1230  
1231     end Acct_main_loop;  
1232
```

## X-A.7.2 Acct\_Visual Package Specification

Display routines used by the account manager test driver.

```

1  with
2    Account_Mgt_Ex,
3    Acct_Types,
4    Authority_List_Mgt,
5    Device_Defs,
6    Long_Integer_Defs,
7    System_Defs,
8    Terminal_Defs;
9
10 package Acct_visual is
11 --
12 -- Function:
13 --   This package contains procedures to display
14 --   information about accounts. It is called by the
15 --   Acct_main procedure.
16 --
17 -- Calls:
18 --   o Display_account   Given an AD displays all information relevant to
19 --                       the account, i. e. pathname, creator, creation,
20 --                       creation time, time last read, time last modified.
21 --                       and the current balance.
22 --
23 --   o List_account     Given a Acct_main.list, displays that list.
24 --
25 -- Exceptions:
26 --
27
28 procedure Display_account (
29   account: Account_Mgt_Ex.account_AD;
30   -- Account that is to be displayed.
31   output: Device_Defs.opened_device;
32   -- Device to use for displaying info.
33   pixel_units: boolean := false;
34   -- Whether to use character- or pixel units.
35   location: Terminal_Defs.point_info);
36   -- Where to display the account.
37 --
38 -- Function:
39 --   Displays relevant information about an account
40 --   in the following format:
41 --
42 --   +-----+
43 --   | NAME:           ///bla/bla/acct/boz01
44 --   | CREATOR:       ///bla/bla/id/bozo
45 --   | CREATED:       12/12/1212   15:43:59
46 --   | LAST READ:    12/12/1212   15:43:59
47 --   | LAST MODIFIED 12/12/1212   15:43:59
48 --   |
49 --   |           Current Balance:   $ 146358.00
50 --   +-----+
51 --
52 --   For accounts that have no passive version the display will
53 --   look like this:
54 --
55 --   +-----+
56 --   | NAME:           local
57 --   | CREATOR:
58 --   | CREATED:
59 --   | LAST READ:
60 --   | LAST MODIFIED:
61 --   |
62 --   |           Current Balance:   $ 146358.00
63 --   +-----+
64 --
65
66 procedure Display_list (
67   list: Acct_Types.list;
68   -- List to display.
69   output: Device_Defs.opened_device;
70   -- Device to use for displaying info.
71   pixel_units: boolean := false;
72   -- Whether to use character- or pixel units.
73   location: Terminal_Defs.point_info);

```

```
74      -- Where to display the list.
75      --
76      -- Function:
77      --   Displays a list of local account in the following format:
78      --
79      --   <ref_number> <stored>      <name>
80      --           1  stored    ///Gemini/State/home/tobiash/savings
81      --           2  local     ///Gemini/State/home/martinb/checking
82      --           3  stored    ///Gemini/State/home/patty/stocks
83      --
84      --
85
86      pragma external;
87
88      end Acct_visual;
```

## X-A.7.3 Acct\_visual Package Body

Display routines used by the account manager test driver.

```

1  --*D*
2  --*D* manage.messages
3  --*D*
4  --*D*   store      :module   = 2 \
5  --*D*                :number   = 1 \
6  --*D*                :msg_name = acknowledge \
7  --*D*                :short    = "Type any character to continue> "
8  --*D* exit
9
10 with
11   Account_Mgt_Ex,
12   Acct_Types,
13   Character_Display_AM,
14   Device_Defs,
15   Directory_Mgt,
16   Incident_Defs,
17   Long_Integer_Defs,
18   Message_Services,
19   Passive_Store_Mgt,
20   System,
21   System_Defs,
22   System_Exceptions,
23   Terminal_Defs,
24   Text_Mgt,
25   Timing_Conversions,
26   Timing_String_Conversions;
27
28
29 package body Acct_visual is
30
31
32   procedure Display_account(
33     account:      Account_Mgt_Ex.account_AD;
34     output:       Device_Defs.opened_device;
35     pixel_units: boolean := false;
36     location: in Terminal_Defs.point_info)
37   is
38     account_untyped: System.untyped_word;
39     FOR account_untyped USE AT account'address;
40     -- Untyped overlay.
41
42     account_info:   Passive_Store_Mgt.passive_object_info;
43     name_value:     System_Defs.text(Acct_Types.name_length_limit);
44     creator_value:  System_Defs.text(Acct_Types.name_length_limit);
45     created_value:  System_Defs.text(22);
46     read_value:     System_Defs.text(22);
47     write_value:    System_Defs.text(22);
48     bal_value:      Long_Integer_Defs.string_integer;
49
50     num_time:       Timing_Conversions.numeric_time;
51
52
53     no_name:        boolean := false;
54     position:       Terminal_Defs.point_info;
55     ID_untyped:     System.untyped_word;
56     FOR id_untyped USE AT account_info.owner'address;
57     num_bal:        Long_Integer_Defs.long_integer;
58
59     tb_line:        constant System_Defs.text := System_Defs.text'(65, 65,
60 "+-----+");
61     side:           constant System_Defs.text := System_Defs.text'(1, 1,
62 "|");
63     name:           constant System_Defs.text := System_Defs.text'(5, 5,
64 "NAME:");
65     creator:        constant System_Defs.text := System_Defs.text'(8, 8,
66 "CREATOR:");
67     created:        constant System_Defs.text := System_Defs.text'(8, 8,
68 "CREATED:");
69     read:           constant System_Defs.text := System_Defs.text'(10, 10,
70 "LAST READ:");
71     write:          constant System_Defs.text := System_Defs.text'(14, 14,
72 "LAST MODIFIED:");
73     bal:            constant System_Defs.text := System_Defs.text'(18, 18,

```

```

74         "CURRENT BALANCE: $");
75
76     begin
77         -- 1. Display account template:
78         --
79         position := location;
80         Character_Display_AM.Ops.Clear_to_bottom(output);
81         Character_Display_AM.Ops.Move_cursor_absolute(
82             opened_dev => output,
83             new_pos     => position);
84         Character_Display_AM.Ops.Write(
85             -- Top line of box.
86             opened_dev => output,
87             buffer_VA  => tb_line.value'address,
88             length     => System.ordinal(tb_line.length));
89
90         for i in 1 .. 7 loop
91             position.vert := location.vert+i;
92             position.horiz := location.horiz;
93             Character_Display_AM.Ops.Move_cursor_absolute(
94                 opened_dev => output,
95                 new_pos     => position);
96             Character_Display_AM.Ops.Write(
97                 -- Left side of box
98                 opened_dev => output,
99                 buffer_VA  => side.value'address,
100                length     => System.ordinal(side.length));
101
102             position.horiz := location.horiz+74;
103             Character_Display_AM.Ops.Move_cursor_absolute(
104                 opened_dev => output,
105                 new_pos     => position);
106             Character_Display_AM.Ops.Write(
107                 -- Right side of box.
108                 opened_dev => output,
109                 buffer_VA  => side.value'address,
110                 length     => System.ordinal(side.length));
111
112         end loop;
113
114         position.vert := location.vert+8;
115         position.horiz := location.horiz;
116         Character_Display_AM.Ops.Move_cursor_absolute(
117             opened_dev => output,
118             new_pos     => position);
119         Character_Display_AM.Ops.Write(
120             -- Bottom line of box.
121             opened_dev => output,
122             buffer_VA  => tb_line.value'address,
123             length     => System.ordinal(tb_line.length));
124
125         position.horiz := location.horiz+1;
126         position.vert := location.vert+1;
127         Character_Display_AM.Ops.Move_cursor_absolute(
128             opened_dev => output,
129             new_pos     => position);
130         Character_Display_AM.Ops.Write(
131             -- Write "NAME:" in position 2,2.
132             opened_dev => output,
133             buffer_VA  => name.value'address,
134             length     => System.ordinal(name.length));
135
136         position.vert := position.vert+1;
137         Character_Display_AM.Ops.Move_cursor_absolute(
138             opened_dev => output,
139             new_pos     => position);
140         Character_Display_AM.Ops.Write(
141             -- Write "CREATOR:" in position 3,2.
142             opened_dev => output,
143             buffer_VA  => creator.value'address,
144             length     => System.ordinal(creator.length));
145
146         position.vert := position.vert+1;
147         Character_Display_AM.Ops.Move_cursor_absolute(
148             opened_dev => output,
149             new_pos     => position);
150         Character_Display_AM.Ops.Write(

```

**PRELIMINARY**

```

151         -- Write "CREATED:" in position 4,2.
152         opened_dev => output,
153         buffer_VA  => created.value'address,
154         length     => System.ordinal(created.length));
155
156     position.vert := position.vert+1;
157     Character_Display_AM.Ops.Move_cursor_absolute(
158         opened_dev => output,
159         new_pos    => position);
160     Character_Display_AM.Ops.Write(
161         -- Write "LAST READ:" in position 5,2.
162         opened_dev => output,
163         buffer_VA  => read.value'address,
164         length     => System.ordinal(read.length));
165
166     position.vert := position.vert+1;
167     Character_Display_AM.Ops.Move_cursor_absolute(
168         opened_dev => output,
169         new_pos    => position);
170     Character_Display_AM.Ops.Write(
171         -- Write "LAST MODIFIED:" in position 6,2.
172         opened_dev => output,
173         buffer_VA  => write.value'address,
174         length     => System.ordinal(write.length));
175
176     position.vert := position.vert+2;
177     Character_Display_AM.Ops.Move_cursor_absolute(
178         opened_dev => output,
179         new_pos    => position);
180     Character_Display_AM.Ops.Write(
181         -- Write "CURRENT BALANCE: $" in position 8,2.
182         opened_dev => output,
183         buffer_VA  => bal.value'address,
184         length     => System.ordinal(bal.length));
185
186
187
188     -- 2. Determine whether "account_AD" references an account
189     -- with a passive version. If yes, get the account's name:
190     --
191     begin
192         -- This block controls the scope of the exception handler.
193         Directory_Mgt.Get_name(
194             obj => account_untyped,
195             name => name_value);
196
197     exception
198     when Directory_Mgt.no_name =>
199         Text_Mgt.Set(
200             dest    => name_value,
201             source  => Acct_Types.local_text);
202         no_name := true;
203
204     when others =>
205         RAISE;
206
207     end;
208
209     --
210     -- 3. Initialize values for
211     -- - Creator
212     -- - Creation Time
213     -- - Time Last Read
214     -- - Time Last Modified
215     -- - Current Balance
216     -- If account is unnamed initialize to "local".
217     --
218     if no_name then
219         -- Account has no name and therefore has not
220         -- been passivated.
221         Text_Mgt.Set(
222             dest    => creator_value,
223             source  => Acct_Types.local_text);
224         Text_Mgt.Set(
225             dest    => created_value,
226             source  => Acct_Types.local_text);
227         Text_Mgt.Set(

```

```

228         dest => read_value,
229         source => Acct_Types.local_text);
230     Text_Mgt.Set(
231         dest => write_value,
232         source => Acct_Types.local_text);
233
234     else
235         -- Account has a name and has been passivated.
236         account_info := Passive_Store_Mgt.Request_passive_object_info(
237             obj => account_untyped);
238
239         Directory_Mgt.Get_name(
240             -- Obtain user name of owner from ID.
241             obj => ID_untyped,
242             name => creator_value);
243
244         num_time := Timing_Conversions.Convert_stu_to_numeric_time(
245             stu => account_info.create_time);
246         Timing_string_conversions.Convert_numeric_time_to_ISO(
247             num_time => num_time,
248             ISO_time => created_value);
249
250         num_time := Timing_Conversions.Convert_stu_to_numeric_time(
251             stu => account_info.read_time);
252         Timing_string_conversions.Convert_numeric_time_to_ISO(
253             num_time => num_time,
254             ISO_time => read_value);
255
256         num_time := Timing_Conversions.Convert_stu_to_numeric_time(
257             stu => account_info.write_time);
258         Timing_string_conversions.Convert_numeric_time_to_ISO(
259             num_time => num_time,
260             ISO_time => write_value);
261     end if;
262
263     -- 4. Get balance and convert to suitable format:
264     --
265     num_bal := Account_Mgt_Ex.Get_balance(account);
266
267     Long_Integer_Defs.Long_integer_image(
268         number => num_bal,
269         image => bal_value);
270
271     -- 5. Display values:
272     --
273     position.horiz := location.horiz+9;
274     position.vert := location.vert+1;
275     Character_Display_AM.Ops.Move_cursor_absolute(
276         opened_dev => output,
277         new_pos => position);
278     Character_Display_AM.Ops.Write(
279         opened_dev => output,
280         buffer_VA => name_value.value'address,
281         length => System.ordinal(name_value.length));
282
283     position.horiz := location.horiz+16;
284     position.vert := position.vert+1;
285     Character_Display_AM.Ops.Move_cursor_absolute(
286         opened_dev => output,
287         new_pos => position);
288     Character_Display_AM.Ops.Write(
289         opened_dev => output,
290         buffer_VA => creator_value.value'address,
291         length => System.ordinal(creator_value.length));
292
293     position.vert := position.vert+1;
294     Character_Display_AM.Ops.Move_cursor_absolute(
295         opened_dev => output,
296         new_pos => position);
297     Character_Display_AM.Ops.Write(
298         opened_dev => output,
299         buffer_VA => created_value.value'address,
300         length => System.ordinal(created_value.length));
301
302     position.vert := position.vert+1;
303     Character_Display_AM.Ops.Move_cursor_absolute(
304         opened_dev => output,

```

**PRELIMINARY**

```

305         new_pos     => position);
306     Character_Display_AM.Ops.Write(
307         opened_dev => output,
308         buffer_VA  => read_value.value'address,
309         length     => System.ordinal(read_value.length));
310
311     position.vert := position.vert+1;
312     Character_Display_AM.Ops.Move_cursor_absolute(
313         opened_dev => output,
314         new_pos     => position);
315     Character_Display_AM.Ops.Write(
316         opened_dev => output,
317         buffer_VA  => write_value.value'address,
318         length     => System.ordinal(write_value.length));
319
320     position.vert := position.vert+2;
321     position.horiz := location.horiz+20;
322     Character_Display_AM.Ops.Move_cursor_absolute(
323         opened_dev => output,
324         new_pos     => position);
325     Character_Display_AM.Ops.Write(
326         opened_dev => output,
327         buffer_VA  => bal_value'address,
328         length     => 31);
329
330     end Display_account;
331
332
333
334     procedure Display_list(
335         list:      Acct_Types.list;
336         -- List to display.
337         output:    Device_Defs.opened_device;
338         -- Device to use for displaying info.
339         pixel_units: boolean := false;
340         -- Whether to use character- or pixel units.
341         location:  Terminal_Defs.point_info)
342         -- Where to display the list.
343
344     is
345
346
347     -- Auxiliary variables:
348     --
349     i:      integer;
350     cur_pos: Terminal_Defs.point_info;
351     yes:    boolean;
352     number: System_Defs.text(5);
353     step:   integer;
354     act_len: integer;
355
356
357
358     begin
359         step := 0;
360         cur_pos.horiz := 1;
361         cur_pos.vert := location.vert;
362         Character_Display_AM.Ops.Move_cursor_absolute(
363             opened_dev => output,
364             new_pos     => cur_pos);
365         Character_Display_AM.Ops.Clear_to_bottom(output);
366
367         cur_pos := location;
368         Character_Display_AM.Ops.Move_cursor_absolute(
369             opened_dev => output,
370             new_pos     => cur_pos);
371
372         for i in Acct_Types.list_pointer_init ..
373             Acct_Types.list_length_limit loop
374
375             if not Text_Mgt.Equal(list(i).name, Acct_Types.empty_text) then
376                 act_len := integer'image(list(i).number)'length;
377
378                 declare
379                     aux_str: string(1..act_len);
380                 begin
381                     aux_str := integer'image(list(i).number);

```

```

382         Text_Mgt.Set (
383             dest => number,
384             source => aux_str);
385     end;
386
387     step := step+1;
388     cur_pos.vert := location.vert + (step mod 8);
389     cur_pos.horiz := location.horiz+3;
390     Character_Display_AM.Ops.Move_cursor_absolute(
391         opened_dev => output,
392         new_pos => cur_pos);
393     Character_Display_AM.Ops.Write(
394         opened_dev => output,
395         buffer_VA => number.value'address,
396         length => System.ordinal(number.length));
397
398     cur_pos.horiz := cur_pos.horiz+5;
399     Character_Display_AM.Ops.Move_cursor_absolute(
400         opened_dev => output,
401         new_pos => cur_pos);
402
403     if list(i).stored then
404         Character_Display_AM.Ops.Write(
405             opened_dev => output,
406             buffer_VA => Acct_Types.stored_text.value'address,
407             length => System.ordinal(Acct_Types.stored_text.length));
408     end if;
409
410     cur_pos.horiz := cur_pos.horiz+Acct_Types.stored_text.length+2;
411     Character_Display_AM.Ops.Move_cursor_absolute(
412         opened_dev => output,
413         new_pos => cur_pos);
414     Character_Display_AM.Ops.Write(
415         opened_dev => output,
416         buffer_VA => list(i).name.value'address,
417         length => System.ordinal(list(i).name.length));
418
419     if step mod 7 = 0 then
420         yes := Message_Services.Acknowledge_msg(
421             msg_id => Incident_Defs.incident_code'
422                 (2, 1, Incident_Defs.information, System.null_word));
423         cur_pos.horiz := 1;
424         cur_pos.vert := location.vert;
425         Character_Display_AM.Ops.Move_cursor_absolute(
426             opened_dev => output,
427             new_pos => cur_pos);
428         Character_Display_AM.Ops.Clear_to_bottom(output);
429     end if;
430
431     end if;
432
433     end if;
434
435     end loop;
436
437     end Display_list;
438
439     end Acct_visual;

```

## X-A.7.4 Account Manager Command File

Account manager command file.

```

1   set.program acct
2   create.invocation_command
3   -- :set_def = acct_cmds
4   --
5   -- Invokes the Account Manager.
6   --
7   end
8
9
10
11  create.runtime_command_set :cmd_def = acct_cmds \
12                                :prompt = "ACCT_MGT> "
13  --
14  -- Runtime commands of the account manager.
15  --
16
17  define.command :cmd_name = create
18  --
19  -- Create a new account with an initial balance.
20  --
21  --
22  define.argument :arg_name = init_balance \
23                                :type = integer
24  set.bounds :value = 0..100000
25  set.mandatory
26  set.description :text = "
27  -- Initial balance of an account.
28  -- Must be between 0 and 100000.
29  "
30  end
31  set.description :text = "
32  -- Description:
33  --   Creates a local account with an initial balance.
34  --   Account is not stored and will go away when program
35  --   terminates unless it is stored prior to exiting.
36  --
37  -- Examples:
38  --   *> create 10000
39  --   Creates an account with an initial balance of 10000.
40  --
41  -- See Also:
42  --
43  "
44  end
45
46
47  define.command :cmd_name = cstore
48  --
49  -- Create and store a new account in one step.
50  --
51  --
52  define.argument :arg_name = pathname \
53                                :type = string
54  set.maximum_length 43
55  set.mandatory
56  set.description :text = "
57  -- Pathname to store the account. Must be
58  -- a valid pathname that is not already in use.
59  -- Caller must have store rights in the referenced
60  -- directory.
61  "
62  end
63
64  define.argument :arg_name = init_balance \
65                                :type = integer
66  set.bounds :value = 0..100000
67  set.mandatory
68  set.description :text = "
69  -- Initial balance of the account. Must be
70  -- greater or equal to zero and less than or equal
71  -- to 100000.
72  "
73  end

```

```

74
75     define.argument :arg_name = authority \
76                   :type     = string
77     set.maximum_length 43
78     set.value_default :value = "none"
79     set.description :text = "
80         -- Specifies an authority list to be stored
81         -- with an account. Has to be created separately
82         -- invoking the manage.authority runtime command.
83         -- Default value is none.
84     "
85 end
86 set.description :text = "
87 -- Description:
88 --   CSTORE creates a local account with an initial balance
89 --   and stores the account with a pathname. The pathname must
90 --   reference an existing directory and must not already be
91 --   in use. The implementation must support stored accounts,
92 --   otherwise System_Exceptions.operation_not_supported will
93 --   be raised.
94 --
95 -- Examples:
96 --   *> cstore 10000 al
97 --   Creates an account called al with an initial balance of
98 --   10000
99 --
100 -- See Also:
101 --
102 "
103 end
104
105
106 define.command :cmd_name = store
107 --
108 -- Store an existing local account
109 --
110
111     define.argument :arg_name = ref_number \
112                   :type     = integer
113     set.description :text = "
114         -- Reference to an account. Has to be
115         -- between 1 and 100.
116     "
117     set.bounds :value = 1..100
118     set.mandatory
119 end
120
121
122     define.argument :arg_name = pathname \
123                   :type     = string
124     set.description :text = "
125         -- Pathname to store the account. Must be
126         -- a valid pathname that is not already in use.
127         -- Caller must have store rights in the referenced
128         -- directory.
129     "
129     set.maximum_length 43
130     set.mandatory
131 end
132
133
134     define.argument :arg_name = authority \
135                   :type     = string
136     set.description :text = "
137         -- Specifies an authority list to be stored
138         -- with an account. Has to be created separately
139         -- invoking the manage.authority runtime command.
140         -- Default value is none.
141     "
141     set.maximum_length 43
142     set.value_default :value = "none"
143 end
144 set.description :text = "
145 -- Description:
146 --   Store an existing active account.
147 --   The implementation must support stored accounts.
148 --   Otherwise this operation will fail and the
149 --   'System_Exceptions.operation' will be raised.
150 --

```

PRELIMINARY

```

151         -- Examples:
152         -- *> store :ref_number = 3 :pathname = p177
153         -- Stores an account that has previously been
154         -- created and assigned local number 3 with
155         -- pathname 'p177'.
156         --
157         -- See Also:
158         --
159         "
160     end
161
162
163     define.command :cmd_name = retrieve
164     --
165     -- Make a stored account available for processing.
166     --
167     define.argument :arg_name = pathname \
168         :type = string
169     set.description :text = "
170         -- Pathname of a account to be retrieved. Can
171         -- be relative, absolute, or network pathname.
172         -- Must be a valid pathname and pathname must
173         -- reference an account.
174         "
175     set.maximum_length :value = 43
176     set.mandatory
177     end
178     set.description :text = "
179         -- Description:
180         -- Retrieve a stored account from a pathname
181         -- and make it available for online processing.
182         --
183         -- Examples:
184         -- *> retrieve :pathname = p177
185         -- Retrieves account named 'p177' in the current
186         -- working directory and places it on the local list
187         -- with the lowest available local number. 'pathname'
188         -- must reference an account. Otherwise operation fails.
189         --
190         -- See Also:
191         --
192         "
193     end
194
195
196
197     define.command :cmd_name = "list"
198     --
199     -- List all accounts available for local processing.
200     --
201     set.description :text = "
202         -- Description:
203         -- List all accounts currently available for
204         -- online processing by ordinal reference number.
205         --
206         -- Examples:
207         --
208         -- See Also:
209         --
210         "
211     end
212
213
214     define.command :cmd_name = display
215     --
216     -- Display all relevant information about an account.
217     --
218     define.argument :arg_name = ref_number \
219         :type = integer
220     set.description :text = "
221         -- Ordinal number referencing a local account
222         "
223     set.bounds :value = 0..100
224     set.value_default :value = 0
225     end
226     set.description :text = "
227         -- Description:

```

```

228      -- Display all relevant information about an account.
229      -- This is
230      -- NAME          full network pathname.
231      -- CREATOR      full name of owner.
232      -- CREATED       time when created.
233      -- LAST READ    time when last read.
234      -- LAST MODIFIED time when last modified.
235      -- CURRENT BALANCE current balance in account.
236      --
237      -- Examples:
238      --
239      -- See Also:
240      --
241      "
242  end
243
244
245  define.command :cmd_name = withdraw
246  --
247  -- Withdraw amount from local account.
248  --
249  define.argument :arg_name = ref_number \
250      :type      = integer
251      set.bounds :value = 1..100
252      set.mandatory
253      set.description :text = "
254          -- Reference to a local account from which
255          -- 'amount' is to be withdrawn.
256      "
257  end
258
259  define.argument :arg_name = amount \
260      :type      = integer
261      set.bounds :value = 0..100000
262      set.mandatory
263      set.description :text = "
264          -- Amount to be withdrawn. Must be less than
265          -- the current balance in the account.
266      "
267  end
268  set.description :text = "
269      -- Description:
270      --   Withdraw a given amount from a local account.
271      --   'amount' must be less than the current balance
272      --   in the account. Otherwise the operation will fail.
273      --
274      -- Examples:
275      --
276      -- See Also:
277      --
278      "
279  end
280
281
282  define.command :cmd_name = deposit
283  --
284  -- Deposit amount in local account.
285  --
286  define.argument :arg_name = ref_number \
287      :type      = integer
288      set.bounds :value = 1..100
289      set.mandatory
290      set.description :text = "
291          -- Reference to a local account in which
292          -- 'amount' is to be deposited.
293      "
294  end
295
296  define.argument :arg_name = amount \
297      :type      = integer
298      set.bounds :value = 0..100000
299      set.mandatory
300      set.description :text = "
301          -- Amount to be deposited.
302      "
303  end
304  set.description :text = "

```

```

305         -- Description:
306         --   Deposits a given amount in a local account.
307         --
308         -- Examples:
309         --
310         -- See Also:
311         --
312         "
313     end
314
315
316     define.command :cmd_name = transfer
317     --
318     -- Transfers an amount from one account to another.
319     --
320     define.argument :arg_name = source \
321         :type = integer
322         set.bounds :value = 1..100
323         set.mandatory
324         set.description :text = "
325             -- Source account for the transfer. The current
326             -- balance in this account must cover the transfer.
327         "
328     end
329
330     define.argument :arg_name = destination \
331         :type = integer
332         set.bounds :value = 1..100
333         set.mandatory
334         set.description :text = "
335             -- Destination account for the transfer.
336         "
337     end
338
339     define.argument :arg_name = amount \
340         :type = integer
341         set.bounds :value = 0..100000
342         set.mandatory
343         set.description :text = "
344             -- Amount to be transferred from 'source' to 'dest'.
345         "
346     end
347     set.description :text = "
348         -- Description:
349         --   Transfers 'amount' from 'source' to 'dest'. Transfer
350         --   happens as one atomic operation in implementations that
351         --   use transactions.
352         --
353         -- Examples:
354         --
355         -- See Also:
356         --
357         "
358     end
359
360
361     define.command :cmd_name = remove
362     --
363     -- Remove an account from the online processing.
364     --
365     define.argument :arg_name = ref_number \
366         :type = integer
367         set.bounds :value = 1..100
368         set.mandatory
369         set.description :text = "
370             -- Reference to a local account.
371         "
372     end
373     set.description :text = "
374         -- Description:
375         --   Remove an account from online processing
376         --   Does not affect an accounts passive version.
377         --
378         -- Examples:
379         --
380         -- See Also:
381         --

```

PRELIMINARY

```

382     "
383 end
384
385
386 define.command :cmd_name = destroy
387 --
388 -- Destroy an account.
389 --
390 define.argument :arg_name = ref_number \
391                 :type      = integer
392     set.bounds      :value = 1..100
393     set.mandatory
394 end
395 set.description :text = "
396     -- Description:
397     -- Destroys an account's passive version
398     -- if the implementation supports stored accounts.
399     -- Otherwise deallocates the account.
400     -- A stored account still has an online version
401     -- after a 'destroy'.
402     --
403     -- Examples:
404     --
405     -- See Also:
406     --
407     "
408 end
409
410
411 define.command :cmd_name = manage.authority
412 --
413 -- Invokes the 'manage.authority' utility.
414 --
415 set.description :text = "
416     -- Description:
417     --
418     -- Examples:
419     --
420     -- See Also:
421     --
422     "
423 end
424
425
426
427 define.command :cmd_name = save
428 --
429 -- Invoke screensaver utility.
430 --
431
432 define.argument :arg_name = "args" \
433                 :type      = string
434     set.description :text = "
435     -- Arguments to be passed on to
436     -- screensaver utility. Type
437     -- arguments exactly as you would
438     -- if you invoked the screensaver
439     -- from a shell, except enclose the
440     -- arguments in quotes.
441     "
442     end
443 end
444
445
446 define.command :cmd_name = "exit"
447 --
448 -- Exits 'acct'
449 set.description :text = "
450     -- Description:
451     --
452     -- Examples:
453     --
454     -- See Also:
455     --
456     "
457     end
458 end

```

## X-A.7.5 Account\_Types\_Ex Package Specification

```
1 with
2   Account_Mgt_Ex,
3   System_Defs;
4
5 package Acct_Types is
6   --
7   -- Global type definitions and constants for accounting program.
8   --
9   -- Constants:
10  --
11  name_length_limit: constant := 43;
12  list_length_limit: constant := 100;
13  message_length: constant := 55;
14  list_pointer_init: constant := 1;
15  empty_text: constant System_Defs.text :=
16              System_Defs.text'(5, 5, "empty");
17  none_text: constant System_Defs.text :=
18              System_Defs.text'(4, 4, "none");
19  local_text: constant System_Defs.text :=
20              System_Defs.text'(5, 5, "local");
21  stored_text: constant System_Defs.text :=
22              System_Defs.text'(6, 6, "stored");
23
24
25  -- Types:
26  --
27  subtype acct_enum is integer range 0 .. list_length_limit;
28
29  type local_account is
30    record
31      AD: Account_Mgt_Ex.account_AD;
32      number: acct_enum;
33      name: System_Defs.text(name_length_limit);
34      stored: boolean;
35    end record;
36
37  type list is
38    array(list_pointer_init .. list_pointer_init+list_length_limit-1)
39    of local_account;
40
41 end Acct_Types;
```

## X-A.7.6 Account\_Mgt\_Ex Package Specification

Common specification for active-only, non-transaction-oriented stored, transaction-oriented stored, and distributed account type managers.

