SYSTEM SERVICES GUIDE
VOLUME 1 OF 2

BiiN™

BiiN™

SYSTEM SERVICES GUIDE VOLUME 1 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

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

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

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

PREFACE

Purpose

The *BiiN™/OS Guide* shows you how to use the services provided by the BiiN™ operating system.

Audience

This manual is intended for both applications programmers and systems programmers. Such programmers use the OS to create:

- Object-oriented applications that provide data protection, data integrity, program modularity, and extensibility
- Applications that manage record-structured files
- Interactive applications that use windows, menus, commands, messages, and forms
- Concurrent applications using multiple processes, including real-time applications
- Distributed applications that provide services at multiple nodes in a network
- New device drivers.

Organization

The *BiiN™/OS Guide* is divided into eleven major parts:

I. Introduction	Introduces the OS and how to make system calls.
II. Support Services	Fundamental services for message handling, text and string handling, using system objects, and transaction processing.
III. Directory Services	Hierarchical directories, lists of directories, user IDs, and authority lists.
IV. I/O Services	Standard I/O access methods and I/O devices.
V. Human Interface Services	Programming interactions with the user: command input, menus, forms, and reports.
VI. Program Services	Concurrent programming and scheduling.
VII. Type Manager Services	Creating new services using new object types.
VIII. Distribution Services	Creating services that exist at multiple nodes in a network and that communicate to provide distributed services.

IX. Device Services
Creating device managers and device drivers.

X. Appendixes Complete listings of examples excerpted in this manual, and a Glossary of terms used in this manual.

Index

The chapters in each part describe major programming areas such as "Using Basic I/O" or "Building Concurrent Programs." A chapter may contain basic concepts about the programming area, specific programming techniques, or both. Many techniques are illustrated with excerpts from BiiN™ Ada examples listed in Appendix X-A.

Related Publications

This manual does not provide detailed reference information for system calls. For descriptions of system calls, see the *BiiN™/OS Reference Manual.* All OS programmers should also see the "Files, Modules, and Views" appendix in the *BiiN™/OS Reference Manual* for important information regarding finding OS files and compiling and linking programs that use the OS.

The following manuals may be of use to you while programming with the OS:

BiiN™ Systems Overview
High-level description of BiiN™ systems.

Getting Started with BiiN™ Systems
How to log in, basic interactive commands, and how to set up your environment.

BiiN™ Systems Programmer's Guide
Languages and tools used to program in the BiiN™ environment, and some application examples.

BiiN™ Ada User's Guide
User's guide for the BiiN™ Ada programming language.

BiiN™ Ada Language Reference Manual
Language reference for the BiiN™ Ada programming language.

BiiN™ C Programming Manual
Programmer's manual for the BiiN™ C programming language. This manual includes a chapter on using the OS from C programs.

BiiN™/OS Reference Manual
Package descriptions for using the OS.

Notation

glossary term	Terms being defined or used for the first time are in *italic* font, and can be found in the Glossary.
`Package_Name`	BiiN™ Ada reserved words and OS package and call names are in `typewriter` font.

The following abbreviations are used throughout this manual:

K	2^{10} = 1,024. For example, 1K bytes equals 1,024 bytes.

M	2^{20} = 1,048,576. For example, 1M bytes equals 1,048,576 bytes.
G	2^{30} = 1,073,741,824. For example, 1G bytes equals 1,073,741,824 bytes.
AD	Access Descriptor, a system object pointer. An AD references a system object.
SRO	Storage Resource Object, which defines memory storage available for a job or node.
TDO	Type Definition Object, which defines an object's type. Each object references the TDO for its type.
GDP	General Data Processor, a central processing unit in a BiiN™ node.

In examples of BiiN™ commands, the user's input is $\boxed{\text{boxed}}$ and the system's prompts and responses are not. For example:

```
clex-> cg hello.c
clex-> link hello.obj :output=hello
clex-> hello
Hello, world!
clex->
```

This manual uses the following notation to describe syntax:

name ::= syntax-exp	A syntactic equation, indicating that the word on the left side symbolizes the expression on the right side.
name	Words in *italic* font are names for other expressions. A name containing hyphens, such as *basic-type-specifier*, should be considered a single word.
`name`	Words and symbols in `typewriter` font are literal characters and character strings.
a-z	Specifies any single character >= *a* and <= *z* in the ASCII collating sequence.
A / B	OR: Specifies a string that matches *A* or a string that matches *B*.
[A]	Brackets surround an optional syntactic element.
A...	Ellipses indicate that one or more elements can be used.
(A)	Parentheses group items to specify an order of evaluation.