```

1  with Authority_List_Mgt,
2      Incident_Defs,
3      Long_Integer_Defs,
4      Object_Mgt,
5      System,
6      System_Defs;
7
8  package Account_Mgt_Ex is
9      --
10     -- Function:
11     --   Type manager for accounts.  An account
12     --   contains a non-negative balance of type
13     --   "Long_Integer_Defs.long_integer".
14     --
15     --   Several aspects of accounts are
16     --   implementation-defined:
17     --
18     --     1. Whether accounts can be passivated.
19     --
20     --     2. What activation model is used for
21     --        accounts.
22     --
23     --     3. Whether account operations are
24     --        atomic, either succeeding completely
25     --        or failing completely.
26     --
27     --     4. Whether an account object can
28     --        simultaneously be used by multiple
29     --        processes within a single job.
30     --
31     --     5. Whether the account manager is
32     --        distributed, providing service at
33     --        at multiple nodes in a distributed
34     --        system, regardless of which nodes
35     --        accounts are stored at.
36     --
37     --     6. Some of the protection provided
38     --        between the account manager and other
39     --        services.
40     --
41     --     7. How and where the account TDO is defined
42     --        (so long as its lifetime is >= the lifetime
43     --        of any account).
44     --
45     --     8. Account attributes.
46     --
47     --     9. Account manager initialization requirements.
48     --
49     -- Calls:
50     --   Is_account           - Checks whether an AD
51     --                        references an account.
52     --
53     --   Create_account      - Creates an account
54     --                        with an initial balance.
55     --
56     --   Create_stored_account - Creates and stores an account.
57     --
58     --   Get_balance         - Returns an account's
59     --                        balance.
60     --
61     --   Change_balance      - Changes an account's
62     --                        balance.
63     --
64     --   Transfer            - Moves an amount between
65     --                        accounts.
66     --
67     --   Destroy_account     - Destroys an account.
68     --
69     -- Messages:
70
71     insufficient_balance_code:

```

PRELIMINARY

```

72     constant Incident_Defs.incident_code :=
73       (0, 1, Incident_Defs.error, System.null_word);
74
75     --*D* manage.messages
76     --*D* store :module=0 :number=1 \
77     --*D* :msg_name=insufficient_balance_code \
78     --*D* :short= \
79     --*D* "An account operation failed because it\
80     --*D* would create a negative balance."
81
82     balance_not_zero_code:
83     constant Incident_Defs.incident_code :=
84       (0, 2, Incident_Defs.error, System.null_word);
85
86     --*D* store :module=0 :number=2 \
87     --*D* :short= \
88     --*D* "An account cannot be destroyed because\
89     --*D* it has a non-zero balance."
90     --*D* exit
91
92     -- Exceptions:
93     --
94     insufficient_balance: exception;
95     pragma exception_value(insufficient_balance,
96       insufficient_balance_code'address);
97     -- An operation failed because it would
98     -- cause a negative account balance.
99
100    balance_not_zero: exception;
101    pragma exception_value(balance_not_zero,
102      balance_not_zero_code'address);
103    -- "Destroy_account" was called on an account
104    -- with a nonzero balance.
105
106    --
107    -- History:
108    -- 11-01-1985: Martin L. Buchanan, Initial version.
109    -- 04-04-1988: Tobias Haas
110    -- Revised in order to unify all
111    -- account manager examples.
112    --
113
114    type account_object is limited private;
115
116    type account_AD is access account_object;
117    pragma access_kind(account_AD, AD);
118    -- User view of an account.
119
120
121    change_rights: constant
122      Object_Mgt.rights_mask :=
123      Object_Mgt.modify_rights;
124    -- Required to change an account's balance.
125
126    destroy_rights: constant
127      Object_Mgt.rights_mask :=
128      Object_Mgt.control_rights;
129    -- Required to destroy an account.
130
131
132    function Is_account(
133      obj: System.untyped_word) -- AD to check.
134      return boolean;
135      -- true if "obj" references an account,
136      -- else false.
137      pragma protected_return(Is_account);
138      --
139      -- Function:
140      -- Checks whether "obj" references an
141      -- account.
142
143
144
145    function Create_account(
146      starting_balance:
147        Long_Integer_Defs.long_integer :=
148        Long_Integer_Defs.zero)

```

```

149     -- Initial balance of the account.
150     return account_AD;
151     -- New account with all type rights and no
152     -- rep rights.
153     pragma protected_return(Create_account);
154     --
155     -- Function:
156     -- Creates an account and returns an AD with all
157     -- type rights. The caller is responsible for
158     -- storing the AD and updating the object.
159     --
160     -- "starting_balance" must be nonnegative.
161     --
162     -- Exceptions:
163     -- insufficient_balance:
164     --     A negative balance was supplied.
165     --
166     -- Passive_Store_Mgt.no_master_AD:
167     --     The object provided to store the AD in, has
168     --     no master AD.
169
170
171     function Create_stored_account(
172         starting_balance:
173             Long_Integer_Defs.long_integer :=
174             Long_Integer_Defs.zero;
175         -- Initial balance of the account.
176         master: System_Defs.text;
177         -- Text record that holds the pathname
178         -- for the master AD.
179         authority:
180             Authority_List_Mgt.authority_list_AD :=
181             null)
182         -- Optional authority list.
183     return account_AD;
184     -- AD to the account with all type rights and no
185     -- rep rights.
186     pragma protected_return(Create_stored_account);
187     --
188     -- Function:
189     -- Creates a new account and stores the master AD
190     -- under the pathname given by "master".
191     -- Caller must have store rights for the named
192     -- directory.
193     -- The pathname cannot already be in use.
194     -- "starting_balance" must be nonnegative.
195     --
196     -- If "authority" is null, then the new account's
197     -- authority list will be either (in that order) the
198     -- containing directory's default authority list, if
199     -- there is one, or the caller's default authority list.
200     -- If none of these three is available,
201     -- "Directory_Mgt.no_default_authority_list" will be
202     -- raised.
203     --
204     -- Exceptions:
205     -- insufficient_balance:
206     --     A negative starting balance was supplied.
207     --
208     -- Directory_Mgt.entry_exists:
209     --     The pathname provided is already in use.
210     --
211     -- Directory_Mgt.no_default_authority_list:
212     --     No authority list was specified, the target
213     --     directory has no default authority list and there
214     --     is no default authority list in the caller's
215     --     process globals.
216
217
218     function Get_balance(
219         account: account_AD)
220         -- Any account.
221     return Long_Integer_Defs.long_integer;
222         -- Current balance.
223     pragma protected_return(Get_balance);
224     --
225     -- Function:

```

PRELIMINARY

```

226     -- Returns an account's current balance.
227
228
229 function Change_balance(
230     account: account_AD;
231     -- Account with change rights.
232     amount: Long_Integer_Defs.long_integer)
233     -- Amount added to balance.
234 return Long_Integer_Defs.long_integer;
235     -- New balance, equal to old balance
236     -- plus "amount".
237 pragma protected_return(Change_balance);
238 --
239 -- Function:
240 -- Adds "amount" to an account's balance
241 -- and returns the new balance. The new
242 -- balance cannot be negative.
243 --
244 -- Exceptions:
245 -- insufficient_balance
246
247
248 procedure Transfer(
249     source_account: account_AD;
250     -- Account with change rights.
251     dest_account: account_AD;
252     -- Account with change rights.
253     amount: Long_Integer_Defs.long_integer);
254     -- Amount transferred from source to
255     -- destination accounts; it can be
256     -- positive or negative. Cannot cause
257     -- a negative balance in either account.
258 pragma protected_return(Transfer);
259 --
260 -- Function:
261 -- Subtracts "amount" from "source_account"
262 -- and adds "amount" to "dest_account".
263 --
264 -- Exceptions:
265 -- insufficient_balance
266
267
268 procedure Destroy_account(
269     account: account_AD);
270     -- Account with destroy rights. The
271     -- account's balance must be zero.
272 pragma protected_return(Destroy_account);
273 --
274 -- Function:
275 -- Destroys an account.
276 --
277 -- The passive version, caller's active version,
278 -- and any master directory entry are destroyed.
279 --
280 -- Notes:
281 -- Any subsequent "Get_balance",
282 -- "Change_balance", or "Transfer" call
283 -- will raise "object_has_no_representation"
284 -- in the "System_Exceptions" package.
285 --
286 -- Exceptions:
287 -- balance_not_zero
288
289
290 pragma external;
291 -- Required if this package is used with the "virtual"
292 -- compilation model, which supports multiple domains
293 -- and multiple subsystems.
294
295 private
296
297 type account_object is
298     -- Empty dummy record. The real object
299     -- format is defined in the package body.
300     record
301         null;
302     end record;

```

```
303  
304 end Account_Mgt_Ex;
```

## X-A.7.7 Account\_Mgt\_Ex (Active Only) Package Body

Active-only package implementation of the account type manager.

```

1  with Access_Mgt,
2      Attribute_Mgt,
3      Long_Integer_Defs,
4      Object_Mgt,
5      Passive_Store_Mgt,
6      System_Defs,
7      System_Exceptions;
8
9  package body Account_Mgt_Ex is
10     --
11     -- Logic:
12     --   This is an 'active-only' implementation of
13     --   the account manager, with these characteristics:
14     --   1. Accounts cannot be passivated.
15     --
16     --   2. Account operations are atomic.
17     --
18     --   3. An account should not be concurrently
19     --   used by more than one process in a
20     --   single job.
21     --
22     --   4. Accounts and the account TDO are local
23     --   to the job that uses them.
24     --
25     --   5. The account TDO has the passive store
26     --   attribute.
27     --
28     --   6. Initialization of the account manager
29     --   is done within each job that uses it.
30     --   Initialization creates the account TDO
31     --   and assigns the passive store attribute
32     --   so that accounts are active-only.
33     --
34
35     use Long_Integer_Defs;
36     -- Import "long_integer" operators.
37
38     type account_rep_object is
39         record
40             balance: Long_Integer_Defs.long_integer;
41             -- Current balance.
42         end record;
43
44     type account_rep_AD is access account_rep_object;
45     pragma access_kind(account_rep_AD, AD);
46     -- Private view of an account.
47
48     account_TDO: constant Object_Mgt.TDO_AD :=
49         Object_Mgt.Create_TDO;
50     -- This declaration is elaborated each time
51     -- this package is initialized, that is, each
52     -- time a job using the package runs. This
53     -- technique for creating a TDO is only useful
54     -- for objects that are completely local to
55     -- a job and never stored or otherwise exported
56     -- outside the creating job.
57
58
59     function Is_account(
60         obj: System.untyped_word)
61         return boolean
62         --
63         -- Logic:
64         --   If "obj" is not null, retrieve the object's
65         --   TDO and check whether it is the account TDO.
66     is
67         use Object_Mgt, System;
68         -- Import "=" for "Object_Mgt.TDO_AD" and
69         -- "System.untyped_word".
70     begin
71         return obj /= System.null_word and then
72             Object_Mgt.Retrieve_TDO(obj) = account_TDO;
73     end Is_account;

```

**PRELIMINARY**

```

74
75
76 function Create_account(
77     starting_balance:
78         Long_Integer_Defs.long_integer :=
79         Long_Integer_Defs.zero)
80     return account_AD
81     --
82     -- Logic:
83     -- 1. Checks starting balance.
84     -- 2. Allocates an account.
85     -- 3. Initialize balance field,
86     -- 4. Remove rep rights on the returned AD.
87 is
88     account: account_AD;
89     account_rep: account_rep_AD;
90     FOR account_rep USE AT account'address;
91     account_untyped: System.untyped_word;
92     FOR account_untyped USE AT account'address;
93     -- One word viewed with three Ada types.
94 begin
95     if starting_balance < Long_Integer_Defs.zero then
96         RAISE insufficient_balance;
97     else
98
99         account_untyped := Object_Mgt.Allocate(
100             size => Object_Mgt.object_size(
101                 (account_rep_object' size + 31)/32),
102             -- Expression computes number of words
103             -- required to hold the number of bits
104             -- in an account.
105             tdo => account_TDO);
106
107         account_rep.all := account_rep_object'(
108             balance => starting_balance);
109
110         account_untyped := Access_Mgt.Remove(
111             AD => account_untyped,
112             rights => Object_Mgt.read_write_rights);
113         RETURN account;
114
115     end if;
116 end Create_account;
117
118
119 function Create_stored_account(
120     starting_balance:
121         Long_Integer_Defs.long_integer :=
122         Long_Integer_Defs.zero;
123     master: System_Defs.text;
124     authority:
125         Authority_List_Mgt.authority_list_AD := null)
126     return account_AD
127     --
128     -- Logic:
129     -- This call is not supported by this implementation.
130     --
131 is
132 begin
133     RAISE System_Exceptions.operation_not_supported;
134     RETURN null;
135
136 end Create_stored_account;
137
138
139 function Get_balance(
140     account: account_AD)
141     return Long_Integer_Defs.long_integer
142     --
143     -- Logic:
144     -- Amplifies read rights on "account" and
145     -- returns the balance field.
146 is
147     account_rep: account_rep_AD;
148     FOR account_rep USE AT account'address;
149     account_untyped: System.untyped_word;
150     FOR account_untyped USE AT account'address;

```

```

151 begin
152   account_untyped := Access_Mgt.Amplify(
153     AD    => account_untyped,
154     rights => Object_Mgt.read_rights,
155     tdo   => account_TDO);
156   return account_rep.balance;
157 end Get_balance;
158
159
160 function Change_balance(
161   account: account_AD;
162   amount: Long_Integer_Defs.long_integer)
163 return Long_Integer_Defs.long_integer
164 --
165 -- Logic:
166 -- 1. Imports rep rights on account if account
167 --    has change rights.
168 -- 2. Adds "amount" to the existing balance to
169 --    compute the prospective new balance.
170 --    "amount" can be positive (a deposit),
171 --    negative (a withdrawal), or zero.
172 -- 3. If new balance would be negative, raises
173 --    "insufficient_balance" and does not change
174 --    the balance.
175 -- 4. If new balance would be positive, then
176 --    stores the new balance and also returns it.
177 -- 5. Makes the update an atomic operation. If anything
178 --    goes wrong the update is rolled back.
179 is
180   account_rep: account_rep_AD;
181   FOR account_rep USE AT account'address;
182   account_untyped: System.untyped_word;
183   FOR account_untyped USE AT account'address;
184   new_balance: Long_Integer_Defs.long_integer;
185   -- Holds the new balance until a decision is
186   -- made whether to store it in the account.
187   old_balance: Long_Integer_Defs.long_integer;
188   -- Holds the old balance in case the operation
189   -- has to be rolled back.
190 begin
191   account_untyped := Access_Mgt.Import (
192     AD    => account_untyped,
193     rights => change_rights,
194     tdo   => account_TDO);
195
196   new_balance := account_rep.balance + amount;
197
198   if new_balance < Long_Integer_Defs.zero then
199     RAISE insufficient_balance;
200
201   else
202     begin
203       old_balance := account_rep.balance;
204       account_rep.balance := new_balance;
205       RETURN new_balance;
206     exception
207       -- An exception in this inner block means
208       -- that something has gone wrong with the
209       -- update. The old balance is restored.
210       when others =>
211         account_rep.balance := old_balance;
212       RAISE;
213     end;
214
215   end if;
216 end Change_balance;
217
218
219 procedure Transfer(
220   source_account: account_AD;
221   dest_account:   account_AD;
222   amount:         Long_Integer_Defs.long_integer)
223 --
224 -- Logic:
225 -- 1. Imports rep rights on both accounts if
226 --    they have change rights.
227 -- 2. Compute the prospective new balances,

```

```

228      -- by subtracting "amount" from the source
229      -- account's balance and adding it to the
230      -- destination account's balance.
231      -- "amount" can be positive, negative,
232      -- or zero.
233      -- 3. If either new balance would be negative,
234      -- raises "insufficient_balance" and does
235      -- not change the balance.
236      -- 4. Assigns the new balances. If an
237      -- exception occurs between assigning the
238      -- new source balance and the new destination
239      -- balance, a handler rolls back the source
240      -- balance to its old value, preserving
241      -- atomicity.
242  is
243      source_rep: account_rep_AD;
244      FOR source_rep USE AT source_account'address;
245      source_untyped: System.untyped word;
246      FOR source_untyped USE AT source_account'address;
247      old_source_bal: Long_Integer_Defs.long_integer;
248      -- Used to remember the old source balance in case
249      -- it needs to be restored if an exception occurs.
250      new_source_bal: Long_Integer_Defs.long_integer;
251      -- Holds the new source balance until a decision is
252      -- made whether to store it in the account.
253
254      dest_rep: account_rep_AD;
255      FOR dest_rep USE AT dest_account'address;
256      dest_untyped: System.untyped word;
257      FOR dest_untyped USE AT dest_account'address;
258      old_dest_bal: Long_Integer_Defs.long_integer;
259      -- Used to remember the old destination balance in case
260      -- it needs to be restored if an exception occurs.
261      new_dest_bal: Long_Integer_Defs.long_integer;
262      -- Holds the new destination balance until a decision
263      -- is made whether to store it in the account.
264
265  begin
266      source_untyped := Access_Mgt.Import (
267          AD => source_untyped,
268          rights => change_rights,
269          tdo => account_TDO);
270      dest_untyped := Access_Mgt.Import (
271          AD => dest_untyped,
272          rights => change_rights,
273          tdo => account_TDO);
274
275      new_source_bal := source_rep.balance - amount;
276      new_dest_bal := dest_rep.balance + amount;
277
278      if new_source_bal < Long_Integer_Defs.zero
279      or else
280          new_dest_bal < Long_Integer_Defs.zero then
281          RAISE insufficient_balance;
282
283      else
284          old_source_bal := source_rep.balance;
285          old_dest_bal := dest_rep.balance;
286          -- Old balances are recorded here
287          -- in case the update will have to be
288          -- rolled back.
289          begin
290              source_rep.balance := new_source_bal;
291              dest_rep.balance := new_dest_bal;
292          exception
293              -- An exception in this inner block means
294              -- that something has gone wrong with
295              -- the update. Restore the old balances to make
296              -- this operation atomic, then
297              -- reraise the exception.
298              when others =>
299                  source_rep.balance := old_source_bal;
300                  dest_rep.balance := old_dest_bal;
301                  RAISE;
302
303      end;
304      RETURN;

```

```

305
306     end if;
307 end Transfer;
308
309
310 procedure Destroy_account(
311     account: account_AD)
312 --
313 -- Logic:
314 -- Imports rep rights on account if account
315 -- has destroy rights.
316
317 -- If account's balance is not zero, raises
318 -- "balance_not_zero".
319 --
320 -- Otherwise, destroys the account.
321 is
322     account_rep: account_rep AD;
323     FOR account_rep USE AT account'address;
324     account_untyped: System.untyped_word;
325     FOR account_untyped USE AT account'address;
326 begin
327     account_untyped := Access_Mgt.Import(
328         AD      => account_untyped,
329         rights => destroy_rights,
330         tdo    => account_TDO);
331
332     if account_rep.balance /= Long_Integer_Defs.zero then
333         RAISE balance_not_zero;
334
335     else
336         Object_Mgt.Deallocate(account_untyped);
337
338     end if;
339 end Destroy_account;
340
341
342 begin
343     declare
344         passive_store_impl: constant
345             Passive_Store_Mgt.PSM_attributes_AD := new
346             Passive_Store_Mgt.PSM_attributes_object;
347         passive_store_impl_untyped: System.untyped_word;
348         FOR passive_store_impl_untyped USE AT
349             passive_store_impl'address;
350     begin
351         Passive_Store_Mgt.Set_refuse_filters(
352             passive_store_impl);
353         Attribute_Mgt.Store_attribute_for_type(
354             tdo      => account_TDO,
355             attr_ID  => Passive_Store_Mgt.PSM_attributes_ID,
356             attr_impl => passive_store_impl_untyped);
357     end;
358 end Account_Mgt_Ex;

```

## X-A.7.8 Account\_Mgt\_Ex (Stored, Non-transaction-oriented) Package Body

Non-transaction-oriented implementation of the type manager for stored accounts.

```

1  with Access_Mgt,
2      Authority_List_Mgt,
3      Directory_Mgt,
4      Long_Integer_Defs,
5      Object_Mgt,
6      Passive_Store_Mgt,
7      System,
8      System_Defs;
9
10 package body Account_Mgt_Ex is
11   --
12   -- Logic:
13   -- This is an implementation of the
14   -- account manager with these characteristics:
15   --
16   -- * Operations are NOT guaranteed to be
17   -- transaction-oriented or atomic.
18   --
19   -- * An account should NOT be concurrently
20   -- used, not by concurrent jobs and not by
21   -- concurrent processes in the same job.
22   --
23   -- * The account TDO must already exist in
24   -- the distributed system's directory structure.
25   -- The "bind" pragma is used to bind to the
26   -- stored TDO.
27   --
28   -- * The multiple activation model is used.
29   --
30
31   use Long_Integer_Defs, -- Import long integer
32                          -- operators.
33       System;           -- Import ordinal operators.
34
35
36   type account_rep_object is
37     record
38       balance: Long_Integer_Defs.long_integer;
39       -- Current balance.
40     end record;
41
42   type account_rep_AD is access account_rep_object;
43   pragma access_kind(account_rep_AD, AD);
44   -- Private view of an account.
45
46
47   account_TDO: constant Object_Mgt.TDO_AD := null;
48   -- This is a constant AD but not really null; its
49   -- filled in with an AD retrieved by the linker.
50   pragma bind(account_TDO,
51              "account");
52   -- Bind to TDO for accounts.
53
54
55   function Is_account(
56     obj: System.untyped_word)
57     return boolean
58     --
59     -- Logic:
60     -- If "obj" is not null, retrieve the object's
61     -- TDO and check whether it is the account's TDO.
62   is
63     use Object_Mgt; -- Import "=" for type "TDO_AD".
64   begin
65     return obj /= System.null_word
66            and then
67            Object_Mgt.Retrieve_TDO(obj) = account_TDO;
68   end Is_account;
69
70
71   function Create_account(
72     starting_balance:
73       Long_Integer_Defs.long_integer :=

```

PRELIMINARY

```

74     Long_Integer_Defs.zero)
75 return account_AD
76 --
77 -- Logic:
78 -- 1. Check the initial balance.
79 --
80 -- 2. Allocate and initialize the account object.
81 --
82 -- 3. Remove rep rights for the exported AD.
83 -- The caller is responsible for storing
84 -- the AD and updating the object.
85 --
86 -- 4. Return the AD without rep rights.
87 is
88 account:         account_AD;
89 account_untyped: System.Untyped_word;
90 FOR account_untyped USE AT account'address;
91 -- Account with no rep rights, viewed with
92 -- either of two types.
93
94 account_rep:     account_rep_AD;
95 account_rep_untyped: System.untyped_word;
96 FOR account_rep_untyped USE AT
97     account_rep'address;
98 -- Account with rep rights, viewed with
99 -- either of two types.
100
101 begin
102 -- 1. Check the initial balance:
103 --
104 if starting_balance < Long_Integer_Defs.zero then
105     RAISE insufficient_balance;
106
107 else
108 -- 2. Allocate and initialize the account object:
109 --
110 account_rep_untyped := Object_Mgt.Allocate(
111     size => (account_rep_object'size + 31)/32,
112     tdo => account_TDO);
113 account_rep.all := account_rep_object'(
114     balance => starting_balance);
115
116 -- 3. Remove rep rights for the exported AD:
117 --
118 account_untyped := Access_Mgt.Remove(
119     AD => account_rep_untyped,
120     rights => Object_Mgt.read_write_rights);
121
122 -- 4. Return the account AD with no rep rights:
123 --
124 RETURN account;
125
126 end if;
127
128 end Create_account;
129
130
131
132 function Create_stored_account(
133     starting_balance:
134         Long_Integer_Defs.long_integer :=
135         Long_Integer_Defs.zero;
136     master:         System_Defs.text;
137     authority:
138         Authority_List_Mgt.authority_list_AD := null)
139 return account_AD
140 --
141 -- Logic:
142 -- 1. Check the initial balance.
143 --
144 -- 2. Allocate and initialize the account object.
145 --
146 -- 3. Remove rep rights for the exported and master
147 -- AD.
148 --
149 -- 4. Store the master AD.
150 -- Use "authority" as authority list to store the

```

```

151      --      account. If "authority" is null, the default
152      --      authority list of the target directory is used.
153      --      If there is none the caller's authority list in
154      --      the process globals is used.
155      --
156      --      5. Passivate the account object itself.
157      --
158      --      6. Return the AD without rep rights.
159  is
160      account:          account_AD;
161      account_untyped: System.untyped_word;
162      FOR account_untyped USE AT account'address;
163      -- Account with no rep rights, viewed with
164      -- either of two types.
165
166      account_rep:      account_rep_AD;
167      account_rep_untyped: System.untyped_word;
168      FOR account_rep_untyped USE AT
169      account_rep'address;
170      -- Account with rep rights, viewed with
171      -- either of two types.
172
173  begin
174      -- 1. Check the initial balance:
175      --
176      if starting_balance < Long_Integer_Defs.zero then
177          RAISE insufficient_balance;
178
179      else
180          -- 2. Allocate and initialize the account object:
181          --
182          account_rep_untyped := Object_Mgt.Allocate(
183              size => (account_rep_object'size + 31)/32,
184              tdo => account_TDO);
185          account_rep.all := account_rep_object'(
186              balance => starting_balance);
187
188          -- 3. Remove rep rights for the exported and
189          --      master AD:
190          --
191          account_untyped := Access_Mgt.Remove(
192              AD => account_rep_untyped,
193              rights => Object_Mgt.read_write_rights);
194
195          -- 4. Store the master AD:
196          --
197          Directory_Mgt.Store(
198              name => master,
199              object => account_untyped,
200              aut => authority);
201
202          -- 5. Passivate the account object itself:
203          --
204          Passive_Store_Mgt.Update(account_rep_untyped);
205
206          -- 6. Return the account AD with no rep rights:
207          --
208          RETURN account;
209
210      end if;
211  end Create_stored_account;
212
213
214  function Get_balance(
215      account: account_AD)
216      return Long_Integer_Defs.long_integer
217      --
218      -- Logic:
219      -- 1. Amplify rep rights on the account AD.
220      --
221      -- 2. Return the balance.
222  is
223      account_rep: account_rep_AD;
224      FOR account_rep USE AT account'address;
225      account_untyped: System.untyped_word;
226      FOR account_untyped USE AT account'address;
227

```

**PRELIMINARY**

```

228 begin
229     account_untyped := Access_Mgt.Amplify(
230         AD      => account_untyped,
231         rights => Object_Mgt.read_write_rights,
232         tdo     => account_TDO);
233
234     return account_rep.balance;
235 end Get_balance;
236
237
238 function Change_balance(
239     account: account_AD;
240     amount:  Long_Integer_Defs.long_integer)
241 return Long_Integer_Defs.long_integer
242 --
243 -- Logic:
244 -- 1. Import the account AD, checking for
245 --    change rights and adding rep rights.
246 --
247 -- 2. If the new balance would be negative,
248 --    then exit with an exception.
249 --
250 -- 3. Otherwise, change the balance, update
251 --    the passive version, and return the
252 --    new balance.
253 is
254     account_rep: account_rep_AD;
255     FOR account_rep USE AT account'address;
256     account_untyped: System.untyped_word;
257     FOR account_untyped USE AT account'address;
258 begin
259     account_untyped := Access_Mgt.Import(
260         AD      => account_untyped,
261         rights => change_rights,
262         tdo     => account_TDO);
263     if account_rep.balance + amount < zero then
264         RAISE insufficient_balance;
265
266     else
267         account_rep.balance :=
268             account_rep.balance + amount;
269         Passive_Store_Mgt.Update(account_untyped);
270         RETURN account_rep.balance;
271
272     end if;
273 end Change_balance;
274
275
276 procedure Transfer(
277     source_account: account_AD;
278     dest_account:   account_AD;
279     amount: Long_Integer_Defs.long_integer)
280 --
281 -- Logic:
282 -- 1. Import the account ADs, checking for
283 --    change rights and adding rep rights.
284 --
285 -- 2. If either new balance would be negative,
286 --    then exit with an exception.
287 --
288 -- 3. Otherwise, change the balances, update
289 --    the passive versions, and return.
290 --
291 -- Warning:
292 -- This implementation is not atomic; a change
293 -- may be made in the source account but not
294 -- in the destination account if an exception,
295 -- system crash, or other error intervenes.
296 is
297     source_rep: account_rep_AD;
298     FOR source_rep USE AT source_account'address;
299     source_untyped: System.untyped_word;
300     FOR source_untyped USE AT source_account'address;
301
302     dest_rep: account_rep_AD;
303     FOR dest_rep USE AT dest_account'address;
304     dest_untyped: System.untyped_word;

```

```

305     FOR dest_untyped USE at dest_account'address;
306     begin
307         source_untyped := Access_Mgt.Import(
308             AD => source_untyped,
309             rights => change_rights,
310             tdo => account_TDO);
311         dest_untyped := Access_Mgt.Import(
312             AD => dest_untyped,
313             rights => change_rights,
314             tdo => account_TDO);
315
316         if source_rep.balance - amount < zero
317         or else
318             dest_rep.balance + amount < zero
319         then
320             RAISE insufficient_balance;
321
322         else
323             source_rep.balance :=
324                 source_rep.balance - amount;
325             dest_rep.balance :=
326                 dest_rep.balance + amount;
327             Passive_Store_Mgt.Update(source_untyped);
328             Passive_Store_Mgt.Update(dest_untyped);
329             RETURN;
330
331         end if;
332     end Transfer;
333
334     procedure Destroy_account(
335         account: account_AD)
336     --
337     -- Logic:
338     -- 1. Import the account AD, checking for
339     --    destroy rights and amplifying rep rights.
340     --
341     -- 2. Check that the account's balance is zero.
342     --    If it isn't, raise an exception. If it
343     --    is, execute the remaining steps.
344     --
345     -- 3. Destroy the account's passive version.
346     --
347     -- 4. Get the name of the account's master
348     --    directory entry (if any). Delete that
349     --    directory entry. Note that other
350     --    entries and even a master AD may remain
351     --    for the account.
352     --
353     -- 5. Deallocate the account's active version.
354     is
355     account_rep: account_rep_AD;
356     FOR account_rep USE AT account'address;
357     account_untyped: System.untyped_word;
358     FOR account_untyped USE AT account'address;
359
360     path_length: integer := 60;
361     -- Initial text length for name assigned
362     -- by "Directory_Mgt.Get_name". If
363     -- insufficient, then the value is
364     -- increased and the operation is
365     -- repeated.
366
367     begin
368         account_untyped := Access_Mgt.Import(
369             AD => account_untyped,
370             rights => destroy_rights,
371             tdo => account_TDO);
372
373         if account_rep.balance /=
374             Long_Integer_Defs.zero then
375             RAISE balance_not_zero;
376
377         else
378             Passive_Store_Mgt.Destroy(account_untyped);
379
380         loop
381             declare

```

PRELIMINARY

```
382     path_text: System_Defs.text(path_length);
383 begin
384     Directory_Mgt.Get_name(
385         obj => account_untyped,
386         name => path_text); -- out.
387     if path_text.length >
388         path_text.max_length then
389         -- Text was lost.  Retry:
390         path_length := path_text.length;
391     else
392         Directory_Mgt.Delete(path_text);
393         EXIT;
394
395     end if;
396     exception
397     when Directory_Mgt.no_name =>
398         EXIT;
399
400     end;
401     end loop;
402
403     Object_Mgt.Deallocate(account_untyped);
404     end if;
405     end Destroy_account;
406
407 end Account_Mgt_Ex;
```

## X-A.7.9 Account\_Mgt\_Ex (Stored, Transaction-oriented) Package Body

Transaction-oriented implementation of the type manager for stored accounts.