An example of syntax notation:

```
appetizer ::=
  [ soup-type ] soup /
  vegetable almondine /
  chips [ & salsa ]

soup-type ::=
  meat noodle /
  cream of vegetable

meat ::=
  chicken / beef

vegetable ::=
  potato / cauliflower / broccoli
```

The following strings are valid appetizers, according to the above syntax:

```
chicken noodle soup
cream of broccoli soup
broccoli almondine
chips
```

CONTENTS

Part I. Introduction.

Part II. Support Services.

Chapter 1. Using Utility Packages

Chapter 2. Using Objects and ADs

Chapter 3. Storing Objects

Chapter 5. Writing Messages

Part III. Directory Services.

Chapter 1. Understanding Directories

Chapter 2. Using Directories

Chapter 3. Protecting Stored Objects

Part IV. I/O Services.

Chapter 1. Understanding I/O Access Methods

Chapter 4. Using Windows

Chapter 5. Using Character Display I/O

Part V. Human Interface Services.

Chapter 1. Understanding Human Interface Services

Chapter 2. Creating a BiiN™ Application Program

Chapter 3. Building New Commands

Chapter 6. Understanding Forms

Chapter 7. Programming with Forms

Chapter 8. Generating Reports

Part VI. Program Services.

Chapter 1. Understanding Program Execution

Chapter 2. Building Concurrent Programs

Chapter 3. Scheduling

Part VII. Type Manager Services.

Chapter 1. Understanding Objects

Chapter 2. Understanding Memory Management

Chapter 6. Building Type Managers for Stored Objects

Chapter 7. Understanding System Configuration

Part VIII. Distribution Services.

Chapter 1. Understanding Distribution

Chapter 2. Building a Distributed Type Manager

Part IX. Device Services.

Chapter 1. Understanding Device Managers and Device Drivers

Part X. Appendixes.

Appendix A. Ada Examples

Appendix B. Glossary

List of Figures

List of Tables

Part I
Introduction

This part of the *BiiN™/OS Guide* provides important concepts and basic programming techniques that are used throughout the system and throughout this manual. You should read these chapters before reading any subsequent chapters in this guide.

The chapters in this part are:

Concepts	Provides an overview of the OS.
Services Areas and Services	Describes the organization of OS packages into service areas and services.
Ada Programming Techniques	Contains common Ada programming techniques used with system calls. (A future release will add a chapter to describe C programming techniques used with system calls.)

Part I Overview

CONCEPTS 1

Contents

This chapter provides an overview of the BiiN™ operating system (OS) for BiiN™ computers. It discusses:

- The *functionality* of the BiiN™ OS relative to other well-known operating systems such as VAX/VMS and UNIX systems
- The object-oriented *architecture* of the BiiN™ OS.

The BiiN™ OS is accessed using System Services. These services provide a variety of operations.

I-1.1 BiiN™ OS Functionality

The OS is made up of logical groups of BiiN™ Ada packages. Each package contains system calls to the BiiN™ OS. These services support and protect applications. These applications can be:

multiprocessing	Providing a common queue of processes for execution by one of many CPUs
fault tolerant	Giving nearly continuous service that protects against accidental or malicious destruction of information
transaction processing	Ensuring the integrity of system and application disk storage
distributed	Supporting location-independent processing, local area networks, circuit switched networks, and public packet switched networks.

In addition, there are several other important features and functions discussed in this chapter.

I-1.2 Transparent Multiprocessing with Multiple Processors

A single BiiN™ node can have multiple processors that share a common addressable memory. Also, multiple computing nodes can be connected into a single distributed system that shares data and resources between nodes. See Figure I-1-1.