```

1  with Access_Mgt,
2      Authority_List_Mgt,
3      Directory_Mgt,
4      Long_Integer_Defs,
5      Object_Mgt,
6      Passive_Store_Mgt,
7      System,
8      System_Defs,
9      System_Exceptions,
10     Transaction_Mgt;
11
12 package body Account_Mgt_Ex is
13     --
14     -- Logic:
15     --   This is an implementation of the
16     --   account manager with these characteristics:
17     --
18     --   * All operations are transaction-oriented,
19     --   participating in any default transaction
20     --   or else creating a transaction for the
21     --   duration of the operation.
22     --
23     --   * An account should not be concurrently
24     --   used by more than one process in a single
25     --   job, unless an external locking protocol
26     --   is used.
27     --
28     --   * The account TDO must already exist in
29     --   the distributed system's directory structure.
30     --   The "bind" pragma is used to bind to the
31     --   stored TDO.
32     --
33     --   * The multiple activation model is used.
34
35
36     use Long_Integer_Defs, -- Import "long_integer", "zero",
37                          -- arithmetic and relational operators.
38         System,           -- Import ordinal operators.
39         Transaction_Mgt;  -- Import transaction calls.
40
41
42     type account_rep_object is
43         record
44             balance: Long_Integer_Defs.long_integer;
45             -- Current balance.
46         end record;
47
48     type account_rep AD is access account_rep_object;
49     pragma access_kind(account_rep_AD, AD);
50     -- Private view of an account.
51
52     account_TDO: constant Object_Mgt.TDO_AD := null;
53     -- This is a constant AD but not really null; its
54     -- filled in with an AD retrieved by the linker.
55     pragma bind(account_TDO,
56                "account");
57     -- Bind to TDO for accounts.
58
59
60     function Is_account (
61         obj: System.untyped_word)
62         return boolean
63     --
64     -- Logic:
65     --   If "obj" is not null, retrieve the object's
66     --   TDO and check whether it is the account's TDO.
67     is
68     use Object_Mgt; -- Import "=" for type "TDO_AD".
69     begin
70         return obj /= System.null_word
71             and then
72             Object_Mgt.Retrieve_TDO(obj) = account_TDO;
73     end Is_account;

```

PRELIMINARY

```

74
75
76 function Create_account(
77     starting_balance:
78     Long_Integer_Defs.long_integer :=
79     Long_Integer_Defs.zero)
80     return account_AD
81     --
82     -- Logic:
83     -- 1. Check the initial balance.
84     --
85     -- 2. Allocate and initialize the account object.
86     --
87     -- 3. Return AD with no rep rights.
88     --
89     -- 4. If any exception occurs, abort any local
90     --     transaction, deallocate the account,
91     --     and reraise the exception.
92     --
93     is
94     account:         account_AD;
95     account_untyped: System.untyped_word;
96     FOR account_untyped USE AT account'address;
97     -- Account with no rep rights, viewed with
98     -- either of two types.
99
100    account_rep:     account_rep_AD;
101    account_rep_untyped: System.untyped_word;
102    FOR account_rep_untyped USE AT
103        account_rep'address;
104    -- Account with rep rights, viewed with
105    -- either of two types.
106
107    begin
108    -- 1. Check the initial balance:
109    --
110    if starting_balance < Long_Integer_Defs.zero then
111        RAISE insufficient_balance;
112    else
113    -- 2. Allocate and initialize the account object:
114    --
115    account_rep_untyped := Object_Mgt.Allocate(
116        size => (account_rep_object'size + 31)/32,
117        tdo => account_TDO);
118    begin
119    -- Inside this block it is guaranteed
120    -- that the object has been allocated.
121    account_rep.all := account_rep_object'(
122        balance => starting_balance);
123
124    -- 3. Remove rep rights for the exported AD:
125    --
126    account_untyped := Access_Mgt.Remove(
127        AD => account_rep_untyped,
128        rights => Object_Mgt.read_write_rights);
129
130    exception
131    -- 4. If any exception occurs, abort any local
132    --     transaction, deallocate the account,
133    --     and reraise the exception:
134    --
135    when others =>
136        Object_Mgt.Deallocate(account_untyped);
137        RAISE;
138
139    end;
140
141    RETURN account;
142
143    end if;
144 end Create_account;
145
146 function Create_stored_account(
147     starting_balance:
148     Long_Integer_Defs.long_integer :=
149     Long_Integer_Defs.zero;
150

```

```

151     master: System_Defs.text;
152     authority:
153         Authority_List_Mgt.authority_list_AD := null)
154     return account_AD
155     --
156     -- Logic:
157     -- 1. Check the initial balance.
158     --
159     -- 2. Allocate and initialize the account object.
160     --
161     -- 3. Remove rep rights for the exported and
162     --     master AD.
163     --
164     -- 4. Start a local transaction if there is
165     --     not a transaction on the stack.
166     --
167     -- 5. Store the master AD.
168     --     Use "authority" as authority list to store the
169     --     account. If no authority list has be explicitly
170     --     specified the default authority of the target
171     --     directory is used. If there is none the the caller's
172     --     authority list in the process globals is used instead.
173     --
174     -- 6. Passivate the account object itself.
175     --
176     -- 7. Commit any local transaction.
177     --
178     -- 8. If any exception occurs, abort any local
179     --     transaction, deallocate the account,
180     --     and reraise the exception.
181     is
182     account:         account_AD;
183     account_untyped: System.untyped_word;
184     FOR account_untyped USE AT account'address;
185     -- Account with no rep rights, viewed with
186     -- either of two types.
187
188     account_rep:     account_rep_AD;
189     account_rep_untyped: System.untyped_word;
190     FOR account_rep_untyped USE AT
191         account_rep'address;
192     -- Account with rep rights, viewed with
193     -- either of two types.
194
195     trans: boolean := false;
196     -- True if a local transaction is started.
197     begin
198     -- 1. Check the initial balance:
199     --
200     if starting_balance < Long_Integer_Defs.zero then
201         RAISE insufficient_balance;
202     else
203     -- 2. Allocate and initialize the account object:
204     --
205     account_rep_untyped := Object_Mgt.Allocate(
206         size => (account_rep_object'size + 31)/32,
207         tdo => account_TDO);
208     account_rep.all := account_rep_object'(
209         balance => starting_balance);
210
211     -- 3. Remove rep rights for the exported and
212     --     master AD:
213     --
214     account_untyped := Access_Mgt.Remove(
215         AD => account_rep_untyped,
216         rights => Object_Mgt.read_write_rights);
217
218     -- 4. Start a local transaction if there is not
219     --     a transaction on the stack:
220     --
221     --
222     if Transaction_Mgt.Get_default_transaction =
223         null then
224         Transaction_Mgt.Start_transaction;
225         trans := true;
226     end if;
227     begin

```

```

228      -- 5. Store the master AD:
229      --
230      Directory_Mgt.Store(
231          name    => master,
232          object  => account_untyped,
233          aut     => authority);
234
235      -- 6. Passivate the account object itself:
236      --
237      Passive_Store_Mgt.Update(account_rep_untyped);
238
239      -- 7. Commit any local transaction:
240      --
241      if trans then
242          Transaction_Mgt.Commit_transaction;
243      end if;
244      exception
245      -- 8. If any exception occurs, abort any local
246      --    transaction, deallocate the account,
247      --    and reraise the exception:
248      --
249      when others =>
250          if trans then
251              Transaction_Mgt.Abort_transaction;
252          end if;
253          Object_Mgt.Deallocate(account_untyped);
254          RAISE;
255
256      end;
257      RETURN account;
258
259  end if;
260  end Create_stored_account;
261
262  function Get_balance(
263      account: account_AD)
264      return Long_Integer_Defs.long_integer
265  --
266  -- Logic:
267  -- 1. Amplify rep rights on the account AD.
268  --
269  -- 2. Loop (in case of retry due to a transaction
270  --    timestamp conflict).
271  --
272  -- 3. If there is no default transaction,
273  --    start a local transaction and flag that
274  --    it is started.
275  --
276  -- 4. Reserve the account object to read-lock
277  --    the passive version and ensure a clean
278  --    and *current* active version.
279  --
280  -- 5. Commit any local transaction, releasing
281  --    the lock.
282  --
283  -- 6. Return the balance from the certainly
284  --    clean active version.
285  --
286  -- 7. If there is a transaction timestamp
287  --    conflict, and if a local transaction was
288  --    started, then abort that transaction, loop
289  --    back, start a fresh transaction, and try
290  --    again.
291  --
292  -- 8. If there is any other exception, then
293  --    abort any local transaction and reraise
294  --    the exception.
295  --
296  is
297      account_rep: account_rep_AD;
298      FOR account_rep USE AT account'address;
299      account_untyped: System.untyped_word;
300      FOR account_untyped USE AT account'address;
301
302      trans: boolean := false;
303      -- True if a local transaction is started.
304  begin

```

PRELIMINARY

```

305 account_untyped := Access_Mgt.Amplify(
306     AD => account_untyped,
307     rights => Object_Mgt.read_write_rights,
308     tdo => account_TDO);
309
310 loop
311     if Transaction_Mgt.Get_default_transaction =
312         null then
313         Transaction_Mgt.Start_transaction;
314         trans := true;
315     end if;
316     begin
317         Passive_Store_Mgt.Reserve(
318             obj => account_untyped,
319             read => true);
320         if trans then
321             Transaction_Mgt.Commit_transaction;
322         end if;
323         RETURN account_rep.balance;
324
325     exception
326     when System_Exceptions.
327         transaction_timestamp_conflict =>
328         if trans then
329             Transaction_Mgt.Abort_transaction;
330         else
331             RAISE;
332         end if;
333     when others =>
334         if trans then
335             Transaction_Mgt.Abort_transaction;
336         end if;
337         RAISE;
338     end;
339 end loop;
340 end Get_balance;
341
342
343
344
345 function Change_balance(
346     account: account_AD;
347     amount: Long_Integer_Defs.long_integer)
348 return Long_Integer_Defs.long_integer
349 --
350 -- Logic:
351 -- 1. Import the account AD, checking for
352 --    change rights and adding rep rights.
353 --
354 -- 2. Loop (in case of retry due to a transaction
355 --    timestamp conflict).
356 --
357 -- 3. If there is no default transaction, then
358 --    start a local transaction and flag that it
359 --    is started.
360 --
361 -- 4. Reserve the account object to write-lock
362 --    the passive version and ensure a clean
363 --    and *current* active version.
364 --
365 -- 5. If the new balance would be negative, abort
366 --    the transaction and exit with an exception.
367 --
368 -- 6. Otherwise, change the balance, update the
369 --    passive version, and commit any local
370 --    transaction, releasing the lock.
371 --
372 -- 7. If there is a transaction timestamp conflict,
373 --    and if a local transaction was started, then
374 --    abort that transaction, loop back, start a
375 --    fresh transaction, and try again.
376 --
377 -- 8. If there is any other exception, then
378 --    abort any local transaction and reraise
379 --    the exception.
380 --
381 -- Notes:

```

```

382 -- It might appear that instead of reserving the
383 -- object, the implementation could simply compute
384 -- the new balance, do the update, and reset the
385 -- active version and retry in the infrequent
386 -- case that "outdated object version" in
387 -- "Passive_Store_Mgt" is raised. However, such
388 -- an implementation would base the checking for
389 -- an insufficient balance on a possibly obsolete
390 -- value, which is unacceptable.
391 is
392   account_rep: account_rep_AD;
393   FOR account_rep USE AT account'address;
394   account_untyped: System.untyped_word;
395   FOR account_untyped USE AT account'address;
396
397   trans: boolean := false;
398   -- True if a local transaction is started.
399   begin
400     account_untyped := Access_Mgt.Import(
401       AD => account_untyped,
402       rights => change_rights,
403       tdo => account_TDO);
404
405     loop
406       if Transaction_Mgt.Get_default_transaction =
407         null then
408         Transaction_Mgt.Start_transaction;
409         trans := true;
410       end if;
411       begin
412         Passive_Store_Mgt.Reserve(account_untyped);
413         if account_rep.balance + amount < zero then
414           RAISE insufficient_balance;
415         else
416           account_rep.balance :=
417             account_rep.balance + amount;
418           Passive_Store_Mgt.Update(account_untyped);
419           if trans then
420             Transaction_Mgt.Commit_transaction;
421           end if;
422           RETURN account_rep.balance;
423         end if;
424       exception
425       when System_Exceptions.
426         transaction_timestamp_conflict =>
427         if trans then
428           Transaction_Mgt.Abort_transaction;
429         else
430           RAISE;
431         end if;
432       when others =>
433         if trans then
434           Transaction_Mgt.Abort_transaction;
435         end if;
436         RAISE;
437       end;
438     end loop;
439   end Change_balance;
440
441 procedure Transfer(
442   source_account: account_AD;
443   dest_account: account_AD;
444   amount: Long_Integer_Defs.long_integer)
445 --
446 -- Logic:
447 -- 1. Import the account ADs, checking for
448 -- change rights and adding rep rights.
449 --
450 -- 2. Loop (in case of retry due to a transaction
451 -- timestamp conflict).
452 --
453 -- 3. If there is no default transaction, then
454 -- start a local transaction and flag that it

```

PRELIMINARY

```

459      --      is started.
460      --
461      --      4. Reserve the account objects to write-lock
462      --      the passive versions and ensure a clean
463      --      and *current* active version.
464      --
465      --      5. If either new balance would be negative, abort
466      --      the transaction and exit with an exception.
467      --
468      --      6. Otherwise, change the balances, update the
469      --      passive versions, and commit any local
470      --      transaction, releasing the lock.
471      --
472      --      7. If there is a transaction timestamp conflict,
473      --      and if a local transaction was started, then
474      --      abort that transaction, loop back, start a
475      --      fresh transaction, and try again.
476      --
477      --      8. If there is any other exception, then
478      --      abort any local transaction and reraise
479      --      the exception.
480  is
481      source_rep: account_rep_AD;
482      FOR source_rep USE AT source_account'address;
483      source_untyped: System.untyped_word;
484      FOR source_untyped USE AT source_account'address;
485
486      dest_rep: account_rep_AD;
487      FOR dest_rep USE AT dest_account'address;
488      dest_untyped: System.untyped_word;
489      FOR dest_untyped USE at dest_account'address;
490
491      trans: boolean := false;
492      -- True if a local transaction is started.
493  begin
494
495      source_untyped := Access_Mgt.Import(
496          AD      => source_untyped,
497          rights => change_rights,
498          tdo     => account_TDO);
499      dest_untyped := Access_Mgt.Import(
500          AD      => dest_untyped,
501          rights => change_rights,
502          tdo     => account_TDO);
503
504      loop
505          if Transaction_Mgt.Get_default_transaction =
506              null then
507              Transaction_Mgt.Start_transaction;
508              trans := true;
509          end if;
510          begin
511              Passive_Store_Mgt.Reserve(source_untyped);
512              Passive_Store_Mgt.Reserve(dest_untyped);
513              if source_rep.balance - amount < zero
514                  or else
515                  dest_rep.balance + amount < zero
516              then
517                  RAISE insufficient_balance;
518              else
519                  source_rep.balance :=
520                      source_rep.balance - amount;
521                  dest_rep.balance :=
522                      dest_rep.balance + amount;
523                  Passive_Store_Mgt.Update(source_untyped);
524                  Passive_Store_Mgt.Update(dest_untyped);
525                  if trans then
526                      Transaction_Mgt.Commit_transaction;
527                  end if;
528                  RETURN;
529              end if;
530          exception
531          when System_Exceptions.
532              transaction_timestamp_conflict =>
533              if trans then
534                  if trans then
535
```

**PRELIMINARY**

```

536         Transaction_Mgt.Abort_transaction;
537     else
538         RAISE;
539
540     end if;
541     when others =>
542         if trans then
543             Transaction_Mgt.Abort_transaction;
544         end if;
545         RAISE;
546
547     end;
548     end loop;
549 end Transfer;
550
551
552 procedure Destroy_account (
553     account: account_AD)
554 --
555 -- Logic:
556 -- 1. Import the account AD, checking for
557 --    destroy rights and amplifying rep rights.
558 --
559 -- 2. Loop in case of retry due to timestamp
560 --    conflict.
561 --
562 -- 3. If there is no default transaction, then
563 --    start a local transaction and flag that it
564 --    is started.
565 --
566 -- 4. Reserve the account object to write-lock
567 --    the passive version and ensure a clean
568 --    and current active version.
569 --
570 -- 5. Check that the account's balance is zero.
571 --    If it isn't, raise an exception. The
572 --    block's exception handler will abort
573 --    any local transaction.
574 --
575 -- 6. Destroy the account's passive version.
576 --
577 -- 7. Get the name of the account's master
578 --    directory entry (if any). Delete that
579 --    directory entry. Note that other
580 --    entries and even a master AD may remain
581 --    for the account.
582 --
583 -- 8. If there is a transaction timestamp
584 --    conflict, and if a local transaction
585 --    was started, then abort that transaction,
586 --    loop back, start a fresh transaction,
587 --    and try again.
588 --
589 -- 9. If any other exception occurs, abort
590 --    any local transaction and reraise the
591 --    exception.
592 --
593 -- 10. Deallocate the account's active version.
594 is
595     account_rep: account_rep_AD;
596     FOR account_rep USE AT account'address;
597     account_untyped: System.untyped_word;
598     FOR account_untyped USE AT account'address;
599
600     trans: boolean := false;
601     -- True if a local transaction is started.
602 begin
603     account_untyped := Access_Mgt.Import (
604         AD => account_untyped,
605         rights => destroy_rights,
606         tdo => account_TDO);
607     loop
608         if Transaction_Mgt.Get_default_transaction =
609             null then
610             Transaction_Mgt.Start_transaction;
611             trans := true;
612         end if;

```

```

613 declare
614     path_length: integer := 60;
615     -- Initial text length for name assigned
616     -- by "Directory_Mgt.Get_name". If
617     -- insufficient, then the value is
618     -- increased and the operation is
619     -- repeated.
620 begin
621     Passive_Store_Mgt.Reserve(account_untyped);
622     if account_rep.balance /=
623         Long_Integer_Defs.zero then
624         RAISE balance_not_zero;
625
626     end if;
627     Passive_Store_Mgt.Destroy(account_untyped);
628
629     loop
630     declare
631         path_text: System_Defs.text(path_length);
632     begin
633         Directory_Mgt.Get_name(
634             obj => account_untyped,
635             name => path_text); -- out.
636         if path_text.length >
637             path_text.max_length then
638             -- Text was lost. Retry:
639             path_length := path_text.length;
640         else
641             Directory_Mgt.Delete(path_text);
642             EXIT;
643
644         end if;
645     exception
646     when Directory_Mgt.no_name =>
647         EXIT;
648
649     end;
650     end loop;
651 exception
652 when System_Exceptions.
653     transaction_timestamp_conflict =>
654     if trans then
655         Abort_transaction;
656     else
657         RAISE;
658
659     end if;
660
661     when others =>
662     if trans then
663         Abort_transaction;
664     end if;
665     RAISE;
666
667     end;
668     EXIT;
669     end loop;
670     Object_Mgt.Deallocate(account_untyped);
671 end Destroy_account;
672
673
674 end Account_Mgt_Ex;

```

## X-A.7.10 Stored\_Account\_TDO\_Init\_Ex Procedure

Initialization procedure for stored account type managers.

```

1  with Account_Type_Name_Ex,    -- Example package.
2     Attribute_Mgt,
3     Authority_List_Mgt,
4     Directory_Mgt,
5     Identification_Mgt,
6     Object_Mgt,
7     Passive_Store_Mgt,
8     Refuse_reset_active_version_Ex, -- Example package
9     System,
10    System_Defs,
11    System_Exceptions,
12    Text_Mgt,
13    Transaction_Mgt,
14    Type_Name_Attribute_Ex,    -- Example package.
15    User_Mgt,
16    Unchecked_conversion;
17
18 procedure Stored_Account_TDO_Init_Ex
19 --
20 -- Logic:
21 --   Initialize TDO for accounts and place it in
22 --   the passive store for use by instances of
23 --   "Stored_Account_Mgt_Ex" at different nodes.
24 --
25 --   The account TDO has the OS passive store
26 --   attribute and the (example) type name attribute.
27 --
28 --   Resetting an account's active version or
29 --   copying accounts are not allowed outside the
30 --   type manager. Other passive store requests
31 --   are allowed.
32 -- History:
33 --   ??-??-????: Martin Buchanan, Initial version.
34 --   12-01-1987: Tobias Haas, Removed 'Refuse_reset_active_version'
35 --               procedure and placed in separate package.
36 --   04-20-1988: Tobias Haas, Added extractor comments, bstex*.ex
37 --   05-06-1988: Tobias Haas, Modified extractor comments, bstex*.ex
38 --   05-20-1988: Tobias Haas, Added handler for Directory_Mgt.
39 --               entry_exists
40 is
41   use Transaction_Mgt;
42   -- Import transaction operators.
43
44   account_name: constant string :=
45     "account";
46   -- pathname of account tdo.
47
48   account_text: System_Defs.text(account_name'length);
49   -- Pathname is placed in this text before calling
50   -- "Directory_Mgt.Store".
51
52   account_TDO: Object_Mgt.TDO_AD;
53   -- TDO for accounts.
54
55   passive_store_impl:
56     Passive_Store_Mgt.PSM_attributes_AD;
57   -- Implementation of passive store attribute
58   -- for accounts.
59
60   type_name_impl: System.untyped_word;
61   -- Implementation of type name attribute
62   -- for accounts.
63
64   owner_only: User_Mgt.protection_set(1);
65   -- Protection set that includes only one ID, namely
66   -- the type manager's owner.
67
68   authority: Authority_List_Mgt.authority_list_AD;
69   -- Authority list that contains only one ID, namely
70   -- the type manager's owner.
71
72   trans: boolean := false;
73   -- Set if local transaction is started.

```

```

74
75
76 function Untyped_from_PSM_attributes is
77     new Unchecked_conversion(
78         source => Passive_Store_Mgt.PSM_attributes_AD,
79         target => System.untyped_word);
80
81
82 function Untyped_from_TDO is
83     new Unchecked_conversion(
84         source => Object_Mgt.TDO_AD,
85         target => System.untyped_word);
86
87
88 begin
89     Text_Mgt.Set(account_text, account_name);
90
91     account_TDO := Object_Mgt.Create_TDO;
92
93     passive_store_impl := new
94         Passive_Store_Mgt.PSM_attributes_object;
95
96     passive_store_impl.reset :=
97         Refuse_reset_active_version_Ex.
98         Refuse_reset_active_version' subprogram_value;
99
100    passive_store_impl.copy_permitted := false;
101
102    Attribute_Mgt.Store_attribute_for_type(
103        tdo => account_TDO,
104        attr_ID => Passive_Store_Mgt.PSM_attributes_ID,
105        attr_impl => Untyped_from_PSM_attributes(
106            passive_store_impl));
107    type_name_impl := Account_Type_Name_Ex'package_value;
108
109    Attribute_Mgt.Store_attribute_for_type(
110        tdo => account_TDO,
111        attr_ID => Type_Name_Attribute_Ex.
112            Get_type_name_attr_ID,
113        attr_impl => type_name_impl);
114
115    owner_only.length := 1;
116    owner_only.entries(1).rights := User_Mgt.access_rights'(
117        true, true, true);
118    owner_only.entries(1).id := Identification_Mgt.Get_user_id;
119
120    authority := Authority_List_Mgt.Create_authority(owner_only);
121
122    if Transaction_Mgt.Get_default_transaction =
123        null then
124        Transaction_Mgt.Start_transaction;
125        trans := true;
126    end if;
127
128    begin
129        Directory_Mgt.Store(
130            name => account_text,
131            object => Untyped_from_TDO(account_TDO),
132            aut => authority);
133        Passive_Store_Mgt.Request_update(
134            Untyped_from_TDO(account_TDO));
135        Passive_Store_Mgt.Request_update(
136            Untyped_from_PSM_attributes(
137                passive_store_impl));
138        Passive_Store_Mgt.Request_update(
139            type_name_impl);
140
141        if trans then
142            Transaction_Mgt.Commit_transaction;
143        end if;
144    exception
145        when Directory_Mgt.entry_exists =>
146            if trans then
147                Transaction_Mgt.Abort_transaction;
148            end if;
149
150    when others =>

```

## PRELIMINARY

```
151     if trans then
152         Transaction_Mgt.Abort_transaction;
153     end if;
154     RAISE;
155
156     end;
157
158 end Stored_Account_TDO_Init_Ex;
```

## X-A.7.11 Account\_Type\_Name\_Ex Package Specification

Type name attribute implementation for stored account type managers.

```
1  with System,
2     Type_Name_Attribute_Ex;
3
4  package Account_Type_Name_Ex is
5     pragma package_value (Type_Name_Attribute_Ex.Ops);
6     --
7     -- Function:
8     --   Defines the type name attribute for accounts.
9     --
10    --   A type that supports this attribute has a
11    --   printable name. For example, a directory
12    --   listing utility could use this attribute to
13    --   print the types of the objects in a
14    --   directory.
15
16
17    function Type_name(
18        obj: System.untyped_word)
19        return string;
20        -- Name of the "account" object type.
21    --
22    -- Function:
23    --   Returns the type name for account objects.
24
25
26    pragma external;
27
28 end Account_Type_Name_Ex;
```

## X-A.7.12 Account\_Type\_Name\_Ex Package Body

Type name attribute implementation for stored account type managers.

```
1 with System;
2
3 package body Account_Type_Name_Ex is
4
5
6   function Type_name(
7     obj: System.untyped_word)
8     return string
9   is
10  begin
11    return "account";
12  end Type_name;
13
14
15 end Account_Type_Name_Ex;
```

## X-A.7.13 Type\_Name\_Attr\_Ex Package Specification

Type name attribute package type.

```
1  with Attribute_Mgt,
2      System;
3
4  package Type_Name_Attribute_Ex is
5      --
6      -- Function:
7      --   Define an attribute that returns a type's name.
8      --
9      --   A type that supports the *type name* attribute has a
10     --   printable name. For example, a directory listing utility
11     --   could use the attribute to print the types of the objects
12     --   in a directory.
13
14     function Get_type_name_attr_ID
15         return Attribute_Mgt.attribute_ID_AD;
16         -- Type name attribute ID, with type rights.
17     --
18     -- Function:
19     --   Returns the type name attribute's attribute ID.
20
21
22
23     package Ops is
24         pragma package_type("tynamattr");
25         --
26         -- Function:
27         --   Provide "Type_name" attribute call.
28
29
30         function Type_name(
31             obj: System.untyped_word)
32             -- Any object that supports
33             -- the type name attribute.
34             return string; -- Name of the object's type.
35         pragma interface(value, Type_name);
36         --
37         -- Function:
38         --   Returns a printable name for an object's type.
39
40
41     end Ops;
42
43
44
45     pragma external;
46
47 end Type_Name_Attribute_Ex;
```

## X-A.7.14 Type\_Name\_Attr\_Ex Package Body

Type name attribute package type.

```

1  with Attribute_Mgt,
2     System_Defs;
3
4  package body Type_Name_Attribute_Ex is
5
6
7     type_name_attr_ID: constant
8         Attribute_Mgt.attribute_ID_AD := null;
9     pragma bind(type_name_attr_ID,
10                "typnamattr");
11     -- Attribute ID is retrieved at link time using the
12     -- specified pathname. Should have store rights.
13
14
15     function Get_type_name_attr_ID
16         return Attribute_Mgt.attribute_ID_AD
17     is
18     begin
19         return type_name_attr_ID;
20     end Get_type_name_attr_ID;
21
22
23     package body Ops is
24         --
25         -- Logic:
26         -- Attribute packages have null bodies.
27
28
29     end Ops;
30
31
32 end Type_Name_Attribute_Ex;
```

## X-A.7.15 Type\_Name\_Attribute\_Init\_Ex Procedure

Creates the type name attribute ID.

```
1  with
2    Attribute_Mgt,
3    Conversion_Support_Ex,
4    Directory_Mgt,
5    Passive_Store_Mgt,
6    System_Defs,
7    Transaction_Mgt;
8
9  procedure Type_Name_Attribute_Init_Ex is
10   --
11   -- Function:
12   --   o Create new attribute.
13   --
14   --   o Store new attribute. If attribute already
15   --     exists, all changes are rolled back and the
16   --     procedure exists
17   --
18   --   o Update new attribute.
19   --
20   -- History:
21   --   05-10-1988:  Tobias Haas:  Initial version.
22
23   typ_nam_attr_ID_AD: Attribute_Mgt.attribute_ID_AD;
24   -- New attribute.
25
26   begin
27     Transaction_Mgt.Start_transaction;
28     -- Transaction ensures that both operations, Store and
29     -- Update, will take place together or not at all.
30     begin
31       typ_nam_attr_ID_AD := Attribute_Mgt.Create_attribute_ID(
32         type_specific => true);
33       -- Create new attribute.
34
35       Directory_Mgt.store(
36         name => System_Defs.text'(10, 10,"typnamattr"),
37         object => Conversion_Support_Ex.Untyped_from_attribute_ID(
38           typ_nam_attr_ID_AD));
39       -- Store attribute. If attribute already exists, this
40       -- operation will cause the Directory_Mgt.entry_exists
41       -- exception to be raised.
42
43       Passive_Store_Mgt.Request_update(Conversion_Support_Ex.
44         Untyped_from_attribute_ID(typ_nam_attr_ID_AD));
45       Transaction_Mgt.Commit_transaction;
46       -- Commit transaction after successful completion of
47       -- both operations.
48
49     exception
50       when Directory_Mgt.entry_exists =>
51         Transaction_Mgt.Abort_transaction;
52         -- If entry exists, roll back any changes.
53
54       when others =>
55         Transaction_Mgt.Abort_transaction;
56         RAISE;
57
58   end;
59   end Type_Name_Attribute_Init_Ex;
```

**X-A.7.16 Refuse\_Reset\_Active\_Version\_Ex Package Specification**

Type-specific implementation for stored accounts.

```
1  with System,
2      System_Exceptions,
3      Passive_Store_Mgt;
4
5  package Refuse_reset_active_version_Ex is
6
7      procedure Refuse_reset_active_version(
8          obj: System.untyped_word);
9
10     --
11     -- Function:
12     --   Handles requests to reset an account's active
13     --   version by refusing such requests.
14     pragma external;
15
16     pragma subprogram_value(
17         Passive_Store_Mgt.
18         Type_specific_reset_active_version,
19         Refuse_reset_active_version);
20
21 end Refuse_reset_active_version_Ex;
```

## X-A.7.17 Refuse\_Reset\_Active\_Version\_Ex Package Body

Type-specific implementation for stored accounts.

```
1 with System,
2     System_Exceptions,
3     Passive_Store_Mgt;
4
5 package body Refuse_reset_active_version_Ex is
6
7   -- History:
8   --   12-01-87: Tobias Haas, initial version.
9   --   04-20-87: Tobias Haas, added extractor comments bstex*.ex
10
11   procedure Refuse_reset_active_version(
12     obj: System.untyped_word)
13   is
14     --
15     -- Function:
16     --   Handles requests to reset an account's active
17     --   version by refusing such requests.
18     --
19
20   begin
21
22     RAISE System_Exceptions.operation_not_supported;
23
24   end Refuse_reset_active_version;
25
26 end Refuse_reset_active_version_Ex;
```

## X-A.7.18 Account\_Mgt\_Ex (Distributed) Package Body

Package body of the distributed account manager.