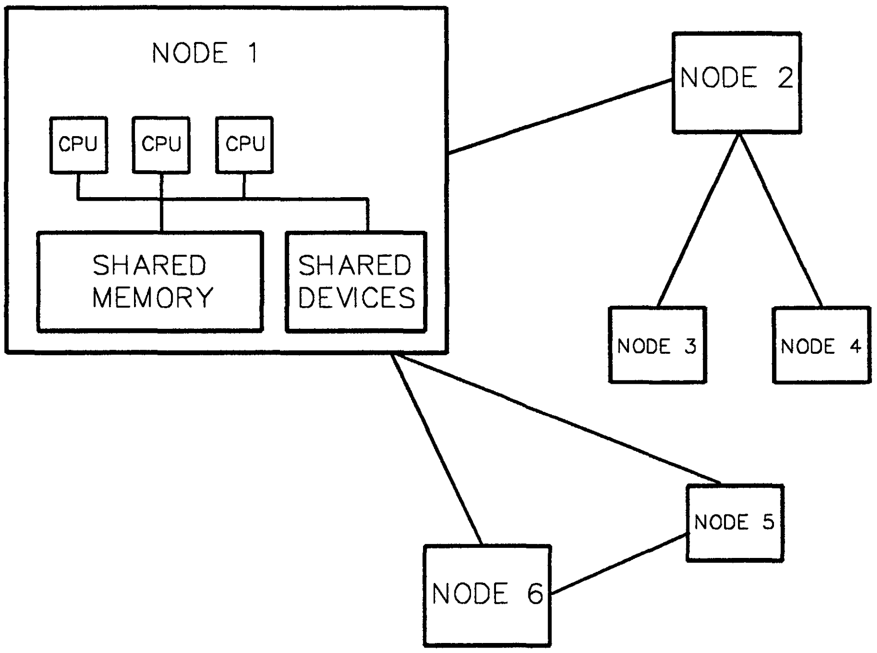


Figure I-1-1. Networked, Distributed, Multiprocessing Nodes

With the strategy employed by other systems, it's difficult for different processes to share memory (in particular, program variables).

- There's no CPU support to efficiently synchronize access to shared data from multiple processes.
- There aren't primitives to help a scheduler make the right scheduling decisions. The scheduler doesn't know when processes are working on the same task and should be scheduled together.

The BiiN™ OS supports the CPU with low-level primitives that handle multiprocessing. Unlike most computer systems on the market today, BiiN™ systems have been designed—from the VLSI-component level to the OS level—to support multiple processors.

Figure I-1-2 shows how the dispatcher handles multiprocessing.

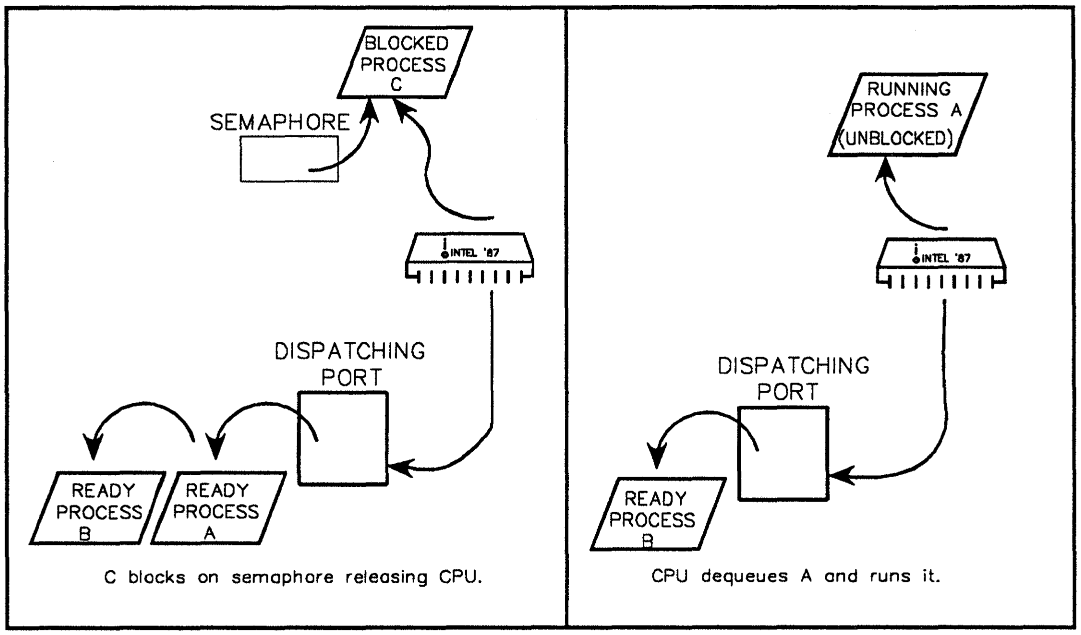


Figure I-1-2. How the Dispatcher Handles Multiprocessing

- A single processor is self-dispatching at a dispatching port.
- Synchronization and communications use high-level instructions:
  - semaphore instructions,
  - communication port instructions, and
  - a dispatching port instruction.
- The multiprocessing is transparent to users.
- *Low-level scheduling (dispatching of processes) is performed by the CPU, with no direct OS intervention.* All CPUs share a common queue of processes, and the work load is evenly shared among all CPUs.
- *The CPU provides synchronization instructions.* A synchronization CPU instruction can suspend a process while allowing other processes to run. This is done without OS intervention. Synchronization instructions such as semaphore locking and unlocking that suspend and release a process are much less cycle-intensive than test-and-set instructions that keep chewing up cycles.
- *Computations are done as jobs.* Initially, a job has one process. Your program can create more processes in the same job. All processes in the same job can share the same address space (for example, the same global program variables). The OS scheduler schedules jobs rather than processes; it schedules jobs into and out of the dispatching mix based on external priorities and resource constraints. It is quite possible for all the processes of a single job to be simultaneously executing—each on a different processor.

I-1.3 Fault-Tolerant Computing

In existing computer systems, protection mechanisms are very limited and have changed little in the last 20 years. If any application or service makes an addressing error, it can overwrite or otherwise corrupt data (or code) in many other parts of the system. Finding an error is difficult because almost any application or service could have caused the error, not just the module that detects the error. Because errors are not confined to one module or data structure, system reliability is limited, and the system becomes less reliable as its software becomes more complex.

The BiiN™ OS detects errors at their source (or, at minimum, nearby) and limits the damage that any one program can cause. The hardware and the OS software work together to make addressing violations impossible; no service can access code or data outside its protected address space.

The non-stop, fault-tolerant engineering of the BiiN™ OS relies on the concept of a *confinement area* within which an error is contained at the time of detection and repair. If a bug is detected, then the damage is known to be confined to the address space accessible to that program.

The BiiN™ OS *supports* hardware fault tolerance. The *BiiN™ Series 20/40 Hardware System Description* and the *BiiN™ Series 60/80 Hardware System Description* describe hardware fault tolerance.

OS support for hardware fault tolerance includes:

- You (or your system administrator) can choose a level of hardware fault tolerance for your particular system configuration.
- You can monitor hardware operations for potential failures.
- You can configure redundant hardware to step in, for example, if a board fails. The hardware-controlled "stepping in" occurs without interrupting your normal servicing.

Your system administrator determines policy. For example, if a board goes out and the system recovers, decisions are required:

- Should the system maintain the same level of fault tolerance and run with fewer processors?
- Or, should the level of fault tolerance be set lower so that checking occurs without recovery and all processors continue functioning?

The BiiN™ OS lets you control the outcome of these decisions. It also supports fault tolerance by providing built-in redundancy. Figure I-1-3 shows how the BiiN™ OS aids fault tolerance.