```

1  with
2    Access_Mgt,
3    Attribute_Mgt,
4    Authority_List_Mgt,
5    Directory_Mgt,
6    Distr_Acct_Call_Stub_Ex,
7    Long_Integer_Defs,
8    Object_Mgt,
9    Passive_Store_Mgt,
10   Semaphore_Mgt,
11   System,
12   System_Defs,
13   System_Exceptions,
14   Transaction_Mgt;
15
16 package body Account_Mgt_Ex is
17   --
18   -- Logic:
19   -- This is an implementation of the distributed
20   -- account manager. It follows the single activation
21   -- model. It has the following characteristics:
22   --
23   -- * All operations on accounts are centralized in
24   --   one home job. The home job is created at the node
25   --   where the first call to this package is made.
26   --
27   -- * Accounts can be stored anywhere on the system.
28   --
29   -- * Initialization, (creating the TDO, the server,
30   --   the service, installing the server, and setting up
31   --   the homomorph template) happen when the package is
32   --   elaborated.
33   --
34   -- * All synchronization is centralized in the
35   --   home job: Transactions are used to synchronize across
36   --   job boundaries and semaphores to synchronize between
37   --   different processes inside one job.
38   --
39   -- * This code is used in the home job and in all
40   --   other jobs. In the home job operations are
41   --   done directly. In all other jobs a call stub
42   --   package is called that issues RPCs
43   --   to the home job to perform the actual operation.
44   --
45   -- * The following picture
46   --   illustrates the structure of the distributed
47   --   implementation. Boxes represent independent jobs
48   --   that may run on any node. The names in the boxes
49   --   are the names of the packages.
50   --
51   --
52   --
53   --
54   --
55   --
56   --
57   --
58   --
59   --
60   --
61   --
62   --
63   --
64   --
65   --
66   --
67   --
68   --
69   --
70   --
71   --
72   --
73   --

```

+-----+	+-----+
Account_Mgt_Ex	Account_Mgt_Ex
Distr_Acct_	Distr_Acct_
Call_Stub	Call_Stub
+-----+	+-----+
Application	Application
Job	Job

+-----+
Distr_Acct_
Server_Stub
Account_Mgt_Ex
+-----+
Server Job
(Home Job)

```

71   -- * ADs to the TDO and the account service are created
72   --   by an initialization routine called Distr_acct_init
73   --   and stored with pathnames. They are retrieved by the

```

```

74      --          various models at link-time.
75      --
76      -- Exceptions:
77      --   no_server_installed:
78      --     Server for accounts is not installed.
79      --
80      -- History:
81      --   01-31-88: Tobias Haas, Initial version.
82      --   06-08-88: Tobias Haas, Design revision.
83
84      use Long_Integer_Defs, -- Import "long_integer", "zero",
85                          -- arithmetic, and relational
86                          -- operators.
87      System,              -- Import ordinal operators.
88      Transaction_Mgt;     -- Import transaction calls.
89
90
91      account_TDO: constant Object_Mgt.TDO_AD := null;
92      pragma bind(account_TDO, "account");
93      -- Constant AD to account TDO. Initially null.
94      -- Filled in at link-time.
95
96      type account_rep_object is
97      -- Representation of an account.
98      record
99          lock: Semaphore_Mgt.semaphore_AD;
100         -- Locking area
101         is_homomorph: boolean;
102         -- If false identifies the object
103         -- as the active version; if true
104         -- as a homomorph.
105         balance: Long_Integer_Defs.long_integer;
106         -- Starting balance.
107     end record;
108     FOR account_rep_object USE
109     record AT mod 32;
110         lock      at 0 range 0 .. 31;
111         is_homomorph at 4 range 0 .. 7;
112         balance   at 8 range 0 .. 63;
113     end record;
114     type account_rep_AD is access account_rep_object;
115     pragma access_kind(account_rep_AD, AD);
116     -- Private view of an account.
117
118
119
120 -----
121 -----
122 --                                     --
123 --                                     IS_ACCOUNT                                     --
124 --                                     --
125 -----
126 -----
127
128     function Is_account (
129         obj: System.untyped_word)
130     return boolean
131     --
132     -- Logic:
133     --   If "obj" is not null, retrieve the object's
134     --   TDO and check whether it is the account's TDO.
135     --
136     is
137     use Object_Mgt; -- Import "=" for type "TDO_AD".
138     begin
139     return obj /= System.null_word
140         and then
141         Object_Mgt.Retrieve_TDO(obj) = account_TDO;
142     end Is_account;
143
144 -----
145 -----
146 --                                     --
147 --                                     CREATE_ACCOUNT                                     --
148 --                                     --
149 -----
150 -----

```

PRELIMINARY

```

151
152 function Create_account(
153     starting_balance:
154         Long_Integer_Defs.long_integer :=
155         Long_Integer_Defs.zero)
156     return account_AD
157     --
158     -- Logic:
159     -- Creates an account by allocating an object
160     -- of type account. Storing the account is the
161     -- responsibility of the caller. Accounts can
162     -- be created in any account.
163     --
164     -- 1. Check initial balance.
165     --
166     -- 2. Allocate and initialize the account
167     -- object.
168     --
169     -- 3. Remove rep rights for the exported and
170     -- master AD.
171     --
172     -- 4. If any exception occurs, deallocate the object
173     -- and return.
174     --
175     is
176     account:         account_AD;
177     account_untyped: System.untyped_word;
178     FOR account_untyped USE AT account'address;
179     -- Account with no rep rights, viewed with
180     -- either of two types.
181
182     account_rep:     account_rep_AD;
183     account_rep_untyped: System.untyped_word;
184     FOR account_rep_untyped use AT
185         account_rep'address;
186     -- Account with rep rights, viewed with
187     -- either of two types.
188
189     trans: boolean := false;
190     -- True if a local transaction has been
191     -- started.
192     begin
193     -- 1. Check initial balance:
194     --
195     if starting_balance <
196         Long_Integer_Defs.zero then
197         RAISE insufficient_balance;
198
199     else
200     -- 2. Allocate and initialize the
201     -- account object:
202     --
203     account_rep_untyped := Object_Mgt.Allocate(
204         size => (account_rep_object'size+31)/32,
205         tdo => account_TDO);
206
207     begin
208     account_rep.all := account_rep_object'(
209         lock      => null,
210         is_homomorph => false,
211         balance   => starting_balance);
212     -- 3. Remove rep rights for the exported and
213     -- master AD:
214     --
215     account_untyped := Access_Mgt.Remove(
216         AD      => account_rep_untyped,
217         rights => Object_Mgt.read_write_rights);
218
219     exception
220     -- 4. If an exception occurs, deallocate the account
221     -- and reraise the exception:
222     --
223     when others =>
224
225         Object_Mgt.Deallocate(account_untyped);
226         RAISE;
227

```

```

228     end;
229     RETURN account;
230
231     end if;
232
233     end Create_account;
234
235 -----
236 -----
237 --
238 --             CREATE_STORED_ACCOUNT
239 --
240 -----
241 -----
242
243
244     function Create_stored_account (
245         starting_balance:
246             Long_Integer_Defs.long_integer :=
247             Long_Integer_Defs.zero;
248         master: System_Defs.text;
249         authority:
250             Authority_List_Mgt.authority_list_AD :=
251             null)
252     return account_AD
253 --
254 -- Logic:
255 -- Any job can create accounts. In order to
256 -- ensure that no multiple active versions of
257 -- any account exist the active version is
258 -- deallocated as soon as it has been
259 -- passivated. Passivating the master AD
260 -- and deallocating the active version
261 -- are enclosed in a transaction.
262 -- These are the steps:
263 --
264 -- 1. Check initial balance.
265 --
266 -- 2. Allocate and initialize the account
267 --    object.
268 --
269 -- 3. Remove rep rights for the exported and
270 --    master AD.
271 --
272 -- 4. Start a local transaction if there is
273 --    not a transaction on the stack.
274 --
275 -- 5. Create a master AD. Use "Store". This also
276 --    sets the object's authority list.
277 --
278 -- 6. Passivate the account.
279 --
280 -- 7. Deallocate the active version of the
281 --    account.
282 --
283 -- 8. Commit any local transaction.
284 --
285 -- 9. If an exception occurs, abort any local
286 --    transaction, deallocate the account
287 --    and reraise the exception.
288 --
289     is
290     account:          account_AD;
291     account_untyped: System.untyped_word;
292     FOR account_untyped USE AT account'address;
293     -- Account with no rep rights, viewed with
294     -- either of two types.
295
296
297     account_rep:      account_rep_AD;
298     account_rep_untyped: System.untyped_word;
299     FOR account_rep_untyped use AT
300     account_rep'address;
301     -- Account with rep rights, viewed with
302     -- either of two types.
303
304

```

**PRELIMINARY**

```

305     trans: boolean := false;
306     -- True if a local transaction has been
307     -- started.
308     begin
309     -- 1. Check initial balance:
310     --
311     if starting_balance <
312     Long_Integer_Defs.zero then
313     RAISE insufficient_balance;
314
315     else
316     -- 2. Allocate and initialize the
317     -- account object:
318     --
319     account_rep_untyped := Object_Mgt.Allocate(
320     size => (account_rep_object'size+31)/32,
321     tdo => account_TDO);
322     account_rep.all := account_rep_object'(
323     lock => null,
324     -- Null because 'lock' is not present
325     -- in passive version.
326     is_homomorph => false,
327     balance => starting_balance);
328
329
330     -- 3. Remove rep rights for the exported and
331     -- master AD:
332     --
333     account_untyped := Access_Mgt.Remove(
334     AD => account_rep_untyped,
335     rights => Object_Mgt.read_write_rights);
336
337
338     -- 4. Start a local transaction if there is
339     -- not one on the stack:
340     --
341     if Transaction_Mgt.Get_default_transaction =
342     null then
343     Transaction_Mgt.Start_transaction;
344     trans := true;
345     end if;
346
347
348     begin
349     -- This block controls the scope of
350     -- the exception handler.
351
352     -- 5. Create the master AD:
353     --
354     Directory_Mgt.Store(
355     name => master,
356     object => account_untyped,
357     aut => authority);
358
359     -- 6. Passivate the representation of the account:
360     --
361     Passive_Store_Mgt.Update(account_rep_untyped);
362
363     -- 7. Deallocate the active version of the
364     -- account:
365     --
366     Object_Mgt.Deallocate(account_rep_untyped);
367
368     -- 8. Commit any local transaction.
369     --
370     if trans then
371     Transaction_Mgt.Commit_transaction;
372     end if;
373
374     exception
375
376     -- 9. If an exception occurs, abort any local
377     -- transaction, deallocate the account and
378     -- reraise the exception:
379     --
380     when others =>
381     if trans then

```

```

382         Transaction_Mgt.Abort_transaction;
383     end if;
384     Object_Mgt.Deallocate(account_rep_untyped);
385     RAISE;
386
387     end;
388     RETURN account;
389
390     end if;
391 end Create_stored_account;
392
393
394 -----
395 -----
396 --
397 --             GET_BALANCE
398 --
399 -----
400 -----
401
402 function Get_balance(
403     account: account_AD)
404     return Long_Integer_Defs.long_integer
405     --
406     -- Logic:
407     -- 1. Amplify rep rights on the account AD.
408     --
409     -- 2. If "is_homomorph" is true:
410     --
411     --     * Call the call stub.
412     --
413     -- 3. If "is_homomorph" is false:
414     --
415     --     * Start transaction if there is not
416     --       one on the stack.
417     --
418     --     * Lock account with a semaphore.
419     --       (Deadlock is avoided by the
420     --       transaction timeout.)
421     --
422     --     * Read current balance.
423     --
424     --     * If an exeception occurs release the
425     --       account and abort any local transaction.
426     --
427     --     * Release the object and commit any local
428     --       transaction.
429
430 is
431     account_rep: account_rep_AD;
432     -- Account with rep rights.
433
434     account_rep_untyped: System.untyped_word;
435     FOR account_rep_untyped USE AT account_rep'address;
436     -- untyped view of account with rep rights.
437
438     account_no_rep_untyped: System.untyped_word;
439     FOR account_no_rep_untyped USE AT account'address;
440     -- Untyped view of account with no rep rights.
441
442     current_balance: Long_Integer_Defs.long_integer;
443     -- Current balance.
444
445     trans: boolean := false;
446     -- Is true if there is a local transaction.
447
448 begin
449     account_rep_untyped := account_no_rep_untyped;
450
451     -- 1. Amplify rep rights:
452     --
453     account_rep_untyped := Access_Mgt.Amplify(
454         AD => account_rep_untyped,
455         rights => Object_Mgt.Read_write_rights,
456         tdo => account_tdo);
457
458     if account_rep.is_homomorph then

```

PRELIMINARY

```

459
460      -- 2. We have a homomorph:
461      --
462
463      -- Call the call stub:
464      --
465      RETURN Distr_Acct_Call_Stub_Ex.
466          Get_balance(account);
467
468  else
469
470      -- 3. We are in the home job for accounts:
471      --
472
473      -- Start a local transaction if there is not one
474      -- on the stack:
475      --
476      if Transaction_Mgt.Get_default_transaction = null
477      then
478          Transaction_Mgt.Start_transaction;
479          trans := true;
480      end if;
481
482      begin
483          -- "P" locks the account object. If another
484          -- process has already locked the object wait
485          -- until the object is released. Transaction
486          -- timeout prevents a deadlock. (A finite timeout
487          -- value has to be set at node initialization.)
488          --
489          Semaphore_Mgt.P(
490              semaphore => account_rep.lock);
491
492          begin
493              -- Read current balance:
494              --
495              current_balance := account_rep.balance;
496
497              -- Release the account:
498              --
499              Semaphore_Mgt.V(
500                  semaphore => account_rep.lock);
501
502              -- Commit any local transaction:
503              --
504              if trans then
505                  Transaction_Mgt.Commit_transaction;
506              end if;
507
508              RETURN current_balance;
509
510          exception
511              -- Release the object:
512              --
513              when others =>
514                  Semaphore_Mgt.V(semaphore =>
515                      account_rep.lock);
516              RAISE;
517
518          end;
519
520          exception
521              -- Abort any local transaction:
522              --
523              when others =>
524                  if trans then
525                      Transaction_Mgt.Abort_transaction;
526                  end if;
527                  RAISE;
528          end;
529      end if;
530
531  end Get_balance;
532
533  -----
534  -----
535

```

```

536 --
537 -- CHANGE_BALANCE
538 --
539 -----
540 -----
541
542
543 function Change_balance(
544     account: account_AD;
545     amount: Long_Integer_Defs.long_integer)
546 return Long_Integer_Defs.long_integer
547 --
548 -- Logic:
549 -- 1. Check "account" for change rights and add rep
550 --    rights.
551 --
552 -- 2. If "is_homomorph" is true make a remote call.
553 --
554 -- 3. If "is_homomorph" is false update the balance
555 --    and return the new balance.
556 --
557 is
558     account_rep: account_rep_AD;
559     -- Account with rep rights.
560
561     account_rep_untyped: System.untyped_word;
562     FOR account_rep_untyped USE AT account_rep'address;
563     -- untyped view of account with rep rights.
564
565     account_no_rep_untyped: System.untyped_word;
566     FOR account_no_rep_untyped USE AT account'address;
567     -- Untyped view of account with no rep rights.
568
569     current_balance: Long_Integer_Defs.long_integer;
570     -- Current balance.
571
572     trans: boolean := false;
573     -- Is true if there is a local transaction.
574
575 begin
576     account_rep_untyped := account_no_rep_untyped;
577     account_rep_untyped := Access_Mgt.Import(
578         AD => account_rep_untyped,
579         rights => change_rights,
580         tdo => account_TDO);
581
582     if account_rep.is_homomorph then
583         RETURN Distr_Acct_Call_Stub_Ex.Change_balance(
584             account => account,
585             amount => amount);
586
587     else
588         if Transaction_Mgt.Get_default_transaction = null
589             then
590             Transaction_Mgt.Start_transaction;
591             trans := true;
592         end if;
593
594         begin
595             Semaphore_Mgt.P(account_rep.lock);
596
597             begin
598                 if account_rep.balance + amount < zero then
599                     RAISE insufficient_balance;
600
601                 else
602                     account_rep.balance := account_rep.balance +
603                         amount;
604                     Passive_Store_Mgt.Update(account_rep_untyped);
605                     Semaphore_Mgt.V(account_rep.lock);
606
607                     if trans then
608                         Transaction_Mgt.Commit_transaction;
609                     end if;
610                     RETURN account_rep.balance;
611
612                 end if;

```

PRELIMINARY

```

613
614     exception
615     when others =>
616         Semaphore_Mgt.V(semaphore =>
617             account_rep.lock);
618         RAISE;
619
620     end;
621
622     exception
623
624     when others =>
625         if trans then
626             Transaction_Mgt.Abort_transaction;
627         end if;
628         RAISE;
629
630     end;
631
632 end if;
633
634 end Change_balance;
635
636 -----
637 -----
638 --
639 --                TRANSFER
640 --
641 -----
642 -----
643
644 procedure Transfer(
645     source_account: account_AD;
646     dest_account:   account_AD;
647     amount:         Long_Integer_Defs.long_integer)
648 --
649 -- Logic:
650 -- 1. Check rights on both ADs and add rep rights.
651 --
652 -- 2. If "is_homomorph" is true make a remote call.
653 --    If "is_homomorph" is false proceed with the
654 --    transfer.
655 --
656 -- 3. If any of the resultant balances are negative
657 --    raise "insufficient_balance".
658 --
659 is
660     source_rep: account_rep_AD;
661
662     source_rep_untyped: System.untyped_word;
663     FOR source_rep_untyped USE AT source_rep'address;
664
665     source_no_rep_untyped: System.untyped_word;
666     FOR source_no_rep_untyped USE AT source_account'address;
667
668     dest_rep: account_rep_AD;
669
670     dest_rep_untyped: System.untyped_word;
671     FOR dest_rep_untyped USE AT dest_rep'address;
672
673     dest_no_rep_untyped: System.untyped_word;
674     FOR dest_no_rep_untyped USE AT dest_account'address;
675
676     trans: boolean := false;
677
678 begin
679     source_rep_untyped := source_no_rep_untyped;
680     source_rep_untyped := Access_Mgt.Import(
681         AD => source_rep_untyped,
682         rights => change_rights,
683         tdo => account_TDO);
684
685     dest_rep_untyped := dest_no_rep_untyped;
686     dest_rep_untyped := Access_Mgt.Import(
687         AD => dest_rep_untyped,
688         rights => change_rights,
689         tdo => account_TDO);

```

```

690
691 if source_rep.is_homomorph then
692     -- Only one of the accounts has to be checked.
693     Distr_Acct_Call_Stub_Ex.Transfer(
694         source_account => source_account,
695         dest_account   => dest_account,
696         amount         => amount);
697     RETURN;
698
699 else
700     if Transaction_Mgt.Get_default_transaction =
701         null then
702         Transaction_Mgt.Start_transaction;
703     end if;
704
705     begin
706         Semaphore_Mgt.P(
707             semaphore => source_rep.lock);
708
709         begin
710             Semaphore_Mgt.P(
711                 semaphore => dest_rep.lock);
712
713             begin
714                 if (source_rep.balance - amount < zero)
715                     or (dest_rep.balance - amount < zero)
716                 then
717                     RAISE insufficient_balance;
718
719                 else
720                     source_rep.balance :=
721                         source_rep.balance - amount;
722                     dest_rep.balance :=
723                         dest_rep.balance - amount;
724                     Passive_Store_Mgt.Update(source_rep_untyped);
725                     Passive_Store_Mgt.Update(dest_rep_untyped);
726                     if trans then
727                         Transaction_Mgt.Commit_transaction;
728                     end if;
729
730                 end if;
731                 RETURN;
732
733             exception
734                 when others =>
735                     Semaphore_Mgt.V(
736                         semaphore => dest_rep.lock);
737                     RAISE;
738
739             end;
740         exception
741             when others =>
742                 Semaphore_Mgt.V(
743                     semaphore => source_rep.lock);
744                 RAISE;
745
746         end;
747     exception
748         when others =>
749             if trans then
750                 Transaction_Mgt.Abort_transaction;
751             end if;
752         RAISE;
753
754     end;
755
756 end if;
757
758 end Transfer;
759
760 -----
761 -----
762 --
763 --
764 --
765 -----
766 -----

```

PRELIMINARY

```

767
768 procedure Destroy_account(
769     account: account_AD)
770     --
771     -- Logic:
772     -- 1. Check rights on "account". Add rep rights.
773     --
774     -- 2. If "is_homomorph" is true make a remote call.
775     --
776     -- 3. If "is_homomorph" is false proceed.
777     --
778     -- 4. Start a local transaction if there is not one
779     --     on the stack.
780     --
781     -- 5. lock the object with a semaphore
782     --
783     -- 6. Check that the account balance is zero,
784     --     otherwise raise an exception.
785     --
786     -- 7. Destroy the account's passive version.
787     --
788     -- 8. Get the name of the object's master directory
789     --     entry. (if any) Remove that entry. Note that
790     --     other entries and even a master AD may remain.
791     --
792     -- 9. If any exception occurs abort any local
793     --     transaction and reraise the exception.
794     --
795     -- 10. Deallocate the account's active version.
796     --
797 is
798     account_rep: account_rep_AD;
799
800     account_rep_untyped: System.untyped_word;
801     FOR account_rep_untyped USE AT account_rep'address;
802
803     account_no_rep_untyped: System.untyped_word;
804     FOR account_no_rep_untyped USE AT account'address;
805
806     trans: boolean := false;
807
808 begin
809     account_rep_untyped := account_no_rep_untyped;
810
811     account_rep_untyped := Access_Mgt.Import(
812         AD => account_rep_untyped,
813         rights => destroy_rights,
814         tdo => account_TDO);
815
816     if account_rep.is_homomorph then
817         Distr_Acct_Call_Stub_Ex.Destroy_account(
818             account => account);
819         RETURN;
820
821     else
822         if Transaction_Mgt.Get_default_transaction =
823             null then
824             Transaction_Mgt.Start_transaction;
825             trans := true;
826         end if;
827
828         begin
829             Semaphore_Mgt.P(
830                 semaphore => account_rep.lock);
831
832             declare
833                 path_length: integer := 60;
834
835             begin
836                 if account_rep.balance /=
837                     Long_Integer_Defs.zero then
838                     RAISE balance_not_zero;
839
840                 end if;
841                 Passive_Store_Mgt.Destroy(account_rep_untyped);
842
843             loop

```

```

844     declare
845     path_text: System_Defs.text(path_length);
846
847     begin
848     Directory_Mgt.Get_name(
849     obj => account_rep_untyped,
850     name => path_text);
851
852     if path_text.length >
853     path_text.max_length then
854     -- text was lost. Try again.
855     path_length := path_text.length;
856
857     else
858     Directory_Mgt.Delete(path_text);
859     EXIT;
860
861     end if;
862
863     exception
864     when Directory_Mgt.no_name =>
865     EXIT;
866
867     end;
868
869     end loop;
870     Semaphore_Mgt.Destroy_semaphore(
871     semaphore => account_rep.lock);
872     Object_Mgt.Deallocate(account_rep_untyped);
873
874     exception
875     when others =>
876     Semaphore_Mgt.V(
877     semaphore => account_rep.lock);
878     RAISE;
879
880     end;
881
882     exception
883     when others =>
884     if trans then
885     Transaction_Mgt.Abort_transaction;
886     end if;
887     RAISE;
888
889     end;
890
891     end if;
892
893     end Destroy_account;
894
895 end Account_Mgt_Ex;

```

## X-A.7.19 Distr\_Acct\_Call\_Stub\_Ex Package Specification

Call stub for the distributed account manager. Routes the type manager's requests.

```

1  with
2    Account_Mgt_Ex,
3    Authority_List_Mgt,
4    Long_Integer_Defs,
5    Object_Mgt,
6    System,
7    System_Defs;
8
9  package Distr_Acct_Call_Stub_Ex is
10   --
11   -- Function:
12   --   Call stub for distributed accounts
13   --   type manager. Packs parameters into buffers and
14   --   makes RPCs. Unpacks the results buffer
15   --   and returns results to front end of type
16   --   manager. "Is_account", "Create_account",
17   --   "Create_stored_account" are always forwarded
18   --   directly to the core and are therefore not
19   --   needed in the call stub.
20   --
21   -- Calls:
22   --
23   --   Get_balance           - Returns an account's
24   --                         balance.
25   --
26   --   Change_balance       - Changes an account's
27   --                         balance.
28   --
29   --   Transfer             - Moves an amount between
30   --                         accounts.
31   --
32   --   Destroy_account      - Destroys an account.
33   --
34
35
36   function Get_balance(
37     account: Account_Mgt_Ex.account_AD)
38     return Long_Integer_Defs.long_integer;
39   pragma protected_return(Get_balance);
40
41
42
43   function Change_balance(
44     account: Account_Mgt_Ex.account_AD;
45     amount: Long_Integer_Defs.long_integer)
46     return Long_Integer_Defs.long_integer;
47   pragma protected_return(Change_balance);
48
49
50
51   procedure Transfer(
52     source_account: Account_Mgt_Ex.account_AD;
53     dest_account: Account_Mgt_Ex.account_AD;
54     amount: Long_Integer_Defs.long_integer);
55   pragma protected_return(Transfer);
56
57
58
59   procedure Destroy_account(
60     account: Account_Mgt_Ex.account_AD);
61   pragma protected_return(Destroy_account);
62
63
64   pragma external;
65   -- Required if this package is used with the
66   -- "virtual" compilation model, which supports
67   -- multiple domains and multiple subsystems.
68
69   private
70
71   type account_object is
72     -- Empty dummy record. The object representation
73     -- is defined in the package body.

```

```
74     record
75         null;
76     end record;
77
78     pragma external;
79
80 end Distr_Acct_Call_Stub_Ex;
```

## X-A.7.20 Distr\_Acct\_Call\_Stub\_Ex Package Body

Call stub for the distributed account manager. Routes the type manager's requests.

```

1  with
2    Account_Mgt_Ex,
3    Distr_Acct_Server_Stub_Ex,
4    Job_Types,
5    Long_Integer_Defs,
6    Object_Mgt,
7    RPC_Call_Support,
8    RPC_Mgt,
9    Semaphore_Mgt,
10   System_Defs;
11
12  package body Distr_Acct_Call_Stub_Ex is
13
14    type account_rep_object is
15      -- Representation of an account.
16      record
17        lock: Semaphore_Mgt.semaphore_AD;
18        -- Locking area
19        is_homomorph: boolean;
20        -- If false identifies the object
21        -- as the active version; if true
22        -- as a homomorph.
23        balance: Long_Integer_Defs.long_integer;
24        -- Starting balance.
25      end record;
26
27      FOR account_rep_object USE
28        record AT mod 32;
29          lock      at 0  range 0 .. 31;
30          is_homomorph at 4  range 0 .. 7;
31          balance   at 8  range 0 .. 63;
32        end record;
33
34    type account_rep_AD is access account_rep_object;
35    pragma access_kind(account_rep_AD, AD);
36    -- Private view of an account.
37
38    service: constant RPC_Mgt.RPC_service_AD := null;
39    -- Distributed account service.
40    -- This is a constant but not really null. Will
41    -- be filled in with an AD retrieved by the linker.
42
43    pragma bind(service, "account_service");
44    -- Bind to account service
45
46
47
48  -----
49  -----
50  --
51  --
52  --
53  -----
54  -----
55
56  function Get_balance(
57    account: Account_Mgt_Ex.account_AD)
58    return Long_Integer_Defs.long_integer
59    --
60    -- Logic:
61    -- Pack Parameters into buffer and make RPC.
62    -- "Get_balance" has ordinal value 1
63    --
64  is
65    account_untyped: System.untyped_word;
66    FOR account_untyped USE AT account'address;
67    -- untyped view of account
68
69    current_balance: Long_Integer_Defs.long_integer;
70    -- Current balance.
71
72    parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
73    -- Buffers for parameters and results.

```

```

74
75     length: System.ordinal;
76     -- Used in remote call to hold actual length of
77     -- results buffer.
78
79     begin
80     -- Pack parameter buffer:
81     --
82     parameters := Distr_Acct_Server_Stub_Ex.buffer' (
83     first_word => account_untyped,
84     second_word => System.null_word,
85     -- irrelevant
86     amount     => Long_Integer_Defs.zero);
87     -- irrelevant
88
89     -- Make the RPC:
90     --
91     length := RPC_Call_Support.Remote_call(
92     service     => service,
93     target_proc => 1,
94     param_buf   => parameters'address,
95     param_length => parameters'size,
96     ADs_present => true,
97     results_buf => results'address,
98     results_length => results'size);
99     -- "length" is not used here.
100
101     current_balance := results.amount;
102     RETURN current_balance;
103
104     end Get_balance;
105
106     -----
107     -----
108     --
109     --
110     --
111     -----
112     -----
113
114     function Change_balance(
115     account: Account_Mgt_Ex.account_AD;
116     amount: Long_Integer_Defs.long_integer)
117     return Long_Integer_Defs.long_integer
118     is
119     account_untyped: System.untyped_word;
120     FOR account_untyped USE AT account'address;
121     -- untyped view of account.
122
123     parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
124     -- Buffers for parameters and results.
125
126     length: System.ordinal;
127     -- Used in remote call to hold actual length of
128     -- results buffer.
129
130     begin
131     parameters := Distr_Acct_Server_Stub_Ex.buffer' (
132     first_word => account_untyped,
133     second_word => System.null_word,
134     -- irrelevant
135     amount     => amount);
136
137     length := RPC_Call_Support.Remote_call(
138     service     => service,
139     target_proc => 2,
140     param_buf   => parameters'address,
141     param_length => parameters'size,
142     ADs_present => true,
143     results_buf => results'address,
144     results_length => results'size);
145     RETURN results.amount;
146
147     end Change_balance;
148
149
150     -----

```

PRELIMINARY

```

151 -----
152 --
153 --                TRANSFER                --
154 --
155 -----
156 -----
157
158 procedure Transfer(
159     source_account: Account_Mgt_Ex.account_AD;
160     dest_account:   Account_Mgt_Ex.account_AD;
161     amount:         Long_Integer_Defs.long_integer)
162 is
163     source_untyped: System.untyped_word;
164     FOR source_untyped USE AT source_account'address;
165
166     dest_untyped:   System.untyped_word;
167     FOR dest_untyped USE AT dest_account'address;
168
169     length: System.ordinal;
170
171     parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
172
173 begin
174     parameters := Distr_Acct_Server_Stub_Ex.buffer'(
175         first_word => source_untyped,
176         second_word => dest_untyped,
177         amount      => amount);
178
179     length := RPC_Call_Support.Remote_call(
180         service      => service,
181         target_proc  => 3,
182         param_buf    => parameters'address,
183         param_length => parameters'size,
184         ADs_present  => true,
185         results_buf  => results'address,
186         results_length => results'size);
187     RETURN;
188
189 end Transfer;
190
191 -----
192 -----
193 --
194 --                DESTROY_ACCOUNT          --
195 --
196 -----
197 -----
198
199 procedure Destroy_account(
200     account: Account_Mgt_Ex.account_AD)
201 is
202     account_untyped: System.untyped_word;
203     FOR account_untyped USE AT account'address;
204
205     parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
206
207     length: System.ordinal;
208
209 begin
210     parameters := Distr_Acct_Server_Stub_Ex.buffer'(
211         first_word => account_untyped,
212         second_word => System.null_word,
213         -- irrelevant.
214         amount      => Long_Integer_Defs.zero);
215         -- irrelevant.
216     length := RPC_Call_Support.Remote_call(
217         service      => service,
218         target_proc  => 4,
219         param_buf    => parameters'address,
220         param_length => parameters'size,
221         ADs_present  => true,
222         results_buf  => results'address,
223         results_length => results'size);
224     RETURN;
225
226 end Destroy_account;
227

```

228 end Distr\_Acct\_Call\_Stub\_Ex;

## X-A.7.21 Distr\_Acct\_Server\_Stub\_Ex Package Specification

Server stub for distributed account manager. Receives and forwards RPC's.

```

1  with
2    Long_Integer_Defs,
3    System;
4
5  package Distr_Acct_Server_Stub_Ex
6    --
7    -- Function:
8    --   This package contains the
9    --   server stub procedure for distributed
10   --   account services.
11   --   Corresponds to RPC_Mgt.server_stub.
12   --
13  is
14  type buffer is
15    -- Buffer used for remote calls.
16    record
17      first_word:   System.unsigned_word;
18      second_word: System.unsigned_word;
19      amount:       Long_Integer_Defs.long_integer;
20    end record;
21
22
23  FOR buffer USE
24    record AT mod 32;
25      first_word   at 0 range 0 .. 31;
26      second_word  at 4 range 0 .. 31;
27      amount       at 8 range 0 .. 63;
28    end record;
29
30  -- Exceptions:
31  --   System_Exceptions.operation_not_supported is raised when
32  --   a target procedure outside the range 0 .. 4 is specified.
33  --
34
35  procedure server_stub(
36    --
37    -- Function:
38    --   Depending on the value of "target_proc",
39    --   unpacks the parameter buffer, makes the
40    --   corresponding call to "Distr_SA_Account_Mgt_Ex",
41    --   packs the results buffer, and returns.
42    --
43    target_proc:   System.unsigned_short;
44    -- The number of the procedure to be called.
45    -- Has to be in the range 0 .. 4. The
46    -- assignments are as follows:
47    --
48    --   0:   Calls Passive_Store_Mgt.Set_home_job
49    --         in order to initialize the server.
50    --
51    --   1:   Calls Account_Mgt_Ex.Get_balance.
52    --
53    --   2:   Calls Account_Mgt_Ex.Change_balance.
54    --
55    --   3:   Calls Account_Mgt_Ex.Transfer.
56    --
57    --   4:   Calls Account_Mgt_Ex.Destroy_account.
58    --
59    version:       System.unsigned_short;
60    -- Not used.
61    param_buf:     System.address;
62    -- Address of parameter buffer.
63    param_length:  System.unsigned_short;
64    -- length of parameter buffer.
65    results_buf:   System.address;
66    -- Address of results buffer.
67    results_length: in out System.unsigned_short;
68    -- Length of results buffer.
69    ADs_returned:  out boolean;
70    -- Are any ADs returned. If false, speeds
71    -- up the call.
72
73  pragma external;
```

```
74  
75 end Distr_Acct_Server_Stub_Ex;
```

## X-A.7.22 Distr\_Acct\_Server\_Stub\_Ex Package Body

Server stub for distributed account manager. Receives and forwards RPC's.

```

1  with
2    Account_Mgt_Ex,
3    Long_Integer_Defs,
4    Object_Mgt,
5    Passive_Store_Mgt,
6    System,
7    System_Exceptions;
8
9  package body Distr_Acct_Server_Stub_Ex is
10  --
11  -- Function:
12  --   This package contains the server stub
13  --   procedure for the distributed account
14  --   service.
15  --
16  -- History:
17  --   01-31-88: Tobias Haas, Initial version.
18  --   04-07-88: Extensive Revision of design.
19
20
21  procedure server_stub(
22    target_proc:      System.short_ordinal;
23    version:          System.ordinal;
24    param_buf:        System.address;
25    param_length:     System.ordinal;
26    results_buf:      System.address;
27    results_length:   in out System.ordinal;
28    ADs_returned:    out boolean)
29  --
30  -- Function:
31  --   Procedure called by the account server
32  --   that unpacks the parameter buffer and
33  --   makes the appropriate calls.
34  --
35  -- Logic:
36  --   Depending on "target_proc" unpacks "param_buf"
37  --   makes the call and packs "results_buf".
38  --
39  is
40    account_TDO_untyped: System.untyped_word;
41    account_TDO: Object_Mgt.TDO_AD;
42    FOR account_TDO USE AT account_TDO_untyped'address;
43
44    account_one_untyped, account_two_untyped:
45      System.untyped_word;
46    account_one, account_two:
47      Account_Mgt_Ex.account_AD;
48    FOR account_one USE AT account_one_untyped'address;
49    FOR account_two USE AT account_two_untyped'address;
50
51    amount: Long_Integer_Defs.long_integer;
52
53    parameters, results: buffer;
54    FOR parameters USE AT param_buf;
55    FOR results USE AT results_buf;
56
57
58  begin
59    case target_proc is
60      when 0 => account_TDO_untyped := parameters.first_word;
61               Passive_Store_Mgt.Set_home_job(
62                 tdo => account_TDO);
63               ADs_returned := false;
64
65      when 1 => account_one_untyped := parameters.first_word;
66               amount :=
67                 Account_Mgt_Ex.Get_balance(
68                   account => account_one);
69               results := buffer'(
70                 first_word => System.null_word,
71                 -- irrelevant
72                 second_word => System.null_word,
73                 -- irrelevant

```

PRELIMINARY

```

74         amount      => amount);
75     ADs_returned := false;
76
77     when 2 => account_one_untyped := parameters.first_word;
78         amount :=
79             Account_Mgt_Ex.Change_balance(
80                 account =>
81                     account_one,
82                 amount =>
83                     parameters.amount);
84         results := buffer'(
85             first_word => System.null_word,
86             -- irrelevant.
87             second_word => System.null_word,
88             -- irrelevant.
89             amount      => amount);
90     ADs_returned := false;
91
92     when 3 => account_one_untyped := parameters.first_word;
93         account_two_untyped := parameters.second_word;
94         Account_Mgt_Ex.Transfer(
95             source_account => account_one,
96             dest_account  => account_two,
97             amount        =>
98                 parameters.amount);
99         results := buffer'(
100             first_word => System.null_word,
101             second_word => System.null_word,
102             amount      =>
103                 Long_Integer_Defs.zero);
104         -- irrelevant.
105     ADs_returned := false;
106
107     when 4 => account_one_untyped := parameters.first_word;
108         Account_Mgt_Ex.Destroy_account(
109             account => account_one);
110         results := buffer'(
111             -- irrelevant.
112             first_word => System.null_word,
113             second_word => System.null_word,
114             amount      =>
115                 Long_Integer_Defs.zero);
116     ADs_returned := false;
117     when others =>
118         RAISE System_Exceptions.operation_not_supported;
119     end case;
120
121     RETURN;
122
123     end server_stub;
124
125     end Distr_Acct_Server_Stub_Ex;

```

**X-A.7.23 Distr\_Acct\_Init Procedure**

Initializes the distributed account manager globally for a distributed system.

```

1  with Account_Type_Name_Ex,    -- Example package.
2  Attribute_Mgt,
3  Authority_List_Mgt,
4  Directory_Mgt,
5  Job_Types,
6  Identification_Mgt,
7  Object_Mgt,
8  Passive_Store_Mgt,
9  Refuse_reset_active_version_Ex, -- Example package
10 RPC_Mgt,
11 System,
12 System_Defs,
13 Transaction_Mgt,
14 Type_Name_Attribute_Ex, -- Example package.
15 User_Mgt,
16 Unchecked_conversion;
17
18 procedure Distr_acct_init
19 --
20 -- Function:
21 -- Initialization procedure for distributed
22 -- account service.
23 --   o Creates TDO.
24 --   o Initializes and stores attributes.
25 --   o Creates the service.
26 --   o Creates and installs the the server.
27 --   o Stores and updates TDO, server, and service.
28 --
29 -- Logic:
30 -- Private ADs are stored with pathnames and
31 -- protected by authority lists. They are retrieved
32 -- by the various modules that are part of the distributed
33 -- account service at link-time.
34 --
35 -- History:
36 -- 06-02-88: Tobias Haas, Initial version.
37 --
38 is
39 use Transaction_Mgt;
40 -- Import transaction operators.
41
42 -- Pathnames:
43 --
44 account_name: constant System_Defs.text :=
45   System_Defs.text'(7, 7, "account");
46 -- Pathname of account tdo.
47
48 service_name: constant System_Defs.text :=
49   System_Defs.text'(15, 15, "account_service");
50 -- Pathname of service AD.
51
52 server_name: constant System_Defs.text :=
53   System_Defs.text'(14, 14, "account_server");
54 -- Pathname of server job AD.
55
56 -- Private ADs:
57 --
58 account_TDO: constant Object_Mgt.TDO_AD :=
59   Object_Mgt.Create_TDO;
60 -- TDO for accounts.
61 server: constant RPC_Mgt.RPC_server_AD :=
62   RPC_Mgt.Create_RPC_server;
63 -- Server for accounts.
64
65 server_job: Job_Types.job_AD;
66 -- Installed server job.
67
68 service: RPC_Mgt.RPC_service_AD;
69 -- Distributed service AD.
70
71 -- Attribute-related stuff:
72 --
73 passive_store_impl:

```

```

74     Passive_Store_Mgt.PSM_attributes_AD;
75     -- Implementation of passive store attribute
76     -- for accounts.
77
78     type_name_impl: System.untyped_word;
79     -- Implementation of type name attribute
80     -- for accounts.
81
82     owner_only: User_Mgt.protection_set(1);
83     -- Protection set that includes only one ID, namely
84     -- the type manager's owner.
85
86     authority: Authority_List_Mgt.authority_list_AD;
87     -- Authority list that contains only one ID, namely
88     -- the type manager's owner.
89
90     type template is
91     record
92         dummy_word0: System.untyped_word;
93         is_homomorph: boolean;
94         dummy_word1: System.untyped_word;
95         dummy_word2: System.untyped_word;
96     end record;
97
98     FOR template USE
99     record AT mod 32;
100         dummy_word0 at 0 range 0 .. 31;
101         is_homomorph at 4 range 0 .. 7;
102         dummy_word1 at 8 range 0 .. 31;
103         dummy_word2 at 12 range 0 .. 31;
104     end record;
105
106     type homomorph_AD is access template;
107     pragma access_kind(homomorph_AD, AD);
108
109     homomorph: homomorph_AD;
110
111     -- Auxiliary Stuff:
112     --
113     trans: boolean := false;
114     -- Set if local transaction is started.
115
116
117     function Untyped_from_PSM_attributes is
118     new Unchecked_conversion(
119         source => Passive_Store_Mgt.PSM_attributes_AD,
120         target => System.untyped_word);
121
122
123     function Untyped_from_TDO is
124     new Unchecked_conversion(
125         source => Object_Mgt.TDO_AD,
126         target => System.untyped_word);
127
128
129     function Untyped_from_service is
130     new Unchecked_conversion(
131         source => RPC_Mgt.RPC_service_AD,
132         target => System.untyped_word);
133
134     function Untyped_from_homomorph is
135     new Unchecked_conversion(
136         source => homomorph_AD,
137         target => System.untyped_word);
138
139     function Untyped_from_job_AD is
140     new Unchecked_conversion(
141         source => Job_Types.job_AD,
142         target => System.untyped_word);
143
144     begin
145     -- 1. Allocate new passive store attribute implementation:
146     --
147     passive_store_impl := new
148         Passive_Store_Mgt.PSM_attributes_object;
149     -- 2. Allocate and initialize homomorph template:
150     --

```

## PRELIMINARY

```

151 homomorph := new template' (
152     dummy_word0 => System.null_word,
153     is_homomorph => true,
154     dummy_word1 => System.null_word,
155     dummy_word2 => System.null_word);
156
157 -- 3. Initialize passive store attribute implementation:
158 --
159 passive_store_impl.homomorph := Untyped_from_homomorph(homomorph);
160
161 passive_store_impl.reset :=
162     Refuse_reset_active_version_Ex.
163     Refuse_reset_active_version' subprogram_value;
164
165 passive_store_impl.copy_permitted := false;
166
167 passive_store_impl.locking_area_start := 0;
168 passive_store_impl.locking_area_end := 0;
169 -- Area in account where semaphore AD will be
170 -- stored when account is activated.
171 -- 4. Store passive store attribute implementation with type:
172 --
173 Attribute_Mgt.Store_attribute_for_type(
174     tdo => account_TDO,
175     attr_ID => Passive_Store_Mgt.PSM_Attributes_ID,
176     attr_impl => Untyped_from_PSM_attributes(
177         passive_store_impl));
178 -- Store PSM attribute.
179
180 -- 5. Initialize type name attribute implementation:
181 --
182 type_name_impl := Account_Type_Name_Ex'package_value;
183
184 -- 6. Store type name attribute implementation with type:
185 --
186 Attribute_Mgt.Store_attribute_for_type(
187     tdo => account_TDO,
188     attr_ID => Type_Name_Attribute_Ex.
189         Get_type_name_attr_ID,
190     attr_impl => type_name_impl);
191
192 server := RPC_Mgt.Create_RPC_server;
193 -- 7. Install server:
194 --
195 server_job := RPC_Mgt.Install_RPC_server(
196     server => server);
197
198 -- 8. Create the service:
199 --
200 service := RPC_Mgt.Create_RPC_service(
201     server => server);
202
203
204 -- 9. Create authority list to protect private ADs:
205 --
206 owner_only.length := 1;
207 owner_only.entries(1).rights := User_Mgt.access_rights' (
208     true, true, true);
209 owner_only.entries(1).id := Identification_Mgt.Get_user_id;
210
211 authority := Authority_List_Mgt.Create_authority(owner_only);
212
213 -- 10. Store and Update the TDO, attributes and service.
214 -- Use transactions to protect these operations:
215 --
216 if Transaction_Mgt.Get_default_transaction =
217     null then
218     Transaction_Mgt.Start_transaction;
219     trans := true;
220 end if;
221
222 begin
223     Directory_Mgt.Store(
224         name => account_name,
225         object => Untyped_from_TDO(account_TDO),
226         aut => authority);
227

```

```

228 Directory_Mgt.Store(
229     name => service_name,
230     object => Untyped_from_service(service),
231     aut => authority);
232
233 Directory_Mgt.Store(
234     name => server_name,
235     object => Untyped_from_job_AD(server_job),
236     aut => authority);
237
238 Passive_Store_Mgt.Request_update(
239     Untyped_from_TDO(account_TDO));
240 Passive_Store_Mgt.Request_update(
241     Untyped_from_PSM_attributes(
242         passive_store_impl));
243 Passive_Store_Mgt.Request_update(
244     type_name_impl);
245 Passive_Store_Mgt.Request_update(
246     Untyped_from_homomorph(homomorph));
247
248
249     if trans then
250         Transaction_Mgt.Commit_transaction;
251     end if;
252 exception
253     when Directory_Mgt.entry_exists =>
254         if trans then
255             Transaction_Mgt.Abort_transaction;
256         end if;
257
258     when others =>
259         if trans then
260             Transaction_Mgt.Abort_transaction;
261         end if;
262         RAISE;
263
264 end;
265
266 end Distr_acct_init;

```

## X-A.7.24 Distr\_Acct\_Home\_Job\_Ex Procedure

Sets the home job of the account service.

```

1  with
2    Distr_Acct_Server_Stub_Ex,
3    Long_Integer_Defs,
4    Passive_Store_Mgt,
5    RPC_Call_Support,
6    RPC_Mgt,
7    System;
8
9
10 procedure Distr_Acct_Home_Job_Ex is
11
12   parameters, results: Distr_Acct_Server_Stub_Ex.buffer;
13   -- Buffers for remote call.
14
15   length:                System.ordinal;
16   -- Gives actual length of results buffer in remote call.
17   -- Not used here.
18
19   service: constant RPC_Mgt.RPC_service_AD := null;
20   pragma bind(service, "account_service");
21   -- Account service. Retrieved at link-time.
22
23   account_TDO_untyped: constant System.untyped_word
24                       := System.null_word;
25   pragma bind(account_TDO_untyped, "account");
26
27 begin
28   -- Set up server as home job
29   --   by calling procedure ``0``:
30   --
31   parameters := Distr_Acct_Server_Stub_Ex.buffer' (
32   first_word => account_TDO_untyped,
33   -- account TDO
34   second_word => System.null_word, -- Irrelevant.
35   amount      => Long_Integer_Defs.zero);
36   -- Irrelevant.
37
38   length := RPC_Call_Support.Remote_call(
39     service      => service,
40     target_proc  => 0,
41     -- Server will call Passive_Store_Mgt.Set_home_job.
42     param_buf   => parameters'address,
43     param_length => parameters'size,
44     ADs_present => true,
45     results_buf => results'address,
46     results_length => results'size);
47
48 end Distr_Acct_Home_Job_Ex;
```

## X-A.7.25 Makefile

Makefile for the the preceding account type manager programs. To use type:

- make acct\_active, or
- make non\_xo, or
- make stored

to create different executable versions of the account type manager. **NOTE:** The distributed type manager is not yet implemented.

```

1  #Definitions:
2  lib = ada_library
3  impl = stored.b
4  messages = "(acct_mgt.s acct_vis.b acct_main.sb)"
5
6  spec_obj = $(lib)/acct_types.s.obj \
7             $(lib)/conversion_support_ex.s.obj \
8             $(lib)/account_mgt_ex.s.obj \
9             $(lib)/acct_visual.s.obj
10
11 body_obj = $(lib)/acct_visual.b.obj \
12            $(lib)/acct_main_loop.b.obj \
13            $(lib)/account_mgt_ex.b.obj
14
15 tdo_spec_obj = $(lib)/type_name_attribute_ex.s.obj \
16               $(lib)/account_type_name_ex.s.obj \
17               $(lib)/refuse_reset_active_version_ex.s.obj
18
19 tdo_body_obj = $(lib)/type_name_attribute_ex.b.obj \
20               $(lib)/account_type_name_ex.b.obj \
21               $(lib)/refuse_reset_active_version_ex.b.obj
22
23 acct_active: $(spec_obj) $(body_obj) acct_active_body
24              link.ada acct_main_loop
25              manage.program acct_main_loop $(messages)
26              -mv acct_main_loop acct_active
27
28 non_xo: $(spec_obj) $(body_obj) non_xo_body stored_account_tdo_init_ex
29         stored_account_tdo_init_ex
30         link.ada acct_main_loop
31         manage.program acct_main_loop $(messages)
32         -mv acct_main_loop non_xo
33
34 stored: $(spec_obj) $(body_obj) stored_body stored_account_tdo_init_ex
35         stored_account_tdo_init_ex
36         link.ada acct_main_loop
37         manage.program acct_main_loop $(messages)
38         -mv acct_main_loop stored
39
40 acct_active_body: $(spec_obj) acct_active.b, account_mgt_ex.b.obj
41                  -ada acct_active.b
42
43 non_xo_body: $(spec_obj) non_xo.b, account_mgt_ex.b.obj
44             -ada non_xo.b
45
46 stored_body: $(spec_obj) stored.b, account_mgt_ex.b.obj
47             -ada stored.b
48
49 $(lib)/acct_visual.b.obj: $(spec_obj) \
50                          acct_vis.b
51                          -ada acct_vis.b
52
53 $(lib)/acct_main_loop.b.obj: $(spec_obj) \
54                              acct_main.sb
55                              -ada acct_main.sb
56
57 $(lib)/acct_visual.s.obj: $(lib)/acct_types.s.obj \
58                          $(lib)/account_mgt_ex.s.obj \
59                          acct_vis.s
60                          -ada acct_vis.s
61
62 $(lib)/acct_types.s.obj: $(lib)/account_mgt_ex.s.obj \

```

PRELIMINARY

```
63             acct_types.s
64     -ada acct_types.s
65
66 $(lib)/account_mgt_ex.s.obj: acct_mgt.s
67     pwd
68     -ada acct_mgt.s
69
70 $(lib)/conversion_support_ex.s.obj: conv.s
71     -ada conv.s
72
73 stored_account_tdo_init_ex: $(tdo_spec_obj) \
74     $(tdo_body_obj) \
75     type_name_attribute_init_ex \
76     acct_tdo.sb
77     -ada acct_tdo.sb
78     type_name_attribute_init_ex
79     link.ada stored_account_tdo_init_ex
80
81 $(lib)/refuse_reset_active_version_ex.b.obj: $(tdo_spec_obj)
82     -ada refuse_reset_av.b
83
84 $(lib)/type_name_attribute_ex.b.obj: $(tdo_spec_obj)
85     -ada typnam.b
86
87 $(lib)/account_type_name_ex.b.obj: $(tdo_spec_obj)
88     -ada actyna.b
89
90 $(lib)/refuse_reset_active_version_ex.s.obj: refuse_reset_av.s
91     -ada Refuse_reset_av.s
92
93 $(lib)/account_type_name_ex.s.obj: $(lib)/type_name_attribute_ex.s.obj \
94     actyna.s
95     -ada actyna.s
96
97 $(lib)/type_name_attribute_ex.s.obj: typnam.s
98     -ada typnam.s
99
100 type_name_attribute_init_ex: typnamattr.sb
101     -ada typnamattr.sb
102     link.ada type_name_attribute_init_ex
```

## X-A.7.26 Named\_copy\_ex Procedure

```

1  with Directory_Mgt,
2      Passive_Store_Mgt,
3      System,
4      System_Defs,
5      System_Exceptions,
6      Transaction_Mgt;
7
8  procedure Named_copy_ex(
9      source: System_Defs.text;
10     dest:   System_Defs.text)
11  is
12     --
13     -- Function:
14     -- Copies object tree at source pathname to
15     -- destination pathname. The source tree is the
16     -- named source passive object and all passive
17     -- objects reachable from it via successive
18     -- master AD references. The destination pathname
19     -- must not already exist.
20     --
21     -- "Named_copy_ex" is transaction-oriented.
22     --
23     -- Exceptions:
24     -- Directory_Mgt.entry_exists
25     -- Directory_Mgt.name_too_long
26     -- Directory_Mgt.no_access
27     -- System_Exceptions.bad_parameter -
28     -- Both the calling process and the
29     -- destination directory have a
30     -- null authority list.
31     -- System_Exceptions.
32     -- transaction_could_not_be_committed
33     --
34     -- Body:
35     -- If there is no default transaction, then a local
36     -- transaction is created and transaction timestamp
37     -- conflicts are handled locally. Any other
38     -- exception is handled by aborting any local
39     -- transaction and reraising the exception.
40     --
41     -- The root object AD is retrieved, a copy stub
42     -- is created, the copy stub AD is stored under
43     -- the destination pathname, and "Copy" is called
44     -- to copy the tree.
45
46     source_AD: System.untyped_word;
47     dest_AD:   System.untyped_word;
48  begin
49     loop
50     declare
51         trans: boolean := false;
52         -- Set if local transaction is started.
53         use Transaction_Mgt;
54         -- Import "=" for "transaction_AD".
55     begin
56         if Transaction_Mgt.Get_default_transaction
57            = null then
58             Transaction_Mgt.Start_transaction;
59             trans := true;
60         end if;
61
62         source_AD := Directory_Mgt.Retrieve(source);
63         dest_AD   := Passive_Store_Mgt.
64             Create_copy_stub(source_AD);
65         Directory_Mgt.Store(name => dest,
66             object => dest_AD);
67         Passive_Store_Mgt.Copy(source_AD, dest_AD);
68     if trans then
69         Transaction_Mgt.Commit_transaction;
70     end if;
71     RETURN;
72
73  exception
74  when System_Exceptions.
```

PRELIMINARY

```
75     transaction_timestamp_conflict =>
76     if trans then
77         Transaction_Mgt.Abort_transaction;
78         -- Loop back and try again if
79         -- transaction started locally.
80     else
81         RAISE;
82         -- Reraise the exception if the
83         -- transaction was already on the
84         -- transaction stack.
85     end if;
86
87     when others =>
88         if trans then
89             Transaction_Mgt.Abort_transaction;
90         end if;
91         -- Abort the transaction if it was
92         -- started locally.
93         RAISE;
94         -- Reraise exception that invoked handler.
95
96     end;
97 end loop;
98
99 end Named_copy_ex;
```

X-A.7.27 `Older_than_ex` Function

```

1  with Long_Integer_Defs,
2      Passive_Store_Mgt,
3      System,
4      System_Exceptions;
5
6  function Older_than_ex(
7      a: System.untyped_word;
8      b: System.untyped_word)
9      return boolean
10 is
11     --
12     -- Function:
13     -- Returns true if object "a"'s passive version is
14     -- older than object "b"'s passive version.
15     --
16     -- Exceptions:
17     -- System_Exceptions.bad_parameter -
18     -- Either "a" or "b" does not have a passive
19     -- version.
20
21     use Long_Integer_Defs;
22     -- Import "<" for long integers.
23
24     a_info: Passive_Store_Mgt.passive_object_info;
25     b_info: Passive_Store_Mgt.passive_object_info;
26 begin
27     a_info := Passive_Store_Mgt.
28         Request_passive_object_info(a);
29     b_info := Passive_Store_Mgt.
30         Request_passive_object_info(b);
31
32     if not a_info.valid or else not b_info.valid then
33         RAISE System_Exceptions.bad_parameter;
34
35     else
36         RETURN a_info.write_time < b_info.write_time;
37
38     end if;
39 end Older_than_ex;

```



# GLOSSARY **B**

This glossary defines important terms used in this manual. Some definitions apply to this manual and some apply to other parts of the BiiN™ system.

## A

### **AD (access descriptor)**

(1) A protected pointer to a system object. An AD identifies a particular object and includes *rights* that determine what operations are allowed on the object via the AD. An AD can also be null, referencing no object. (2) In Ada, one of the alternatives used by pragma `ACCESS_KIND`.

### **abort**

Terminate a transaction unsuccessfully, reversing all changes associated with the transaction.

### **abstract data type**

A data type with an unspecified representation. An abstract data type is defined entirely by its supported operations. OS object types such as files and directories are abstract data types.

### **access**

Read or modify an object or datum.

### **access descriptor (AD)**

A protected pointer to a system object. An AD identifies a particular object and includes *rights* that determine what operations are allowed on the object via the AD. An AD can also be null, referencing no object.

### **access method (AM)**

A distinct way to use a device, defined by a set of I/O operations (typically `Open`, `Close`, `Read`, and `Write`). There are four access methods: byte-stream I/O, record I/O, character display I/O, and graphics display I/O. Each method is defined by a separate BiiN™ Ada package. Each device (pipe, file, directory, and so forth) supported by an access method has a different subset of the total operations available for the access method.

### **access rights**

Bits in an access descriptor (AD) that restrict the sets of operations you can perform to manipulate an object. Access rights consist of three type rights bits (typically mapped to use, modify, and control for a particular service) and two representation rights bits (read and write). Type rights can be thought of as permissions granted to a caller by a service's type manager. The permissions allow the caller to perform certain operations on the type manager's objects. The representation rights bits are used only by type managers to read from and write to the representation of a particular type of system object.

**access type**

An Ada type consisting of pointers to values of a specified second type. Values of a particular access type are represented by either ADs, virtual addresses, linear addresses, or heap offsets. The `access_kind` pragma is used to specify the representation of an access type. Each access type also includes the special value `null`, indicating a pointer to nothing. If an access type is represented with ADs then referenced values are represented by system objects.

**action**

(1) A record that specifies an event to be signaled, a destination to which the event is signaled, and an optional two-word message to all receivers of the event. A valid destination is a process, a job, or an event cluster. (2) In SMS, the user-defined command to be executed when a condition on a target is satisfied. The possible actions for an SMS event include sending a `mail` message to the subscriber, broadcasting a message, or executing a BiiN™ CL command script in a batch session.

**activation model**

A characteristic of an object type that specifies how objects of the type are activated. The multiple activation model activates an object in any job or node. The single activation model activates an object only in a particular home job (for local objects) or home node (for global objects); another job or node that attempts to activate the object instead activates a *homomorph*, a token object that stands in place of the actual object.

**activate**

To create an active version of a system object from its current passive version. Objects are activated automatically when there is no active version of an object and a program references the object's representation. Activating an object activates ADs in the object but does not activate referenced objects.

**active memory**

The virtual memory of a particular BiiN™ node, as distinct from the passive store of a distributed BiiN™ system.

**active AD**

An AD in active memory, represented by one memory word.

**active object**

A system object in active memory.

**active version**

An active object that has been activated from a passive version. An object can have multiple active versions, in different jobs or at different nodes.

**active-only object**

An object that does not and cannot have a passive version. An object's type determines whether or not it can be passivated.

**actual parameter**

Value or variable supplied as a parameter in a specific invocation of a call.

## **Ada**

A standard programming language for programming large-scale and real-time systems. BiiN™ Ada is a complete implementation of Ada as specified by ANSI/MIL-STD-1815A, 22 January 1983. The BiiN™ Ada implementation adds implementation-defined pragmas and attributes as the standard allows.

## **address**

A value that can be used to access a particular object or memory location. An address may be an AD, virtual address, linear address, or physical address. Physical addresses are only used by the hardware and inside the OS.

## **address space**

A set of memory locations. Each location is an <address, value> tuple. Address spaces include object address spaces, virtual address spaces, linear address spaces, and physical address spaces.

## **address translation**

The process of converting a linear address or virtual address to a physical address. Address translation may trigger paging or object activation to load needed information into physical memory.

## **advisory parameter**

A parameter that advises a service but does not dictate its actions. A service may ignore an advisory parameter or substitute a different value.

## **aggregate**

(1) An Ada composite value, of an array or record type, consisting of element values listed within parentheses. (2) In C, an array, structure, or union.

## **age factor**

The rate at which a waiting job ages in the scheduler's waiting queue (regardless of priority or service level). On every scan of the waiting queue, the age factor is added to the job's age to determine a new age. The larger the aging factor, the faster a job ages, and the sooner it rises to the front of the waiting queue.

## **alias**

(1) In general, an entity that stands for another entity. (2) In the BiiN™ OS, a non-master passive AD. (3) In BiiN™ C, an identifier that is defined with the `#pragma alias` preprocessor control and is used to associate an identifier with its external definition. This type of alias is needed to refer to functions or data implemented in other languages with different forms for identifiers. (4) In the BiiN™ Systems Object Module Format, a two-byte number used as an abbreviation for a symbolic name in a single object module. (5) In CLEX, a short command that stands for a longer command.

## **alias AD**

A passive AD that is not a master AD. An alias AD can refer to an object stored on a different volume set than the AD itself.

## **amplify**

Add rights to an AD to some object. Amplifying rights is a privileged operation, requiring an AD to the object's TDO, with amplify rights.

**amplify rights**

A type right for TDOs, required to amplify rights on ADs.

**argument**

(1) Values specified as part of a command. Arguments are defined with the `manage.commands` utility. An argument may be *mandatory* or *optional*. An argument has a name (prefixed by a colon: `:argument_name`), a type (one of: *boolean*, *integer*, *pointer*, *range*, *string*, *string list*, or *derived*), and a value (`[=some_value]`). Optional arguments may have a default value. Arguments may be entered in *named* or *positional* notation. (2) An expression that appears within the parentheses of a subprogram call. The expression is evaluated and the result is copied into the corresponding parameter of the called function.

**array type**

A structured data type consisting of a fixed number of components or elements, which are all of the same type.

**ASCII (American Standard Code for Information Interchange)**

A standard seven-bit code representing alphabetic, numeric, punctuation, mathematical, and control characters.

**atomic operation**

An operation that always succeeds completely or fails completely. An atomic operation never produces partial output or partial changes in its environment before failing. An atomic operation may also acquire locks to ensure that intermediate results are not visible to concurrent operations.

**attribute**

(1) A property that can be associated with multiple system objects or object types. (2) A language-defined characteristic of a named Ada entity, such as `'size` or `'image`. Some Ada attributes are functions.

**attribute call**

A subprogram invocation where the module implementation used is selected at invocation-time, based on the object type of the invocation's first actual parameter.

**attribute entry**

An `<attribute ID, attribute value>` tuple that gives an attribute's value for a particular system object or object type.

**attribute ID**

A system object that identifies an attribute.

**attribute instance**

An attribute value stored in a particular TDO.

**attribute list**

A system object that contains a list of object-specific attribute entries, for a particular object.

**attribute package**

A package that has different implementations for different system object types or system

objects. For example, `Byte_Stream_AM.Ops` is an attribute package. An attribute package can only contain subprograms.

**authority list**

List of IDs and associated type rights. An authority list is associated with an object, and a caller must hold an ID that matches one in the authority list, with the appropriate rights, before the caller can access that object.

**B**

**backup service**

The OS service that manages backup and restore operations.

**base priority**

The lowest priority a process can have. It is determined initially by the SSO priority of its job (for a job's initial process) or by the base priority of its parent (for a spawned process). It may be changed by the user or the system administrator.

**basic disk**

A device that supports low-level access to a disk as an array of sectors or bytes via record I/O or byte stream I/O.

**basic I/O service**

The OS service that manages byte stream I/O, standard Ada I/O, and common I/O definitions.

**basic streamer**

A device that supports low-level access to a streamer tape via record I/O and byte stream I/O.

**batch job**

A job that consists of a batch of requests ( a background job with no attached user). Like interactive jobs, batch jobs run in normal memory, have limited processor claim, and have a lower priority than real-time and time-critical jobs.

**bi-paged object**

An object representation in which the object is so large that its page table must also be paged. A bi-paged object's size ranges from 8M bytes to 4G bytes.

**body**

A BiiN™ Ada program unit containing the declarations and statements that implement a package, subprogram, or task specification.

**byte stream I/O**

An I/O access method that provides data transfer as an uninterpreted stream of bytes. Some implementations support random access to particular byte positions in the stream.

**blocked**

State of a process that is unable to execute because it is waiting on an event, a port, or a semaphore.

**Boolean**

(1) Either true or false. (2) In BiiN™ Pascal, a predefined type.

**built-in commands**

Commands built into BiiN™ CL, part of all command sets. Built-in commands entered to CLEX or to a program are executed by the command service itself.

**byte**

A unit of memory containing eight bits and aligned at an 8-bit boundary. Each byte has a distinct address, whether linear, virtual, or physical addresses are used. Bits in a byte are numbered from 0 to 7.

**C****call**

(1) A subprogram. (2) A particular invocation of a subprogram. (3) To invoke a subprogram.

**central system**

Central part of a BiiN™ node, containing one or more P7 GDPs, one or more system buses, and shared memory.

**Channel Processor (CP)**

A P7 component that handles I/O transfers between a BiiN™ node's central system and I/O subsystems. The CP is the main hardware component of an I/O module.

**character display I/O**

An interactive access method that provides operations on character display terminals. Character display I/O is defined by the `Character_Display_AM` package. Character display I/O can also be used for output to printers.

**character display device**

A device that displays and manipulates ASCII characters on a two-dimensional surface. Typical examples are printers and windows on terminal screens; typical operations on such devices include input, output, cursor movement, manipulation of the display surface, control and status activities, and identifying and changing the attributes associated with a device.

**character terminal**

A terminal that has some subset of the features specified in the *ANSI X3.64* standard; for example, character insertion and deletion, line insertion and deletion, cursor positioning, and scrolling. The *DEC VT-100* (a trademark of Digital Equipment Corp.) is a typical character terminal.

**character terminal manager**

A device manager that supports access to character terminals.

**character terminal user agent**

Software that allows a user to control the windows on a character terminal. It is provided by the character terminal manager.

**child process**

A process that is created (spawned) by another process (called the parent process).

## **CL (Command Language)**

The BiiN™ command language, used for invoking and controlling the execution of programs and scripts. CL is implemented by the command service.

## **Clearinghouse**

A BiiN™ database that keeps track of where objects and IDs are within an entire distributed system. While objects and IDs are actually stored on physical nodes, the Clearinghouse keeps track of which node houses which objects and IDs.

## **clearinghouse service**

The OS service that provides packages to manage the Clearinghouse to store names and node addresses across a distributed system.

## **CLEX (Command Language Executive)**

The BiiN™ command interpreter of BiiN™ CL commands. CLEX is used for invoking and controlling the execution of programs and BiiN™ CL scripts.

## **cluster**

Group of I/O queues that are serviced together. A cluster represents a group of devices, typically those serviced by the same CP task.

## **clustered file**

A structured file whose records are organized in related groups ("clusters") according to a clustering b-tree organization index.

## **command**

(1) A directive to a program (including *CLEX* itself) or script. A command consists of a *command name* followed by *command arguments* or *control options*. An *invocation command* is given to *CLEX* to invoke a program or BiiN™ CL script. *Runtime commands* are entered to control the operation of a program or BiiN™ CL script. *Built-in commands* are part of the command language (BiiN™ CL) itself. Commands are processed either by *CLEX* (*CLEX commands* and *invocation commands*), or by the Command Handler (*built-in commands* and *runtime commands*). (2) In mass storage I/O modules, a command defines the operation to be performed by the I/O Module.

## **command history**

A record of all entered commands. There are several *built-in commands* provided by the command service to create, list, and re-execute a command history (a *history log file*). There is also a *control option*, `::history`, which creates a history log file for the given command.

## **command name**

A sequence of characters, such as `create.alias`, that identifies a BiiN™ CL command. The command name is the first part of a complete *command*. There may be two parts in a command name, the verb (`create`) and the noun (`alias`), separated by a period. Command names may be shortened to the minimum unique abbreviation (`c.al`).

## **command script**

A file containing a sequence of BiiN™ CL commands that are interpreted by *CLEX*. A command script differs from a command file in two important ways: (1) You can pass arguments to a command script, but not to a command file. (2) The command script is

interpreted as a separate job, whereas a command file is executed in the program's environment.

**command service**

The service that parses and returns commands for programs (including CLEX itself) and BiiN™ CL scripts. Built-in commands are processed by the command service itself.

**command set**

A command set defines the *runtime commands* currently available. A program using the command service always has at least one command set. All command sets include the BiiN™ CL *built-in commands*.

**commit**

Complete a transaction successfully. If the transaction is not contained in some other transaction, then any changes associated with the transaction are made permanent.

**compaction**

A memory management daemon that relocates system objects and other memory segments to reduce fragmentation of normal memory. Compaction is transparent to application software.

**compilation unit**

(1) In general, a building block of a program or subsystem that, when compiled, produces a single object module. (2) When using the BiiN™ Application Debugger, a single unit of compilation, defined differently for each BiiN™ language and corresponding to a single object module. Referred to as a CU. (3) In BiiN™ Ada a specification or body of BiiN™ Ada package, subprogram, or task, presented for compilation as an independent text. (4) In BiiN™ C, any primary source file (excluding those that are "included").

**compiler**

A system program that translates high-level language source files into one or more object modules (contained in one or more object module files, depending on the language).

**concurrent**

Happening at the same time.

**concurrent program**

A program divided into pieces that appear to execute simultaneously.

**concurrent programming service**

The OS service that supports concurrent programs, programs with multiple processes and jobs executing together.

**configurable object**

A representation of a hardware or software component of a BiiN™ node that must be configured at node initialization, or can be dynamically added to a running system.

**configuration service**

The OS service that manages configuration of a BiiN™ node.

**consistency level**

Within transactions, the level of interference a transaction can tolerate within a file. A transaction can have level 1, level 2 or level 3 consistency.

**constant**

A value that does not change; can be either symbolic (named) or literal.

**constraint**

(1) BiiN™ Ada restriction on the set of possible values of a type or subtype. A range constraint specifies lower and upper bounds on the values of a scalar type. An accuracy constraint specifies the relative or absolute error bound on values of a real type. An index constraint specifies lower and upper bounds on an array index. A discriminant constraint specifies particular values of the discriminants of a record type or private type. (2) In BiiN™ SQL, a restriction on the set of possible values that may be stored in a column.

**constraint\_error exception**

BiiN™ Ada built-in exception raised by the BiiN™ Operating System or the BiiN™ Ada runtime system when a runtime constraint is violated. Common causes of `constraint_error` are (a) a value that violates a constraint in an assignment statement or subprogram call; or (b) a null access descriptor parameter.

**control option**

A predefined directive to a command that modifies the execution behavior or the I/O behavior of the command. A control option consists of a name (prefixed by a double colon, `::control_option`), and a value (`[=] value`).

**control rights**

One of three type rights. By convention, control rights are required to destroy or restructure an object.

**countable global object**

A global object that exists so long as any job may be using it. ADs to countable global objects are local ADs; such ADs cannot be stored in global objects.

**create right**

A type rights for TDOs and SROs. Creating an object requires create rights on the new object's TDO and on the SRO used to allocate the object.

**CRP**

The *current record pointer* represents the current location in a *structured file*.

**current directory**

Current location in a directory structure. If a relative pathname is specified, names are looked up starting from this directory. The current directory is always stored in process globals.

**current record pointer**

See CRP.

**cursor**

(1) In BiiN™ SQL, a named query. The cursor mechanism itself is a pointer that provides

row by row access to the result table produced by the query. The cursor can be moved with **FETCH** or **FETCH BACK**. (2) A special marker that identifies specific cells within a frame buffer. For example, a *write* operation might write characters at a cursor's current location and then move the cursor to a new location.

## D

### **daemon**

A server process that provides a service asynchronously. For example, daemons service pool queues, batch queues, and timed request queues. Memory management daemons provide compaction, and garbage collection.

### **data abstraction**

The design principle that data representation should be concealed from users of a data type, and that data should be defined to users in terms of its behavior, not its representation.

### **data area**

A set of disk space allocations on a single volume set. The primary data area contains the file's actual data. Secondary data areas are used to allocate space for indexes.

### **data definition service**

The OS service that manages data definitions.

### **deadlock**

A situation that occurs when two or more processes are blocked and each process is waiting for resources or signals controlled by other blocked processes.

### **debug object (DO)**

The (internal) part of a domain that contains the symbolic debug information for the domain. A debug object is composed of one or more debug units.

### **deallocate**

Destroy an object's representation in active memory. If the object has a passive version, then its active version can later be recreated.

### **declaration**

A program construct that associates a name with a program entity, such as a type, constant, variable, or subprogram.

### **default**

Value used for an actual parameter if no value is specified in the invocation.

### **default transaction**

Transaction (if any) at the top of a process's transaction stack. The default transaction is usually the most recent transaction started by the process. Transaction operations use the caller's default transaction if no transaction is explicitly specified.

### **default value**

A value assigned to a formal parameter when the corresponding actual parameter is omitted.

### **delete**

An operation used to remove a record, directory, character, object, or other entity.

**derived**

(1) In BiiN™ CL, an argument type. A derived argument's type is *derived* from the value's representation. A value of `true` or `false`, or just an argument name, implies a *boolean*; a series of digits implies an *integer*; a double period, optionally with an integer on either side, implies a *range*; a value in quotation marks is considered a *string*; string values in parentheses imply a *string list*. (2) A category of data types supported in BiiN™ C: arrays, pointers, structures, and unions.

**device**

Physical or logical entity that supports one or more access methods.

**device class**

A specification that defines the device-specific details necessary to access a member of a class of devices using an I/O mechanism.

**device driver service**

The OS service that supports device drivers.

**device manager**

Module that implements all operations on a particular device type. Implementations of each access method supported by the device type are part of the device manager.

**Device Services**

The OS service area that provides support to write and use device drivers.

**directory**

System object that associates names (entry names) with non-null ADs. A directory is the main way to associate a name with the AD's underlying object.

**directory entry**

A <name, AD> pair stored in a directory.

**directory name**

Part of a pathname that names the directory containing the named entry.

**Directory Services**

The OS service area that supports associating names with objects, protecting objects stored in directories, and retrieving objects based on a given name.

**discrete type**

A BiiN™ Ada enumeration type or integer type. Discrete types are used for array indexing, for loop iteration variables, and for choices in `case` statements and record variants.

**discriminant**

Record component that can determine the subtype of, or the presence or absence of, other record components.

**discriminant constraint**

Constraint on a record subtype that specifies a value for each discriminant of the record type.

**disk volume label**

A printable name assigned when a disk volume is logically initialized. This name is stored on the disk volume and does not have to be unique.

**dispatch**

Bind a ready process to an available General Data Processor (GDP) for execution.

**dispatching mix**

The set of jobs that are eligible for execution on a node. All processes in a job move in and out of the dispatching mix together, under control of the BiiN™ Operating System scheduler. A process can be blocked or suspended for other reasons while it is in the dispatching mix.

**dispatching port**

System object at which ready processes are queued to be dispatched and executed by P7 GDPs.

**distributed**

Property of a service that can be transparently accessed from different nodes in a BiiN™ distributed system.

**distributed service**

A service that can be transparently accessed from different nodes in a BiiN™ distributed system. For example, the object service, transaction service, naming service, and filing service are distributed services.

**distributed system**

A collection of hardware systems (*nodes*) connected by networks and sharing a common clearinghouse and one figurehead naming domain. The operating system unifies all the nodes into a single system, by providing distributed services that make data and resources accessible from any node.

**domain**

In architectural terms, a domain object, its associated linear address space, and software-predefined system objects.

**domain object**

A system object that defines and protects an execution environment.

**E****elaboration**

(1) Execution of a declaration in a BiiN™ Ada program unit or block. Elaboration executes any initialization code for variables or packages elaborated. (2) In BiiN™ Ada, the elaboration of a declaration is the process by which the declaration achieves its effect (such as creating an object); this process occurs during program execution. (3) When using the BiiN™ Application Debugger, the process by which program entities come into existence at run time. For example, the elaboration of a variable declaration involves allocating memory for a variable. A program entity cannot be accessed by the debugger until it has been elaborated.

**embedded object**

An object representation that is contained entirely in the object's descriptor. Only zero-length objects and semaphores use embedded representations.

**emulation**

An object that interprets higher-level printing functions for a printer and produces the expected output by simulating the function using more primitive functions available on the target printer.

**enumerated**

In BiiN™ CL, an argument subtype (of type string). An enumerated value has a defined set of allowable string values; for example, "start", "middle", "end".

**enumeration type**

Discrete type with values listed in the type declaration. Values of an enumeration type can be identifiers or (in BiiN™ Ada) character literals.

**error**

(1) One of the levels of diagnostics generated by the BiiN™ Ada, C, FORTRAN, COBOL, and Pascal compilers and the BiiN™ Systems Linker. Errors are conditions that may affect the generated output, but from which the compiler or linker can recover (by ignoring an operand or operation, modifying or ignoring a statement, and so on). Processing continues and output can be generated. However, the output may no longer do what you intended. (2) One of the exit codes provided by the BiiN™ Ada, C, FORTRAN, and Pascal compilers and the BiiN™ Systems Linker. This exit code indicates that one or more error or serious error diagnostics were issued.

**event**

(1) An indication of the occurrence of some activity within the system that concerns a process or group of processes. Events are local or global depending on the scope of their effect. (2) In SMS, a change in state of some object that is of interest to a user. An SMS event consists of a target, a condition and an action.

**event cluster**

System object that groups up to 32 events. Each process and job has its own associated event cluster. Programs can create additional event clusters and associate processes with them.

**event handler**

A procedure executed asynchronously in response to an event. Handler execution interrupts normal execution of the process that receives the event.

**environment variable**

Another name for a BiiN™ CL variable, especially those variables that control the behavior of an executing program.

**exception**

(1) In general, an error condition. (2) A BiiN™ Ada-defined error indication. To raise an exception transfers control to an exception handler. If the current block or call does not contain a handler for a raised exception, then the exception is propagated to the calling block or call, which may handle the exception or propagate it further. (3) A run-time condition

that may cause the output of a program to be wrong due to an algorithmic mistake in the source program or due to invalid input; also called a run-time error. The term exception implies that, in some cases, a routine can be called to handle the situation, and then processing can continue normally. (4) Raised by BiiN™ SQL procedures that are called by BiiN™ Ada procedures as an alternative to the standard SQLCODE parameter.

**exception handler**

A sequence of statements executed in response to an exception. Known as a trap handler in FORTRAN.

**executable program**

A collection of software modules that has been linked (using the BiiN™ Systems Linker) and is ready for execution on a BiiN™ system. An executable program must have a main entry point and should (but need not) have all of its symbolic references resolved.

**execute**

(1) To perform machine instructions. (2) To perform an I/O Module operation.

**execution environment**

Consists of a linear address space partitioned into four regions (static data, instruction, stack, and operating system-reserved), a set of global and floating-point registers, an instruction pointer, and an arithmetic control register.

**F**

**fault**

A processor-detected error during program execution. For example, if an addition operation overflows, the GDP detects the error and raises a fault, which is handled by the BiiN™ Operating System as an exception.

**fault tolerant**

Property of a hardware configuration that lets it continue operating after a component failure without losing or corrupting data or programs.

**field**

(1) In Pascal, a component or element of a record type. (2) In the BiiN™ operating system, a contiguous portion of a record that is an instance of a single data item.

**file**

(1) A collection of information on a physical input or output device. (2) A system object that stores data on disk, organized for efficient random access, reading, and writing. Files cannot contain access descriptors. Files support byte-stream I/O and record I/O. (3) In BiiN™ Pascal, a predefined type.

**file organization**

Data structure used for a file; one of: stream, sequential, clustered, hashed, unordered, and relative.

**filing volume set**

A volume set providing external storage space for files and objects.

**filing service**

The OS service that manages files and records.

**floating-point type**

A numeric data type that represents numbers using exponential notation:  $f \cdot 2^{**}e$  (where  $f$  is a positive or negative fraction, normally in the range:  $0.5 \leq |f| < 1.0$ ; and  $e$  is a signed integer). Floating-point numbers can represent a wide range of numbers, but with incomplete precision. They are called "floating point" because the radix point "floats" based on the varying exponent, instead of being determined by a fixed scale factor determined by the data type.

**form**

A displayable, interactive document with labels and spaces for entering data.

**formal parameter**

A parameter as viewed within the subprogram it is a parameter for. A formal parameter has a name, a type, and a mode. Each subprogram invocation associates a different actual parameter with each formal parameter.

**form description**

A DDef that describes the physical layout and interactive capabilities of a form.

**form service**

The OS service that manages forms.

**fragmentation**

The division of free storage into multiple non-contiguous segments, caused by the normal operation of heap allocation, deallocation, and garbage collection.

**frame buffer**

The drawing space of a virtual terminal. An application writes to the frame buffer associated with a virtual terminal. Part of the frame buffer is visible to a user through a window; this visible part is called a view.

**frozen memory**

Memory for system objects that are never swapped out to disk and never relocated by compaction. Contrast with *normal memory*. Frozen objects can be accessed without page faults or delays due to compaction. However, resizing a frozen object may make it inaccessible during the resize operation.

**full pathname**

A pathname with three leading slashes. The BiiN™ OS evaluates full pathnames by first discovering which node to begin from, which may require a call to the Clearinghouse.

**function**

(1) A BiiN™ Ada, FORTRAN, or Pascal subprogram that returns a value to its caller. (2) The primary unit from which C language programs are constructed. Functions need not return a value to the caller. All C functions are external; that is, a function cannot contain another function. (3) In BiiN™ SQL, one of a set of five "built-in" functions that take the rows in a table or the set of values in a column as an argument (MIN, MAX, SUM, AVG, COUNT).

## G

## G

$2^{30} = 1,073,741,824$ . For example, 1G bytes equals 1,073,741,824 bytes.

**garbage collection**

The process of identifying and reclaiming active objects that can no longer be accessed. Garbage collection reclaims both memory and object descriptors for reuse. Garbage collection is asynchronous and transparent to applications software. A global garbage collector reclaims global garbage and runs at every node, under administrative control. A local garbage collector is configured in a job if the running program requests it.

**garbage object**

An active object that cannot be accessed because it cannot be reached via active ADs. A garbage object can be reclaimed by garbage collection.

**generic object**

An object used as just a memory segment. A generic object does not have a type manager and all generic objects have the same TDO.

**generic package**

An Ada template for a package. Such a template can be instantiated with parameters at compile-time to create a package.

**generic subprogram**

An Ada template for a subprogram. Such a template can be instantiated with parameters at compile-time to create a subprogram.

**generic unit**

BiiN™ Ada template for a set of packages or subprograms. A package or subprogram created using the template is an instance of the generic unit. A generic instantiation is the kind of declaration that creates an instance. A generic unit is written as a package or subprogram specification prefixed by a generic formal part that may declare generic formal parameters. A generic formal parameter is either a type, subprogram, variable, or constant.

**global**

(1) An object or entity that is not local to a particular job. (2) A program-defined entity, such as a type, constant, or variable, that is declared outside a particular subprogram.

**global AD**

An AD that can be stored in a global object. A global AD's local bit is zero. A global AD normally references a global object.

**global debug table (GDT)**

A table of compilation units and their addresses generated by the BiiN™ Systems Linker.

**global garbage collector**

A memory management daemon that reclaims global garbage at a node. The global garbage collector is invisible to applications software. A system administrator controls a node's global garbage collector.

**global memory**

The collection of global objects in a node's active memory, combined with the free global memory available in the node's global SROs.

**global object**

A system object that exists outside of any particular job. A global object may be a countable global object or unbounded global object.

**global SRO**

An SRO used to allocate global objects. A node's active memory contains two global SROs, the normal global SRO and the frozen global SRO.

**global variable**

Global variables exist for the duration of a session. Variables created or modified by a program are local to the creating job, unless specified as global. Global variables are inherited by subsequent jobs in the same session.

**H****handler**

Code that is invoked by the BiiN™ Operating System or a language run-time system in response to an asynchronous occurrence rather than an application call. A handler can be an event handler, exception handler, or interrupt handler.

**handler object**

The handler object is a compiler-defined object that contains a table of the exception handlers defined in a domain. It is used by the compiler's runtime system to find the correct handler for a given exception.

**hashed+file**

A *structured file* whose records are organized according to a hashed organization index.

**HDLC service**

The OS service that manages High-Level Data Link Control communication.

**head object**

The initiating member of a pair of configurable objects associated with each other. A head object is characterized by its ability to function normally without being attached to another configurable object.

**high-level scheduling**

Putting a job in the hardware dispatching mix. When a job is invoked, it is enqueued on a scheduling port served by a scheduling daemon. When the daemon is activated, it removes the job from the port and schedules it by enqueueing the job's initial process at the end of one of the queues in a dispatching port. The port has 32 queues, ordered in priority from 0 (lowest) to 31 (highest). A process enqueued in this manner is said to be *in the mix*.

**history**

A record of occurrences.

**history log file**

A file of commands entered, and messages written, for a given job, session, or command. See *command history*.

**home directory**

Directory in which a user is placed after a successful login. The home directory is typically the highest directory owned by the user. All other stored objects owned by the user are normally subordinate to the home directory.

**home node**

The node at which a stored object's home volume set is currently mounted.

**home volume set**

The volume set that contains a stored object's passive version.

**homomorph**

An active version created as a token in place of a single-activation object that is only activated in a different home job or home node. The object's type manager must communicate with its counterpart at the home job or home node in order to access the object. Users of an object, outside its type manager, cannot distinguish between a homomorph and the object that it stands in place of.

**Human Interface Services**

The OS service area that provides integrated packages for quickly developing applications. All services in this area are based on a data definition (*DDef*) that supports the idea of building complex structures from small pieces (forms and reports), and that might be used to create informational output.

I

**ID**

(1) A system object that represents a particular class of access to a BiiN™ system. Each user is represented by an ID. Each group of users that share access to particular objects can be represented by an ID. The "world" class, denoting access granted to arbitrary other users, is represented by an ID. Application programs and type managers can use IDs to restrict access to stored objects to only certain programs or modules. (2) An index that identifies the device or controller to which an I/O module command/operation is directed.

**ID list**

A system object that contains a list of IDs. Each process has an associated ID list, referenced by its process globals, used for authority list evaluation in retrieving stored ADs protected by authority lists.

**I/O message**

A data transfer mechanism that is composed of four parts: a common, fixed part, a part for the exclusive use of a device driver and I/O processor, a part for the exclusive use of a device manager, and an array of buffer descriptions.

**I/O shared queues**

A data transfer mechanism employing an input and output queue per device. Designed for low-speed, character-oriented I/O, such as character terminals and printers.

**I/O Services**

The OS service area that supports all input/output to and from files and devices.

**image module**

An independently linked, protected, and potentially shareable piece of software that is bound to a program at runtime. Image modules support runtime linking, protection, and sharing.

**incident**

A BiiN™ construct that assigns a unique identifier, an *incident code*, to each error or exceptional situation. An incident code references a message file, an individual message within that file, and a severity level.

**incident code**

Representation of a software incident. An incident code indicates the module which defines the incident, the incident number within the module, the incident severity, and a pointer to a message file.

**index**

The mechanism in which a data value is presented to an ordered list that contains the location of the desired value in a file. The index does not often contain all the values of the data item, but simply a limiting range of values. An organization index is an index for a clustered file or hashed file that influences the placement of records in the primary data area. An alternate index is an index in a structured file that in no way influences the placement of records in the primary data area.

**index constraint**

A restriction on a BiiN™ Ada array type or subtype that specifies the lower and upper bounds (and thus the number of values) for each index (subscript) of the array.

**index type**

The type of the array selector or index that is used to reference an element of a Pascal array. A Pascal index type must be an ordinal type.

**initial\_age**

A job's age when it first enters the scheduler's waiting queue of swapped-out jobs. Larger values indicate older jobs. The job at the head of the queue is the oldest job and is scheduled for execution before the other jobs in the queue.

**input event**

An action performed by the user when interacting with an application through a terminal window. Typical examples are *mouse* and *keyboard* input events. Input events are forwarded to the application.

**input focus**

The virtual terminal to which a physical terminal's keyboard and mouse input are connected at a given time.

**instance**

Member of a class. For example, an instance of an attribute, an instance of a generic package.

**instantiation**

Operation performed by the BiiN™ Ada compiler to create an instance of a generic package or subprogram.

**instruction object**

The predefined system object that contains the code belonging to a particular domain. This object represents the instruction region, region 1.

**integer**

(1) An exact representation of a positive, negative, or zero value. (2) In BiiN™ CL, an argument or variable type. (3) One of the data types of BiiN™ FORTRAN. In BiiN™ FORTRAN, an integer datum can occupy 1, 2, 4, or 8 bytes; the default is 2 or 4, depending on the value of the compiler's `:intsize` argument. (4) In standard Pascal, a sequence of decimal digits. In BiiN™ Pascal, a sequence of binary, octal, decimal, or hexadecimal digits.

**integer type**

(1) Any type containing only whole numbers in a particular range. (2) One of the C-language data types `char` or `int` (all sizes, signed or unsigned). (3) One of the Pascal data types: `char` or `integer`.

**interactive job**

A job that interacts with a human user. Interactive jobs run in normal memory, have limited processor claim, and have a lower priority than real-time and time-critical jobs.

**interrupt**

Asynchronous hardware signal indicating some occurrence (such as I/O) that requires action by an I/O module.

**interrupt handler**

A procedure invoked in response to an interrupt.

**interrupt reply procedure**

A subprogram specified by a device manager in an I/O message that enables a device manager to process the reply information contained in an I/O message that has been serviced by either an I/O processor or a device driver.

**invocation command**

A BiiN™ CL command that invokes (calls and starts) a program or BiiN™ CL script.

**J****job**

A system object that represents an executing program. Each job has its own storage resource and its own address space. Each job has its own processing resources; scheduling for a node is done on a per-job basis. Resource control and reclamation is done on a per-job basis. A job can contain multiple processes executing concurrently.

**K****K**

$2^{10} = 1,024$ . For example, 1K bytes equals 1,024 bytes.

**key**

A value used to designate a data item in a *record*. A *primary key* is a key value that uniquely identifies a record in a file. A key value that does not uniquely identify a record in a file is a *secondary key*.

**kidnapped process**

Process interrupted by an interrupt handler. The process is restored to its prior state and resumes execution when the handler completes.

**L****LAN service**

The OS service that manages Local Area Network communication.

**library unit**

A compilation unit that is not a subunit of another unit. Library units belong to a program library.

**lifetime**

A system object characteristic that determines how long an object can exist and how the object can be deallocated. There are three possible lifetimes: local, countable global, and unbounded global. Local objects are local to a job, exist no longer than their job, and can be deallocated by job termination or a local garbage collector. Countable global objects are shared by one or more jobs and can be deallocated when the jobs are no longer using the objects. Unbounded global objects have an unbounded lifetime and can be reclaimed by global garbage collection when the objects are no longer accessible via any AD.

**lifetime check**

A check, whenever an AD is copied, to ensure that a local AD is not copied into a global object. Attempting such a copy raises `System_Exceptions.lifetime_violation`.

**limited type**

A BiiN™ Ada type that does not allow assignment or comparisons for equality.

**linear address**

A word interpreted as a 32-bit ordinal that specifies a byte offset into a linear address space. Bits 30 and 31 specify one of four region objects. Bits 0-29 specify a byte offset into the selected region. Region 0 contains static data. Region 1 contains instructions. Region 2 is a stack. Region 3 is used by the OS and is identical for all linear address spaces at a particular node.

**linear address space**

A  $2^{32}$  byte (4G byte) address space partitioned into four regions, defined by a domain and a particular process. A domain contains ADs for region 0 (static data object) and region 1 (instruction object). A domain contains a subsystem ID that determines which of a process's stacks is used as region 2. Region 3 is defined by the OS and never changes. The linear address space contains holes where region objects are less than 1G byte in size.

**link object**

A system object with a system object type that supports the BiiN™ Operating System link attribute. When an AD for a link object is retrieved from a directory, an associated link evaluation function is called to evaluate the link and return a different AD. For example, a symbolic link system object contains a pathname. Retrieving an AD for a symbolic link triggers the retrieval of the AD named by the pathname in the symbolic link object.

**linker**

The BiiN™ software tool that combines the object modules created by the BiiN™ Ada, C, FORTRAN, COBOL, Pascal, and SQL compilers with the languages and systems environment to build an executable program. Besides producing the executable program directly from the object modules created by compilers, the linker can also produce image modules from object modules.

**literal**

(1) A symbol or number that represents a specific value rather than naming a value defined elsewhere (variable or constant) or describing a computation (expression). A literal can be a numeric literal, enumeration literal, character literal, or string literal. (2) In BiiN™ SQL, the representation of character strings, exact numeric values (FIXED) and approximate numeric values (FLOAT).

**lock**

An entity that allows a *transaction* or *opened device* to ensure that it alone has access to a particular resource.

**local**

(1) An object or entity that is local to a particular job. (2) A scope of an entity, such as a constant or variable, that is declared and visible only within a particular subprogram or block.

**local AD**

An AD that is local to a job. A local AD cannot be contained in or copied to a global object.

**local bit**

A bit in an AD that is one in a local AD and zero in a global AD. The local bit is not interpreted in null ADs.

**local garbage collector**

A memory management daemon that reclaims local garbage within a job. A running program must request local garbage collection or else no daemon is created for the job. Once requested, local garbage collection is invisible to the application.

**local object**

A system object that is local to a particular job. When a job terminates, all its local objects are deallocated.

**local SRO**

An SRO used to allocate local objects. Each job has one local SRO.

**local variable**

Local variables exist only for the duration of a job. A variable created or modified by a program is local to the creating job, unless specified as global.

**low-level scheduling (dispatching)**

Assigning a process to a processor. Each processor has a pointer to a dispatching port. When a processor is available to execute a process, it dequeues the first process from the highest numbered, non-empty queue in the port, and executes it.

# M

## M

$2^{20} = 1,048,576$ . For example, 1M bytes equals 1,048,576 bytes.

### **mandatory argument**

An argument that must be entered as part of a complete command.

### **mass storage service**

The OS service that manages disk and tape storage.

### **master AD**

The first access descriptor stored in passive store for a particular object. An object's passive version is deleted when its master AD is deleted. If a master AD is stored in a directory entry and other directory entries on the same volume set reference the same object, then deleting the master AD converts the AD in one of those other entries to a master AD, preserving the object.

### **medium-level scheduling**

The process of dynamically assigning priorities to executing processes. Medium-level scheduling considers a process's running priority, service class, and dynamic behavior.

### **memory type**

The kind of memory used by a system object, either normal memory or frozen memory.

### **menu service**

The OS service that manages menus.

### **message**

(1) Information issued by an executing program in response to some internal or external incident. A message can have three levels (short, long, and help) and can exist in various message languages (English, German, etc.). (2) Information used in executing the action associated with an SMS event. For an action class of `command`, the message becomes a process global that contains information for the batch job that is triggered by the event. For an action class of `mail`, the message is sent to the mailboxes listed in the action refinement.

### **message file**

The container for a program's messages.

### **message service**

The OS service that manages system and application errors and messages.

### **message stack**

A stack that can be used to push and pop messages as execution continues. A message stack can thus contain a traceback of an error's propagation path from the point of error back through the various layers of software to the topmost level. Each process has a message stack associated with it.

### **menu**

A list of choices provided by a program. There are two types of menus: "pull-down menus" from Window Services, and "screen menus" from the Menu Facility. Pull-down menu titles are displayed in a line at the top of a window; selecting a pull-down menu title causes the

menu itself to be displayed. A screen menu (with its *menu items*) is displayed in a window under program control. Screen menus may have hierarchies of menus and submenus.

### **Menu Editor**

System utility used to interactively create and modify menus.

### **Menu Handler**

Ada package that processes menus.

### **menu item**

Element of a *menu* representing one of the choices available in the menu. Composed of the displayed menu item text, and the returned menu item index; see the `Window_Services` package.

### **mode**

The mode of a variable is either "read-only", meaning that the variable can only be read, or "read-write", indicating that the variable may be read or assigned a value.

### **modify rights**

One of three type rights. By convention, modify rights are required to change an object's state.

### **monitor service**

The OS service that supports monitoring of program execution.

### **multiple activation model**

An activation model that activates an object in any job or node. Compare with *single activation model*.

## **N**

### **name**

(1) A character string label for an object or a stored AD. (2) A program-defined label for a program entity, such as a type, variable, constant, exception, package, or subprogram.

### **name space**

A name space is a list of directories to be searched by the BiiN™ OS when looking for an object. This is similar in function to the UNIX environment variable `PATH` or the MS-DOS `PATH` command.

### **named association**

A BiiN™ Ada construct that binds a parameter or an aggregate member to a value; has the form `name => value`.

### **named notation**

(1) Entering an argument value to a command by specifying the name of the argument. (2) A BiiN™ Ada construct.

### **naming service**

The OS service that provides packages to manage pathnames, directories, and lists of directories.

**node**

A single BiiN™ hardware system. Multiple nodes can be combined into a single *distributed system*.

**node pathname**

A pathname with one leading slash. The BiiN™ OS evaluates node pathnames beginning at the calling node's root (top) directory.

**normal memory**

Memory for system objects that can have pages swapped out to disk and that can be relocated by compaction. Contrast with *frozen memory*. Accessing a normal object may encounter delays waiting for pages or waiting for compaction to relocate the object.

**null**

(1) An invalid address, a pointer to nothing. (2) In general, empty or missing.

**O****offset**

An unsigned displacement from some base address, typically from the beginning of an object. An offset is in bytes unless other units are explicitly specified.

**object**

(1) A typed, protected memory segment. Such an object is also called a system object. (2) In Ada: a typed container for a value, such as a variable or constant. An Ada object may or may not be represented by a separate memory segment.

**object address space**

Up to  $2^{26}$  system objects simultaneously addressable in a particular node's active memory.

**object descriptor**

A data structure used to hold various system object characteristics: size, location in memory, AD to the object's TDO, and other information. Object descriptors are internal to the OS; object descriptors are only described because it is difficult to explain how objects are located, sized, and typed without mentioning them.

**object index**

A field in an AD that identifies a particular object. In an active AD, the object index is a 26-bit index into the node's object table, selecting the object's descriptor.

**object orientation**

(1) A set of characteristics that enhance the coherence and security of integrated systems. The principal characteristic of object orientation is the use of protected data structures called objects to represent parts of the system itself as well as application entities. Objects are addressable and protected by cooperating hardware and software mechanisms. (2) An intuitive style of user interface that emphasizes representation of real-world entities rather than implementation-oriented details.

**object representation**

The contents of a system object. An object's representation can contain from zero to 4G bytes. The representation is not synonymous with the object itself because an object has several other characteristics, such as object type and attributes. Accessing an object's representation requires an AD or virtual address with rep rights.

**object section**

In the BiiN™ OMF, a contiguous portion of an object.

**object service**

The OS service that provides calls to manage objects, access to objects, and storage of objects.

**object table**

An object that contains all object descriptors for objects that are in a node's active memory or that have active ADs on the node. There is one object table per node. The object table is internal to the OS; it is described only because it is difficult to explain how objects are located, sized, and typed without mentioning the object table.

**object tree**

A collection of passive objects, beginning with a single root object, and linked by master ADs. An object  $x$  is in the tree if and only if  $x$  is the root object or another object in the tree contains  $x$ 's master AD. Because master ADs cannot refer to objects on other volume sets, all objects in an object tree are on the same volume set as the root object.

**object type**

A set of object attributes that indicates such characteristics as its purpose, visibility, and usability by other system elements. Some types define objects that are recognized by the processor and for which special instructions are provided. Software-defined types can be manipulated only by a type manager corresponding to the type of the object.

**object-specific attribute**

An attribute that is defined differently or not defined at all on a per-object basis.

**operator**

A programming language element that specifies an operation to be performed on one or more operands in an expression.

**operating system**

The OS provides:

- General management of objects: object-oriented storage, protection, naming, and programming.
- Control and accounting for system resources, such as memory and processing resources, in a multiuser environment.
- Device-independent I/O access methods.
- Support for concurrent programming.
- Distributed services, so that applications built on those services are naturally distributed.
- High-level services commonly needed by many applications, such as messages, structured files, commands, forms, and reports.

System Services is the programmer's interface to the OS.

**optional argument**

An argument to a command that need be entered only if a value other than the default is desired.

**organization pathname**

A pathname with 2 leading slashes. The BiiN™ OS evaluates organization pathnames by first discovering which node to begin from, which may require a call to the Clearinghouse.

**outside environment object (OEO)**

An object that references the command definitions and messages associated with a program. These are used by the command language executive (CLEX).

**P****package**

An Ada module containing logically related types, constants, variables, exceptions, subprograms (calls), and tasks. A package is represented by two separate compilation units, a package specification and a package body.

**package body**

The implementation of an Ada package specification. The body includes implementations for each subprogram in the package specification, any private data and subprograms internal to the body, and any needed package initialization code.

**package specification**

The external interface to an Ada package. Declarations in the public part of a package specification can be used from outside the package. A package specification can also contain a private part that provides information needed by the compiler but not available to external users.

**package type**

A package specification that can have alternate bodies, with a body selected for each call depending on the object type of the first actual parameter. Compare with *attribute call*.

**page**

(1) A 4K-byte memory block, aligned on a 4K-byte boundary. (2) A printed page.

**page descriptor**

A data structure that locates a particular memory page and that contains access rights and status information for the page.

**page table**

A table that locates the pages of a paged object. The table contains an array of page descriptors.

**page table directory**

A page table that locates the pages of a large page table that is itself paged.

**paged object**

A large object that is stored in multiple pages of physical memory. The object descriptor for a paged object references a page table that in turn references the pages of the object.

**paging**

The process of moving pages between physical memory and a swapping volume set. Pages are loaded into physical memory on demand. Modified pages are written to the swapping volume set by an asynchronous paging daemon.

**panning**

Moving a view up or down in its frame buffer in order to see a different part of the frame buffer. Also called *scrolling*.

**parameter**

A value or variable that can be different for each invocation of a subprogram, and thus is supplied for each invocation. A formal parameter represents a parameter within a subprogram body. An actual parameter is the actual value or variable supplied for a particular invocation.

**parameter mode**

For an Ada parameter, one of:

- in                      The parameter is a value that is read but not written.
- out                     The parameter is a variable that is assigned but not read.
- in out                 The parameter is a variable that can be read or assigned.

**passivate**

Copy an active version of a system object to its passive version.

**passive AD**

An AD in passive store.

**passive object**

A system object in passive store, a passive version.

**passive version**

An object's version in passive store. An object can also have zero or more active versions.

**passive store**

The distributed object filing system for storing system objects on disk. Compare with *active memory*.

**pathname**

(1) A string of names that contains slashes and is a "path" of directories from a point in a directory structure to an entry. BiiN™ uses four kinds of pathnames: relative, node, organization, and full. (2) A series of base names, separated by slashes, that uniquely identifies an element in a form.

**physical address**

A 32-bit address of a physical memory location or memory-mapped device register.

**physical address space**

The 2<sup>32</sup> byte address space used by the BiiN™ hardware.

**physical memory**

A node's semiconductor memory, whether normal RAM (volatile, read-write), battery-backed-up RAM (non-volatile, read-write) or ROM/EPROM/EEPROM (non-volatile, read-only for normal uses). Compare with *active memory*.

**physical terminal**

A video display device with a keyboard. It may also have a pointing device (mouse).

**pipe**

A software-defined object that supports interprocess communication (in one direction only). One process writes to the pipe and the other reads from it. The pipe uses a fixed-size buffer to hold data written by the first process but not yet read by the second process. The writing process will block if the buffer is full, and the reading process will block if the buffer is empty (the processes resume when these conditions no longer hold).

**pointer**

(1) A variable that contains the address of another variable or of a function. (2) In BiiN™ CL, an argument or variable type. A pointer value is a pathname to a passivated object.

**port**

An interprocess communications mechanism consisting of queued data structures that use shared memory and provide communications for processes within a single job. Ports contain messages, blocked processes, or are empty. Ports are the appropriate message mechanism when fast and simple message passing is needed.

**positional notation**

Providing the value of a command argument by specifying the value at the appropriate position in the command's argument list.

**pragma**

A directive to the Ada compiler, embedded in an Ada source file. Pragmas can provide important semantic information, such as how pointers are represented, or whether a sub-program can be called from another language.

**print device**

A device created by an application through which data is spooled or printed directly.

**print service**

The OS service that manages printers.

**printer**

An object that represents a physical printer connected to the system.

**printinfo**

A set of attributes describing the capabilities of a printer.

**procedure**

(1) A program unit in BiiN™ Ada, FORTRAN, or Pascal that is invoked by a call statement. Unlike a function, a procedure does not return a value. (2) In BiiN™ COBOL, a paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division. (3) In BiiN™ SQL, a collection of one or more SQL statements that can be called by a host language module. Procedures are grouped into SQL modules. (4) A program in CP microcode that forms a part of an IOM microcode program.

**process**

The smallest unit of scheduling; a single thread of execution; represented by a processor-recognized object. Processes specify execution environments for running programs.

**process globals**

A data structure that defines the environment in which a process executes. It is a list of ADs associated with the process.

**process preemption**

Forcing a running process to relinquish the processor to another process waiting in the dispatching port. It occurs if the waiting process has a higher priority than the running process and is a preemptive process (has a priority higher than the preemptive threshold).

**processor claim**

The number of time slices available to the processes in a job during each scheduling cycle. When the claim is exhausted, the scheduler terminates the job if it has exceeded its time limit, or obtains more processor claim if it hasn't (allowing the job to continue).

**program**

(1) A complete collection of software modules that are designed to accomplish a given piece of work. There are several kinds of programs: dialogue programs (which accept runtime commands), start-and-go-programs (which accept runtime commands), application programs, and system utilities. A program may be invoked interactively from the keyboard or batched in a BiiN™ CL script. An executable program is the linked version of a program.  
(2) In Ada, a program is composed of a number of compilation units, one of which is a subprogram called the main program. Execution of the program consists of execution of the main program, which may invoke subprograms declared in the other compilation units of the program.

**program building service**

The OS service that provides support for building programs: creation, execution, and debugging.

**program object**

The root of a network of objects that comprise a program. A program object is created by the linker and referenced by a program AD. The linker stores the program AD in a directory after creating the program. A program consists of a program object, a global debug table (GDT), an outside environment object (OEO), and one or more domain objects.

**Program Services**

The OS service area that provides support for concurrent programming, program building, and resource control.

**protection service**

The OS service that provides packages to manage users, IDs and authority lists.

**protection set**

List of IDs and associated access rights. A protection set is associated with an ID, and a caller must hold an ID that matches one in the protection set, with the appropriate rights, before the caller can access that ID.

**public data object**

An object containing data that can be referenced from other domains (domains that have an AD to the public data object in their static data objects.)

**pull-down menu**

A menu that is activated by a mouse and which appears only on explicit request of the user. After a user has selected *menu items* from the menu, the program can determine the menu choices by calling the appropriate *terminal access method*.

**R**

**range**

In BiiN™ CL, an argument or variable type. Range values are composed of two integers that are separated by a double period (`lower_integer..upper_integer`).

**rank**

(1) Default order in which spool files will print. (2) Default order in which subform group instances will be displayed in a form.

**read rights**

A type right required for many devices and opened devices, in order to read data using an I/O access method. Read rights rename use rights.

**read rep rights**

Rights bit that must be 1 to read an object's representation. ADs and virtual addresses contain read rep rights.

**real-time job**

A job that is executed in real time because it cannot wait for objects to be brought into memory or for another job to finish with a processor before executing. Real-time jobs have very high priority and infinite processor claim. They run in frozen memory, and are not subject to the scheduling process. If they block for I/O, the hardware reschedules them immediately.

**real type**

A simple data type that represents a floating-point number.

**record**

(1) In the BiiN™ OS, an element of a structured file. Each record in a structured file has a unique *record ID* that can be used to access the record. A record has a *format* that is either *fixed-length* or *variable-length*. (2) In COBOL, the most inclusive data item. The level-number for a record is 01. A record may be either an elementary item or a group item. (3) In BiiN™ Pascal, a predefined type. (4) The unit of information in an object module. The BiiN™ Systems Object Module Format specifies about a dozen records, each of which contains specific information about the object module. These records are a header record, various symbol and object definition and reference records, and an end-of-module record.

**record access method**

An access method that transfers data in record-like units, in various access modes.

**record type**

A structured data type consisting of a fixed number of components (fields), possibly of different types, that are referenced by means of identifiers.

**recovery agent**

Process provided on each node by the OS that detects I/O processor failures and maintains a

table of existing I/O messages. Device managers keep this list current by calling `DD_Support.Register_IO_message` each time they create an I/O message.

**region**

(1) An area within a form. Valid regions are: the form as a whole, a subform, a group, a screen field or an enumeration. (2) A linear address space is partitioned into four 1-gigabyte system objects called regions. Region 0 contains static data, region 1 contains instructions, region 2 contains the stack, and region 3 is used by the operating system. Calling another domain in the current subsystem can change regions 0 and 1. Calling a domain in another subsystem can also change region 2. If a region contains less than one gigabyte, then the linear address space contains invalid parts. Reading or writing with an invalid linear address raises `System_Exceptions.length_violation`.

**relative file**

A structured file whose records are organized in an array of fixed-size record slots that may or may not contain information. A relative file can be read or written in any order.

**relative pathname**

A pathname with no leading slashes. The BiiN™ OS evaluates relative pathnames relative to a specific directory; by default, the current directory.

**rep rights**

Rights bits required to read or write an object's representation. ADs and virtual addresses contain rep rights. There are two rep rights: read rep rights and write rep rights.

**representation type**

An object characteristic that specifies which of the four kinds of object representation is used: embedded, simple, simply-paged, or bi-paged.

**report**

A printed or displayed document containing labelled data, often presented in columns and hierarchical groups with subtotals and totals.

**report description**

A DDef that describes the format of a report and the data to be printed in it.

**report service**

The OS service that manages reports.

**reservation service**

The OS service that supports the reservation of devices for exclusive use by a session.

**resource priority**

A process's resource priority. When an interactive or batch process requests the use of a resource (for example, a disk), the process's priority is raised to the sum of its base, bias, and resource priorities (but still in the range 1 to 10).

**resource service**

The OS service that supports resource control and accounting.

**rights**

Bits in an AD that control access to a system object. There are two kinds of rights: rep rights and type rights. Rep rights are required to read or write an object's representation. Rep rights are checked and enforced by the CPU. Type rights are required to invoke certain type manager calls with an object. The interpretation of type rights varies for different object types. Type rights are checked and enforced by type managers. Rights are not interpreted in null ADs.

**rights mask**

A record representing rights to be checked, added, or removed in an AD.

**running priority**

The priority at which an interactive or batch process is currently running. It fluctuates between the process's base priority and the priority of the resource the process requested most recently.

**runtime command**

A command that is processed by a program, using the command service. Runtime commands are defined in command sets. Command sets can be stored in the program's outside environment object (OEO), or as separate objects.

**S****scalar type**

A data type whose variables have a single value; also called a *simple type*.

**scheduler**

A collection of hardware and software entities that together schedule the execution of jobs (and thus processes). The scheduler seeks to maximize the use of system resources by scheduling processors, physical memory, and I/O devices.

**scheduling service object (SSO)**

An object that determines the type of scheduling a job receives by specifying the job's service class, priority, time slice, memory type, initial age, and age factor. An SSO is associated with a job when the job is invoked. The system administrator is responsible for creating different types of SSOs and controlling access to them, thus controlling the type of service granted to different jobs.

**scheduling service**

The OS service that manages scheduling of jobs and processes.

**scope**

(1)The part of a form in which an element exists and can be referenced. A form element is in a form, or contained in a subform, a group, or a pile, i.e., in another form element. At any one time the editing scope extends only to elements located directly in the form, or directly in a subform or group, or directly on a pile. Only elements in the editing scope can be edited. (2)The portion of a program in which a program entity exists and can be referenced.

**scrolling**

Moving a view up or down in its frame buffer in order to see a different part of the frame buffer. Also called *panning*.

**semaphore**

An object for controlling and synchronizing access to data that may be shared by concurrent processes.

**sequential file**

A structured file whose records are organized in the sequence they are physically written. A sequential file must be read in exactly the same order that it was written.

**service**

A logically related set of packages or other program modules. A service provides completely procedural solutions to problems. Applications call services on behalf of users, but users do not directly interact with services. Compare with *tool* and *utility*.

**service class**

Denotes the general class of service a job is to receive. Four service classes are defined: realtime, time-critical, interactive, and batch.

**service area**

A logically related set of services.

**session**

A grouping of jobs belonging to one instance of a user's interaction with the system. A session typically contains several jobs. A session is usually an interactive logon/logoff period, but can also be the running of a batch command file.

**set**

In BiiN™ Pascal, a predefined type.

**simple object**

An object representation that fits entirely into all or part of one memory page. A simple object's size ranges from 64 bytes to 4K bytes.

**simply-paged object**

An object representation that requires multiple memory pages, but with a page table that fits entirely into all or part of a memory page. Compare with *bi-paged object*. A simply-paged object's size ranges from 8K bytes to 4M bytes.

**single-activation model**

An activation model that activates an object only in a particular home job (for local objects) or home node (for global objects); another job or node that attempts to activate the object instead activates a *homomorph*, a token object that stands in place of the actual object.

**spin lock**

A synchronization device used during the processing of I/O messages with calls that raise and restore interrupt handler priority levels.

**spool file**

A buffer maintained by a spool queue that holds data from print device objects which is to be printed.

**spool queue**

A spool device that must be installed before anything can be printed.

**spool service**

The OS service that manages spoolers.

**SSO priority**

The priority defined in a job's SSO.

**stable store**

Non-volatile RAM storage that is used to optimize I/O throughput from active memory to disk. Using stable store, writes to disk can be delayed indefinitely, which greatly reduces I/O access time.

**stack**

System object that provides a stack of frames that each contain the state of a particular subprogram call.

**standard kernel image**

Factory-supplied OS preconfigured to run on a system disk and a console terminal.

**starter image**

A self-contained, linked image that does not need a secondary store (such as a disk) for operation, and which is booted into memory from a distribution channel (such as a tape) for the sole purpose of executing certain system utilities to prepare the physical system to be operable under an OS standard kernel.

**statement**

(1) A program construct that defines actions to be performed by the program. (2) A source program construct at which a breakpoint can be set when using the BiiN™ Application Debugger. In general, any construct that is considered a statement in the formal definition of the language is also considered a statement by the debugger. However, the following constructs are *not* considered statements for debugging purposes:

- Any declaration in any language (or definition in C) other than a variable declaration (definition) involving dynamic initialization or a subprogram declaration (definition).
- Any declaration (as opposed to definition) in C.

In addition, subprogram declarations are *always* considered statements by the debugger, regardless of their treatment by the source language.

**static data object**

System object that contains the data for a particular domain. This object represents the static data region (region 0).

**storage resource object (SRO)**

An object used to allocate other objects. An SRO provides access to available memory and to available object table entries. The SRO used to allocate an object determines the object's memory type and whether the object is local or global. Each job has a local SRO, used to allocate objects local to the job. Each node has two global SROs, one for normal memory and one for frozen memory.

**stream file**

A stream of bytes that allows random byte positioning. This UNIX-like file organization is useful if you simply want to read and write bytes.

**string**

(1) In BiiN™ CL, an argument or variable type. String values are sequences of characters, enclosed in single or double quotation marks (e.g., 'string' or "string"). If there are no spaces, tabs, or linefeeds in a string, the quotation marks are optional. One string subtype is *enumerated*, for which a set of allowable string values is defined. (2) In standard Pascal, a sequence of one or more characters, enclosed by apostrophes, representing a value of type CHAR (if a single character) or of type PACKED ARRAY [1..n] OF CHAR, where *n* is a positive integer equal to the number of array elements. (3) In BiiN™ Pascal, STRING is a reserved word, used as a type denoter.

**string list**

In BiiN™ CL, an argument or variable type. String list values are sets of strings, enclosed in parentheses (e.g., (string1, string2, string3)). The string values may be separated by spaces, tabs, or commas. If a string list contains just one string value, the parentheses are optional.

**structured file**

A file containing records of either fixed or variable length. Structured files optionally can have indexes. Structured files are useful if you need a way to maintain record structures. Structured file I/O is typically accomplished using record I/O. A structured file can have one of these organizations: *clustered*, *hashed*, *relative*, *sequential*, or *unordered*.

**subnet**

Informal term for subnetwork.

**subnet service**

The OS service that provides network-independent communication between nodes within a subnet.

**subprogram**

(1) A procedure, function, or subroutine written in any BiiN™ programming language. (2) In a form, a processing routine or key catcher.

**subprogram type**

An Ada subprogram specification that can have alternate bodies.

**subtransaction**

A transaction that is contained within another transaction.

**subsystem**

One or more domains that share a common stack (that is, they have a single subsystem ID).

**Support Services**

The OS service area that provides common definitions and utility packages that are of use to all other services.

**swapping volume set**

A volume set providing external storage for virtual memory.

**symbolic link**

A symbolic link contains a pathname. Symbolic link evaluation retrieves whatever AD is stored with that pathname.

**System Configuration Object (SCO)**

A sequence of configuration commands that attach and start configurable objects during the booting of the system to put the configurable objects into operable states.

**system SCO**

A sequence of configuration commands that attach and start those configurable objects (typically hardware components) required to complete node initialization of the OS.

**T****tag bit**

A 33rd bit that tags each memory word and indicates whether the word contains a valid AD. A tag bit of 1 indicates a valid AD. A tag bit of 0 indicates a data word or a null AD.

**tail object**

An object that must be *attached* to a configurable object before it can become functional.

**temporary file**

A file that is unnamed when created and exists only for the duration of the current job (unless explicitly named and saved).

**terminal access method**

One of two currently supported methods for procedural interaction with a terminal: character (`Character_Display_AM`), or graphics. Contains calls to access the screen and input devices.

**terminal service**

The OS service that manages terminals and windows.

**time-critical job**

A job that has less stringent time constraints than a realtime job. Time-critical jobs have the same priority as realtime jobs, but limited processor claim (they are rescheduled in round-robin fashion when a time slice expires). They need not run in frozen memory, since their time constraints can tolerate page faults.

**time limit**

The total processing time available to a job (and its descendant jobs). When the processes in a job exhaust the job's processor claim, the scheduler terminates the job if it has exceeded its time limit, or obtains more processor claim if it hasn't (allowing the job to continue).

**time slice**

The amount of processing time assigned to each process in a job in each dispatching cycle. (It does not include time spent on interrupts, processor preemption, or waiting at a port or on a semaphore). When a process exhausts its time slice, it is generally redispached with the same time slice value. However, each job has a processor claim value that determines the

total processor time available to all the processes in the job. When the job's processes have used  $n$  time slices and exhausted the processor claim, the job is reexamined by the scheduler and either terminated or granted additional processor claim (and the processes resume execution).

**timing service**

The OS service that manages system time, timed requests, time computations, and time format conversions.

**TM concurrent programming service**

The OS service that provides concurrent programming support for advanced type managers.

**TM object service**

The OS service that provides object and memory operations for building advanced type managers.

**TM transaction service**

The OS service that manages transactions within a type manager.

**transaction**

A system object that groups related operations so that either all the operations succeed, or all are aborted and undone.

**transaction service**

The OS service that provides calls to start and resolve transactions.

**transaction stack**

A per-process stack of transactions. The top transaction on the stack is the *default transaction* for any transaction-oriented operations.

**transport service**

The OS service that provides network-independent communication between nodes.

**type**

A label that distinguishes one kind of entity from another. The type of an entity typically determines the entity's allowed values, allowed operations, and representation.

**type definition object (TDO)**

An object that represents one type of system object. A TDO contains type-specific attribute entries for the type. These attribute entries are inherited by all objects of the type.

**type manager**

A program module that conceals the representation of an object type and that provides all basic operations for the object type. One module may act as a type manager for more than one object type. Several type managers that work closely together to manage some aspect of the system (for example, filing) constitute a "service".

**type rights**

Rights bits required to invoke certain type manager calls with an object. ADs and virtual addresses contain type rights. There are three type rights: use rights, modify rights, and control rights. The interpretation of type rights varies for different object types. A type manager may also rename the type rights that it uses.

**type-specific attribute**

An attribute that can only be defined once for an object type. The attribute entry is stored in the object type's TDO. All objects of the type inherit the attribute entry.

**Type Manager Services**

The OS service area that provides packages to build *type managers*, software modules that implement new object types and their attributes.

**U****unique identifier (UID)**

An identification number that is never changed or reused once it is assigned to a particular entity. A UID securely identifies the entity for all time and all systems. For example, each BiiN™ node is assigned a UID.

**unbounded global object** A system object that is not local to any job and that has an unbounded lifetime. An unbounded global object can be reclaimed by global garbage collection when it is no longer accessible via any AD.

**unordered file**

A *structured file* whose records are organized according to available free space.

**use rights**

One of three type rights. By convention, use rights are required to read an object's state.

**user**

(1) In general, one entity using the services of another. For example, a program is a user of system services. (2) The person sitting at the terminal issuing commands and entering data.

**user interface**

The part of a program that accepts user input, displays messages, and creates output.

**user SCO**

A sequence of configuration commands that attach and start configurable objects (typically software modules) of a configuration that are not required to complete node initialization of the OS.

**utility**

Program or BiiN™ CL script that is invoked interactively from the CLEX > prompt. It is supplied by the system to perform a particular service for some group of users. Developers may create new utilities. A utility may or may not have *runtime commands*.

**utility service**

The OS service that provides system definitions, texts, string lists, and long integers.

## V

**variable**

(1) A datum whose value can change during program execution. (2) In CLEX, a named and typed datum containing a value; also called an *environment variable*. A variable's *mode* is either "read-only" or "read-write". A variable's type is one of: *boolean*, *integer*, *pointer*, *range*, *string*, or *string list*. A variable may be read (and, if "read-write", set) either interactively (using the *built-in commands* for variables: `create.variable`, `list.variable`, `remove.variable`, `set.variable`) or procedurally (using the *environment service*). The scope of a variable may be either *global* or *local*. Passivated variables are stored in *variable groups*; some groups are predefined for use by CLEX, programs, and scripts. Variables are stored and passivated with the `manage.variable_groups` utility. (3) In FORTRAN, the term "variable" does not include array elements. (4) In COBOL, a data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

**variable group**

A group of BiiN™ CL (environment) variables, associated with one or more BiiN™ services, programs, or applications. A variable in a variable group is identified by the group name, a period, and the variable's name. For example, CLEX uses the `cli.` (command line interface) variable group, which contains the current directory's pathname, command input prompt string, and so on.

**version**

(1) In general, a variation of a file that reflects the state of its development. (2) In the BiiN™ Software Management System, a member of a version group. A version captures a point in the evolution of a file (object).

**view**

(1) In BiiN™ SQL, a view is a named query that may be used as a table. In effect, views are virtual tables derived from the underlying base tables. They do not take up physical space. (2) A copy of an image module that makes available only a subset of the procedures defined by the image module from which it was derived. Executable programs may be linked to views, much like image modules and linker libraries. Views are a form of information hiding. (3) The visible part of a frame buffer.

**virtual address**

A location within an object, given by a 32-bit byte offset and an AD to the object. A virtual address can also be null, referencing no object. An active virtual address contains two words aligned on a word boundary. The first word is the offset; the second word is the AD.

**virtual address space**

Up to  $2^{58}$  bytes simultaneously accessible: Up to  $2^{32}$  bytes in each of up to  $2^{26}$  system objects.

**virtual memory**

A memory management feature that supports a logical view of memory (for example as a collection of varying-size objects) that is distinct from the physical address space. Virtual memory requires hardware address translation, which is provided by the CPU. Virtual memory also implies support for logical memories larger than the physical memory, with the obvious problems being avoided by juggling parts of memory to and from disk.

**virtual terminal**

A device which, to an application, appears indistinguishable from a physical terminal. It provides a screen-like drawing space for the output of characters or graphics, and a keyboard and mouse for input.

**volume**

Logical storage area for storing files and objects. Volumes are members of volume sets.

**volume number**

A sequential number assigned to each volume in a volume set when created that identifies it relative to other volumes on the volume set.

**volume set**

A logical disk containing volumes used to store files and objects. Volumes of volume sets can span multiple physical disk devices.

**volume set name**

Name assigned when a volume set is created. It must be unique on all disk volumes that contain the volume set's volumes.

**W**

**window**

A portion of a terminal screen in which I/O can occur.

**word**

A unit of memory containing 32 value bits and an associated tag bit. A word is always aligned on a 4-byte boundary. Value bits in a word are numbered from 0 to 31.

**work queue mechanism**

A work queue data structure and two associated interrupt handlers designed to aid device driver writers in maintaining and initiating I/O requests for directly-connected devices.

**working set model**

A model for the reclamation of primary memory pages. The working set of a job is dynamically defined as the set of primary memory pages referenced by the job in the last time quantum,  $T$ , measuring backwards from a given time  $t$ . Every  $T$  time units the scheduler determines the working set for each running job. Any pages that have not been accessed in that time period are returned to a pool of free pages.

**write rights**

A type right required for many devices and opened devices, in order to write or change data using an I/O access method. Write rights rename modify rights.

**write rep rights**

Rights bit that must be 1 to write an object's representation. ADs and virtual addresses contain write rep rights.



\$OEO V-4-4  
\$status V-4-4

2-space view I-1-9

configure utility VII-7-9

\_AD I-3-2

\_VA I-3-2

## A

Access descriptor I-3-2, II-2-3, VII-1-12

Access methods IV-1-2

Access modes

Indexed-random IV-9-7

Indexed-sequential IV-9-5

Physical-random IV-9-4

Physical-sequential IV-9-3

structured files IV-9-3

Access rights I-1-15

Activation model II-3-12

choosing one II-3-14

Active-only object II-3-6

AD I-1-13, I-3-2

alias VII-1-17

passive VII-1-17

retrieving III-1-8

AD activation II-3-5

Address space

protection I-1-10

Age factor VI-3-7

Alias AD II-3-8

Alias entry III-1-2

Allocating a buffer I-3-6

Alternate index IV-8-15

Application program IV-4-2, V-2-2

clear and close a form V-2-18

closing record access to a form V-2-18

commands V-2-3

creating a file-sorting procedure V-2-26

creating a menu group V-2-13

creating a pipe V-2-27

creating a report-printing procedure V-2-27

creating a sort DDef V-2-25

creating a window V-2-11

defining a file V-2-20

defining a form V-2-17

defining a message V-2-16

defining a report V-2-22

defining the invocation command V-2-11

design V-2-4

displaying a form for user input V-2-18

displaying a message V-2-16

displaying a read-only form V-2-20

fetching a value from a form V-2-18

form and report DDefs from a file's DDef V-2-5

forms V-2-4

getting a menu selection V-2-14

getting a report's DDef V-2-23

getting and opening a form V-2-17

initializing a report V-2-23

installing and enabling a menu group V-2-13

menus V-2-3

messages V-2-3

modifying a form V-2-17

opening a file V-2-20

opening a report's input device V-2-22

opening a report's output device V-2-23

printing a report V-2-24

processing the invocation command V-2-11

producing a report from a sorted file V-2-25

producing a report V-2-21

reading a data record from a form V-2-18

record access to a form V-2-17

relation to Human Interface Services V-1-2

reports V-2-4

spawning concurrent processes V-2-27

storing data into a form's field V-2-19

updating a file V-2-20, V-2-21

updating an indexed file V-2-20

using a menu selection V-2-14

waiting for concurrent processes V-2-28

windows V-2-3

writing a data record into a form V-2-19

Attribute

Ada VII-4-3

ID VII-4-5

implementation of VII-4-7

instance VII-4-5

new VII-4-5

Attributes

OS IV-1-5

window IV-5-6

Authority list III-3-2

access, illustration III-3-2

calls to manage III-3-9

changing an object's III-3-11

changing the default III-3-10

creating III-3-9

multiple objects sharing, illustration III-3-6

Passive Store Protection VII-1-18

using the default III-3-9

Authority-based protection III-3-3

Authority\_List\_Mgt III-3-2

## B

Banner page IV-6-7

Base priority VI-3-8

Batch job VI-3-6

Batch process VI-3-5

Booting a node VII-7-3

Buckets IV-7-4

Buffer IV-2-3

allocating I-3-6

Building

indexes IV-8-14, IV-8-15

Byte\_Stream\_AM.Ops IV-4-16

## C

Call Stack

trimming VII-5-5

Canonical pathname III-1-7

Character display devices IV-5-2

- Character display I/O
  - input model IV-5-4
  - output model IV-5-4
- Character fields V-6-5
- Character terminal user agent IV-4-15
- Character terminals IV-4-3
- Character\_Display\_AM IV-5-2, V-5-3
- Character\_Display\_AM.Ops IV-4-18, IV-5-2
- Character\_Terminal\_Mgt IV-4-2
- CL variables V-4-2
- CL\_Defs V-3-2
- Clearinghouse I-1-11, III-1-5
- Cluster IX-1-8
- Command
  - alternatives V-3-12
  - arguments V-3-8
  - control options V-3-9
  - file V-3-4
  - history V-3-3
  - named arguments V-3-8
  - names V-3-7
  - positional arguments V-3-8
  - sets V-3-7
  - Unix conventions V-3-4
- Command language variables V-4-2
  - active memory buffers V-4-3
  - active memory V-1-9
  - buffers V-1-10
  - built-in .variable commands V-1-10
  - concepts V-1-9
  - creating a variable V-4-8
  - creating V-1-11
  - displaying variable names V-4-6
  - evaluation of V-1-10
  - job variables V-1-10, V-4-3
  - local variables V-1-10, V-4-3
  - passive store V-1-9
  - reading a value as a string V-4-7
  - reading a value V-4-5
  - reading V-1-11
  - removing a variable V-4-8
  - setting a value from a string V-4-7
  - setting a value V-4-5
  - setting V-1-11
  - variable groups V-4-3
- Command name space III-4-2
- Command service V-1-6, V-1-8, V-3-2
  - program-defined commands V-1-7
  - types of commands V-1-7
  - use of command definition V-1-7
- Command\_Execution V-1-8, V-3-2
- Command\_Handler V-3-2
- Commands V-1-7, V-3-2
  - built-in control V-3-5
  - built-in runtime V-3-5
  - defining help messages V-3-11
  - defining invocation V-3-13
  - defining runtime V-3-4, V-3-14
  - defining V-1-8, V-3-10
  - entering runtime V-3-12
  - executing from program V-3-16
  - processing arguments V-3-15
  - processing runtime V-3-16
  - reading as text V-3-16
  - reading from devices V-3-4, V-3-11
  - reading invocation V-3-14
  - record access V-3-4, V-3-12
  - runtime V-3-7
  - startup file V-3-12
  - syntax V-1-7
  - types of V-3-4

- Concurrency IV-1-3, IV-10-3
- Concurrent program VI-2-2
- conditional P VI-1-17
- Configurable Object (CO) VII-7-5
- Configurable object VII-7-6
- Configuration VII-7-2
- Configuration attribute VII-7-5
- Configuration.Ops.Attach VII-7-6
- Configuration.Ops.Start VII-7-7
- Control break V-8-5
- Control group V-8-5
  - field V-8-5
  - footings V-8-9
  - headings V-8-9
  - hierarchy V-8-5, V-8-8
- Cpu scheduling VI-3-3
- Create call, example VII-3-9
- create.form V-6-4, V-7-6
- create.report V-8-7
- create.standard\_form V-1-5
- create.standard\_report V-1-6
- create.variable V-1-10
- Creating
  - files IV-8-12
- Current record pointer (CRP) IV-9-2
- Cursor IV-5-4

## D

- Data
  - Abstraction VII-1-2
- Data areas
  - primary IV-7-3
  - secondary IV-7-3
- Data definition (DDef)
  - defining an index key DDef IV-8-12
  - record DDef IV-8-6
- Data fields V-6-9
- Data\_Definition\_Mgt V-1-4, V-1-11, V-1-14, V-1-16, V-8-7
- Date field V-6-6
- DDefs V-1-4
- DDF\_UTILITY\_Support V-1-4
- Deadlock IV-10-10, VI-1-19
- Deallocation
  - Deallocating Passive Versions VII-2-10
  - Explicit VII-2-10
  - Job Termination VII-2-10
- Debug object VI-1-7
- Default formats V-6-7
- Default transaction II-4-3
- define.standard\_form V-1-14
- define.standard\_report V-1-16
- Definition V-1-4
- destination V-6-20
- Destination path register V-7-10
- Destroy call, example VII-3-11
- Device IV-1-2
  - closing IV-2-2
  - opening IV-2-2
- Device class IX-1-4
- Device driver IX-1-4
- Device manager IX-1-3, IX-1-4
- Device object IX-1-3
- Direct mode IV-6-3, IV-6-8
- Direct printing IV-6-9
- Directories
  - protecting III-1-5
- Directory III-1-2
  - creating a directory III-2-2

- deleting a directory entry III-2-4
- listing directory contents III-2-5
- node's default directories III-1-8
- retrieving a directory AD from process globals III-2-7
- retrieving an AD in a directory III-2-4
- standalone III-1-10
- storing a directory entry III-2-3
- transactions and III-1-10
- using a pattern to filter directory contents III-2-7
- valid entry names III-1-2

- Directory entry
  - alias entry III-1-2

- Dirty read II-4-5
- Dispatching VI-3-4
- Distributed system I-1-8
- Domain object VI-1-4

## E

- edit.form V-1-5, V-1-14, V-6-4
- edit.key\_map V-1-6
- edit.report V-1-6, V-1-16, V-8-7
- End-of-file IV-2-3, IV-9-9

- Entry name
  - valid names III-1-2

- Enumeration
  - overlaid V-6-8
  - scattered V-6-8

- Environment service V-1-8
- Environment\_Mgt V-1-9, V-4-2
- Error II-5-2

- Error decision V-8-13
- Evaluation III-3-6
  - during Retrieve, example, illustration III-3-8
  - during Retrieve, illustration III-3-7

- Event VI-1-12
  - establishing a handler for VI-2-10
  - handler VI-1-12
  - signaling an VI-2-9
  - signaling VI-1-12
  - table of local event values VI-1-16
  - waiting for VI-2-11

- Event clusters VI-1-12
  - global VI-1-12
  - local VI-1-12, VI-1-16

- Event\_Mgt VI-1-12, VI-2-3

- Example program
  - data files V-2-6
  - menus V-2-6
  - overview V-2-5
  - processing V-2-9
  - setup V-2-9
  - source code files V-2-8
  - termination V-2-10

- Escape character V-8-9

- Exception II-5-6
  - recovering from I-3-8
- Exception handler I-3-8

## F

- Fault tolerance I-1-5
  - OS support for I-1-5

- File access modes
  - indexed-random IV-9-19
  - indexed-sequential IV-9-20
  - physical-random IV-9-17
  - physical-sequential IV-9-18

- File descriptors
  - index descriptors IV-7-9
  - logical file descriptor IV-7-8
  - physical file descriptors IV-7-8
- File organization IV-7-4

- Files
  - creating IV-8-12
  - nontransaction-oriented files IV-10-3
  - primary data area IV-7-3
  - secondary data area IV-7-3
  - stream IV-7-2
  - structured IV-7-2

- Form V-6-2
  - annotated V-1-13
  - CL variables V-7-3
  - contraction V-6-13
  - creating V-1-13
  - defining V-1-14
  - elements V-1-14
  - example V-1-12
  - executable V-6-3, V-7-2
  - initial values V-7-8
  - using V-1-13

- Form DDef V-6-4

- Form description V-7-2

- Form elements V-6-4

- Form service V-1-12, V-1-13

- Form sheet V-6-5

- form.decimal\_character V-8-10

- Form\_Handler V-1-13

- Format string V-7-4, V-8-9

- Forms V-6-12

- Frame buffer IV-5-2, IV-4-2

- Frozen memory I-1-11

- Full pathname III-1-6

## G

- Garbage Collection
  - Global VII-2-10, VII-2-13
  - Local VII-2-10

- setting/changing local VII-5-5

- starting local VII-5-5

- stopping local VII-5-6

- Garbage collector (GCOL) VII-2-11

- Global debug table (GDT) VI-1-3

- Graphics terminal user agent IV-4-15

- Graphics terminals IV-4-3

- Group V-6-9

- deployed V-6-10

- instances V-7-11

## H

- Handler object VI-1-8

- Handling recoverable exceptions I-3-8

- Head object VII-7-6

- High-level scheduling VI-3-3

- Human Interface Services V-1-2

- features V-1-4

- utilities V-1-4

**I**

I/O message IX-1-5  
 I/o scheduling VI-3-9  
 ID III-3-3  
     calls to manage III-3-9  
     changing an object's owner III-3-11  
     contents of III-3-4  
     illustration III-3-4  
     record IV-7-3  
     type rights for III-3-5  
     user ID III-3-11  
     when first created III-3-9  
 ID list III-3-3, III-3-4  
     illustration III-3-5  
 Identification Mgt III-3-2  
 Image module VI-1-2  
 Import I-3-6  
 Importing operators I-3-5  
 Incident II-5-2, II-5-4  
 Incident code II-5-4  
 Independent transaction II-4-6  
 Index  
     b-tree IV-7-4, IV-8-3  
     defining a key DDef IV-8-12  
     hashed IV-7-4, IV-8-3  
     index descriptors IV-7-9  
     organization key IV-7-7, IV-8-3  
     performance IV-8-5  
 Indexed-random access IV-9-7  
 Indexed-sequential access IV-9-5  
 Indexes  
     alternate IV-8-15  
     building IV-8-14, IV-8-15  
     organization IV-8-14  
 Initial age VI-3-7  
 Input focus IV-4-2  
 Instruction object VI-1-6  
     Optimized Handling of VII-2-13  
 Interactive job VI-3-6  
 Interactive process VI-3-5  
 Interlanguage calling conventions V-6-13  
 Interprocess communication VI-1-12  
 Interrupt reply procedure IX-1-6  
 Is call, example VII-3-8

**J**

Job VI-1-9, VI-2-2  
     and multiprocessing I-1-4  
     batch VI-3-6, VII-2-12  
     interactive VI-3-6, VII-2-12  
     real-time VI-3-6, VII-2-12  
     time-critical VI-3-6, VII-2-12  
 Job variables V-1-10  
 Job\_Admin VI-1-11  
 Job\_Mgt VI-1-9, VI-1-11, VI-3-5

**K**

Key catcher V-7-11  
 Key catchers V-6-15  
 Key list V-6-19, V-7-11

**L**

Line end decision V-8-13  
 list.variable V-1-10  
 Local Address Space  
     getting information about VII-5-6  
 Local variables V-1-10  
 Locking area II-3-15  
 Locks  
     acquiring IV-10-6  
     escalating IV-10-7  
     lock modes IV-10-5  
     opened devices IV-10-11  
     releasing IV-10-7  
     transactions IV-10-2  
 Logging I-1-6  
 Logically delete VII-2-10  
 Long integer II-1-4  
     illustration II-1-2  
     using a literal II-1-7  
 Long-term logging IV-10-11  
 Long\_Integer\_Defs II-1-8  
 Low-level scheduling VI-3-4

**M**

make.script V-1-7, V-3-4  
 manage.commands V-1-5, V-3-2  
 manage.messages V-1-6  
 manage.variable\_group V-1-5, V-1-9  
 Master AD II-3-7, III-1-3  
     restrictions II-3-8  
     transferring mastership II-3-9  
 Medium-level scheduling VI-3-5  
 Memory  
     active VII-1-16  
     Compaction VII-2-13  
     control VII-5-2  
     monitor VII-5-2  
     Pool VII-2-4  
     Primary VII-2-5  
     Secondary VII-2-5  
     shared over distributed system I-1-3  
     Virtual Paging VII-2-12  
 Memory address  
     overlay I-3-5  
 Memory Allocation  
     Frozen VII-2-7  
     Normal VII-2-7  
 Memory scheduling VI-3-9  
 Memory type VI-3-7  
 Menu V-1-11, V-5-2  
     checking an item V-5-7  
     defining V-1-11  
     getting a menu selection V-5-6  
     group V-1-12  
     help V-1-12  
     interactions V-1-11  
     item numbers V-5-3  
     item V-1-12  
     reading V-1-12  
     selecting V-1-12  
     title V-1-12  
 Menu group V-5-2  
     creating a DDef V-5-4  
     defining V-5-4  
     disabling V-5-7  
     enabling V-5-6  
     installing V-5-5

removing V-5-8  
 Menu service V-1-11  
 Menus IV-4-13  
 Message II-5-3  
 Multiple activation II-3-12

**N**

N-space view I-1-9  
 Name space III-4-2  
     changing command name space in process globals III-4-4  
     changing command name space in user profile III-4-4  
     command name space III-4-2  
     creating III-4-3  
     reading directories in a III-4-2  
 Name\_Space\_Mgt III-4-2  
 Named notation V-3-8  
 Naming domain III-1-6  
 Network of paths V-6-20, V-7-9  
 next\_path\_element V-6-14, V-6-20, V-7-10  
 Non-stop computing I-1-5  
 Nontransaction-oriented files IV-10-3  
 Null element V-6-9  
 Number  
     record IV-7-3  
 Numeric fields V-6-6

**O**

Object I-1-13, II-2-2, VII-1-2  
     Activation VII-2-12  
     active-only VII-3-8  
     Attributes VII-1-6  
     Bipaged VII-2-8  
     Countable Global VII-2-9  
     debug VI-1-7  
     Descriptors VII-2-5  
     domain VI-1-4  
     Embedded VII-2-8  
     generic II-2-4, VII-1-14  
     Global VII-2-9  
     handler VI-1-8  
     How Objects Work VII-1-5  
     instruction VI-1-6  
     Local VII-2-9  
     outside environment VI-1-4  
     Paged VII-2-8  
     program VI-1-3  
     Protection VII-1-6  
     public data VI-1-6  
     Simple VII-2-8  
     Sizes VII-1-5  
     stack VI-1-6  
     static data VI-1-5  
     Table VII-2-5  
     type VII-3-2  
     Types VII-1-5  
     Unbounded VII-2-9  
     Why use VII-1-2  
 Object activation II-3-4  
 Object passivation II-3-5  
 Object tree II-3-9  
 Object versions II-3-3  
 Opened device IV-1-2  
 Option field V-6-7  
 Organization index IV-8-14  
 OS service  
     building a new I-1-13

Outside environment object (OEO) VI-1-4  
 Overflow  
     recovering from I-3-7  
 Overlapped windows IV-4-7  
 Overlay I-3-5

**P**

P VI-1-17  
 Package VII-4-3  
     type-specific instance of VII-4-3  
 Package-level variable I-3-3  
 Page  
     footer V-8-4  
     Frame VII-2-4  
     header V-8-4  
 Page body area V-8-3, V-8-4  
 Page footings V-8-9  
 Page heading V-8-9  
 Page table (PT) VII-2-8  
 Page table directory (PTD) VII-2-8  
 Paired calls I-3-9  
 Passivation dependencies II-3-6  
 Passive AD II-3-7  
     as universal identifier II-3-10  
 Passive object  
     characteristics II-3-10  
     copying II-3-20  
     creating II-3-16  
     destroying II-3-19  
     getting information II-3-22  
     lifetime II-3-9  
     requesting an update II-3-18  
     updating II-3-18  
 Passive store II-3-1, VII-1-16  
     default behavior II-3-15  
     for generic objects II-3-10  
     transaction support II-3-14  
     type manager support II-3-15  
 Passive store attribute II-3-15  
 Physical terminal IV-4-2  
 Physical-random access IV-9-4  
 Physical-sequential access IV-9-3  
 Pile V-6-10  
 Pipe VI-1-14, VI-1-19  
     connecting processes with a VI-2-12  
 Pipe\_Mgt VI-1-15, VI-2-3  
 Port mechanism I-1-11  
 Positional notation V-3-8  
 Pragma  
     bind VII-4-5  
     package\_type VII-4-6  
     package\_value VII-4-6  
 Pragma bind VII-3-8  
 Primary data area IV-7-3  
 Print  
     area IV-6-4  
     delays IV-6-7  
     device IV-6-3, IV-6-8  
     file IV-6-8  
     position IV-6-4  
     termination message IV-6-7  
 print.file V-8-11  
 Printer IV-6-12  
     adding IV-6-12  
     list IV-6-3  
     type IV-6-5  
 Printinfo IV-6-5, IV-6-12  
 Priority VI-3-4

- base VI-3-8
- resource VI-3-8
- running VI-3-8
- SSO VI-3-8
- Process I-1-11, VI-1-9, VI-2-2
  - batch VI-3-5
  - creating a VI-2-5
  - getting information about VI-2-7
  - interactive VI-3-5
  - real-time VI-3-5
  - suspending and resuming VI-2-7
  - terminating a VI-2-8
  - time-critical VI-3-5
- Process globals IV-1-9, VI-1-10
  - getting an entry VI-2-4
  - retrieving IV-2-4
  - setting an entry VI-2-4
  - table of values VI-1-11
- Process\_Admin VI-1-11
- Process\_Mgt VI-1-10, VI-2-3
- Process\_Mgt\_Types VI-1-10, VI-2-3
- Processing routine V-7-10
- Processing routines V-6-14
- Processor
  - claim VI-3-5
  - multiple I-1-2
  - preemption VI-3-4
- Program VI-1-2
  - definition VI-1-2
  - execution VI-1-9
  - invocation VI-1-8
  - module VII-3-2
  - structure VI-1-2
- Program object VI-1-3
- Program\_Mgt VI-1-2, VI-1-9
- Protected\_call, pragma example VII-3-14
- Protected\_return, pragma example VII-3-14
- Protection
  - Memory VII-1-3
  - system object I-1-15
- Protection set III-3-5
- Public data object VI-1-6
- Public type, example VII-3-4

## R

- Rank IV-6-3, V-6-11
- Read lock II-4-5
- Real-time job VI-3-6
- Real-time process VI-3-5
- Record
  - access modes IV-7-9
  - DDef layout IV-8-6
  - defining a DDef IV-8-10
  - ID IV-7-3
  - number IV-7-3
  - record id IV-9-16
  - record number IV-9-16
  - size IV-7-3
- Record i/o V-6-4
  - deleting records IV-9-14
  - inserting records IV-9-13
- Record ID IV-7-3
- Record number IV-7-3
- Record print layout V-8-4, V-8-8
- Record size IV-7-3
- Record AM.Ops IV-4-17
- Records IV-7-3
- Recovering from record overflow I-3-7

- Recovery IV-10-3, IV-10-11
- Recovery agent IX-1-8
- Reference Counting VII-2-10
- Region V-6-15, V-7-11
- Releasing
  - locks IV-10-7
- remove.variable V-1-10
- Report V-8-2
  - defining V-1-16
  - description V-8-6
  - details V-1-16
  - example V-1-15
  - footing V-8-3
  - footings V-8-9
  - format V-1-15
  - heading V-8-3
  - headings V-8-9
  - parts V-1-16, V-8-2, V-8-6
  - printing V-1-17
- Report service V-1-15
- Representation of objects I-3-2
- Resize rule IV-4-10
- Resource priority VI-3-8
- Resource-driven priorities VI-3-7
- Rights VII-1-12
  - representation II-2-4, VII-3-2
  - type II-2-4
- Rights evaluation III-3-6
- Root transaction II-4-4
- Running priority VI-3-8

## S

- Scheduling I-1-4
  - cpu VI-3-3
  - high-level VI-3-3
  - I/O VI-3-9
  - low-level VI-3-4
  - medium-level VI-3-5
  - memory VI-3-9
- Scheduling service object (sso) VI-3-6
- Screen fields V-6-5
  - protected V-6-9
- Secondary data area IV-7-3
- Semaphore VI-1-17
- Semaphore\_Mgt VI-2-3
- Service class VI-3-4, VI-3-6
- Service configuration attribute VII-7-5, VII-7-6
- Session VI-1-9
- set.variable V-1-10
- Shared data structures
  - locking VI-2-13
- Shared queues IX-1-8
- Sheet elements V-6-5
- Short-term logging IV-10-11
- Single activation II-3-14
- Size
  - record IV-7-3
- Spool
  - device IV-6-5
  - file IV-6-3
  - queue priority IV-6-3
  - queue IV-6-3
- Spooled mode IV-6-3, IV-6-8
- SSO priority VI-3-6, VI-3-8
- SSO\_Admin VI-3-6
- SSO\_Types VI-3-7
- Stack object VI-1-6
- Standalone directory III-1-10

Standard report V-8-8  
 Static data object VI-1-5  
 Storage  
   Claim VII-2-7  
 Stream files IV-3-2  
   copying IV-3-6  
   creating temporary files IV-3-7  
   creating IV-3-5  
   deleting IV-3-6  
   emptying IV-3-6  
 String list II-1-2  
   creating a string list II-1-6  
   illustration II-1-2  
   invalid (overflow) II-1-3  
   reading elements from II-1-6  
 String\_List\_Mgt II-1-6  
 Structured files  
   access modes IV-9-3  
 Subform V-6-9  
 Subprogram-level variable I-3-3  
 Subtransaction II-4-4  
 Subtransactions  
   locking IV-10-8  
 Swapping memory I-1-10  
 Symbolic key V-6-16  
 Symbolic link III-1-4, III-5-2  
   illustration III-5-2  
 Synchronization I-1-4  
   system I-3-2  
 System components VII-7-5  
 System Configuration Object (SCO) VII-7-4  
 System Configuration Object (SCO) VII-7-8  
 System object I-1-14  
 System SCO VII-7-8  
 System Service VII-4-4  
 System variable groups V-1-10  
   system\_Defs I-3-2  
   system\_exceptions I-3-3

T

Tab bit II-2-3  
 Tail object VII-7-6  
 TDO VII-3-3  
   as stored object VII-3-8  
 Temporary files IV-3-4  
   creating IV-3-7  
 Terminal attributes IV-4-5  
 Terminal\_Defs IV-5-2  
 terminal\_input V-6-15, V-7-10  
 Terminals  
   character IV-4-3  
   graphics IV-4-3  
 test.form V-7-6  
 Text II-1-3  
   declaring a constant II-1-4  
   illustration II-1-2  
   invalid (overflow) II-1-3  
   procedure with text result II-1-5  
   recovering from overflow I-3-7  
   using a literal II-1-4  
 Texts V-6-5  
 Time limit VI-3-5  
 Time slice VI-3-7  
 Time-critical job VI-3-6  
 Time-critical process VI-3-5  
 Timestamp conflicts II-4-6  
 Transaction II-4-1  
   and job termination II-4-6

coding rules II-4-4  
 deadlock avoidance II-4-6  
 independent transaction II-4-6  
 locking II-4-5  
 recovering from timestamp conflicts II-4-8  
 timeouts II-4-5  
   using only when needed II-4-7  
   using II-4-6  
 Transaction stack II-4-3  
 Transactions I-1-7  
 Translation tables V-7-6  
 trigger\_key V-6-15, V-7-11  
 Type I-1-13, II-2-3  
   Manager VII-1-13, VII-3-2  
   unchecked conversion I-3-3  
 Type manager II-2-3, VII-4-3  
 Type managers and data abstraction VII-3-3  
 Type rights I-1-15  
   evaluating rights to an object III-3-6  
 Typemgr\_Support VI-1-18

U

Unchecked conversion I-3-3  
 unchecked\_conversion I-3-4  
 Update VII-3-13  
 User IV-4-2  
 User agent IV-4-15  
 User variable groups V-1-10  
 User\_Mgt III-3-2  
 Using paired calls I-3-9

V

v VI-1-17  
 Variable I-3-3  
 Variables V-1-9  
 View IV-4-2, IV-5-2, VI-1-2  
 Virtual address I-3-2  
 Virtual memory VII-2-2  
 Virtual terminal IV-4-2  
 Volume set III-1-6  
   Filing VII-2-4  
   mirrored I-1-6  
   Swapping VII-2-4

W

Window IV-4-2, IV-5-2  
   input model IV-4-6  
   output model IV-4-7  
 Window attributes IV-5-6  
 Window coordinates IV-4-4  
 Window service V-1-11  
 Window\_Services IV-4-2, IV-5-2, V-5-3  
 Working set VI-3-9  
 Write lock II-4-5

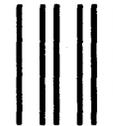




We'd like your comments . . .

Your comments help us produce better documentation. Please fill out the form on the reverse of this card, then fold it, tape it shut, and drop it in the mail. (Postage is free if mailed within the United States.) We will carefully review your comments. All comments and suggestions become the property of BiiN™.

Fold here



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES



**BUSINESS REPLY MAIL**  
FIRST CLASS      PERMIT NO. 226      HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

**BiiN™**  
Technical Publications Department  
2111 NE 25th Ave.  
Hillsboro, OR 97124-9975



Please use this form to help us evaluate this manual and improve the quality of future versions. Mailing instructions are on the other side of this form.

If you want to order publications, see Page ii of this manual.

Manual Title: \_\_\_\_\_ Revision Number (Page ii): \_\_\_\_\_

Please fill in the squares below with a rating of 1 to 10, with 1 being worst and 10 being best:

- |   |  |
|---|--|
| <input type="checkbox"/> Technical completeness | <input type="checkbox"/> Usefulness of material for your needs |
| <input type="checkbox"/> Technical accuracy     | <input type="checkbox"/> Quality and relevance of examples     |
| <input type="checkbox"/> Readability            | <input type="checkbox"/> Quality and relevance of figures      |
| <input type="checkbox"/> Organization           |  |

If you gave a 4 or lower in any category, please explain here:

---

---

---

---

Please list any suggestions you have for improving this manual:

---

---

---

---

Please list any technical errors you found:

---

---

---

---

Please tell us your name and address. (If you would like us to call you for more specific comments about this book, please list your phone number as well.)

Name: \_\_\_\_\_

Address: \_\_\_\_\_

---

---

Phone (Optional): \_\_\_\_\_

Thank you for taking the time to fill out this form.