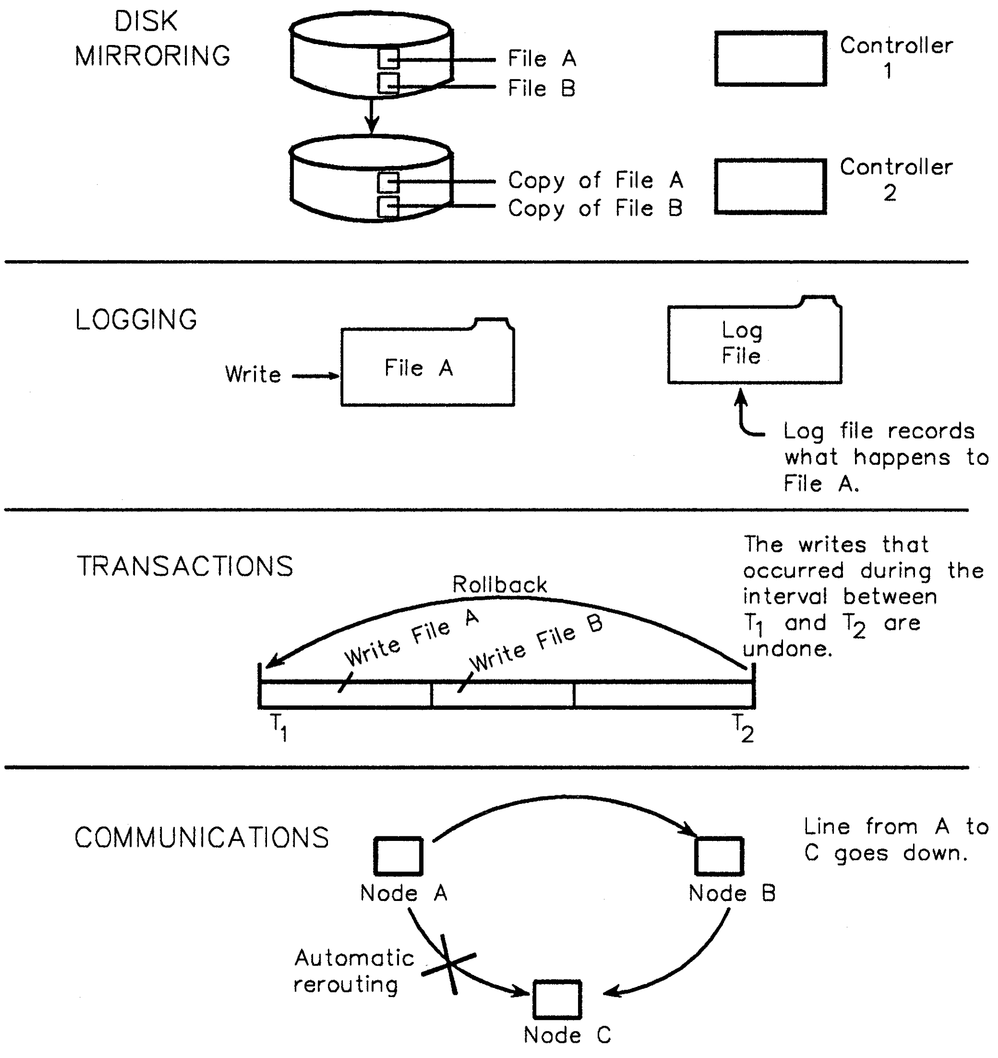


Figure I-1-3. How the BiiN™ OS Aids Fault Tolerance

- *Volume sets can be mirrored.* If a file exists on a mirrored volume set, the file exists on two disks.
  - If one disk or I/O controller goes down, the data on a mirrored volume set remains accessible.
  - Mirroring can be re-established (online and transparent to applications using the disk) if the bad disk comes back up.
- *Files and directories can be logged.* Everything that happens to the file or directory can be written to a log. After a disk crash, the file or directory can be restored from a previously saved back-up copy. Once restored, the file can be rolled forward to a specific date/time based on log entries.
- *Incomplete transactions are undone.* If a system crash occurs before a transaction is completed, all effects of the transaction are automatically undone. (See the next section for more on transactions.)

- *Communication is automatically rerouted.* The nodes in a multi-node system can be connected with redundant connections controlled by separate I/O controllers. If a connection is lost, communication gets rerouted.

I-1.4 Transaction Processing and DBMS Support

Transactions are a familiar concept to most mainframe DBMS users. Basically, transactions group file writes so that either all occur at once, or none occur at all.

Although transactions are primarily used to protect data in files, the BiiN™ OS extends the concept of transactions to include directories and other resources managed by non-filing services.

Most conventional systems build transactions into a database layer:

- To use transactions, programmers are forced to learn a DBMS. Existing files and programs must be converted to DBMS formats. This is acceptable for programmers who are familiar with query languages such as SQL. It's not, however, always the best solution for programmers who want quick record access using the existing files of their ported applications.
- Because conventional OS filing does not provide the right structures for database systems, transactions build into a database layer can be hard to implement. DBMS software must build file structures (for example, a file cache on top of virtual memory) using the primitives supplied by the OS. This is inefficient.

Transactions and other DBMS filing functions are built into the BiiN™ OS. The BiiN™ filing service offers:

- UNIX-style byte stream files *and* special BiiN™ record files
- hashed or b-tree indexes for record files
- sequential, relative, clustered, hashed, and unordered file organizations
- one or more key values (of multiple data types) for an index key
- support for null values
- true variable-length records and true variable-length fields within a record
- integration with the BiiN™ Data Definition Facility (known in other systems as a data dictionary facility).
- record-level locking integrated with transaction-support
- different levels of consistency including level 3 as defined by IBM SYSTEM R.
- sorting and merging large collections of records
- database joins, projects, and selects
- logging, integrated with backup/restore, so that a file can be backed up and later rolled forward from a log.

A major performance advantage of the BiiN™ file service is file buffering that uses a file cache in a special RAM-based stable store. Several configurations are available. For example:

- A configuration that is fully duplicated, ECC protected (with spare-bit), battery-backed-up, with each component powered by separate power supplies and separate batteries

- A configuration that is accessible from two busses (in case one bus goes down).

Because of the reliability of the stable store, writes to disks may be delayed indefinitely. Completing a transaction or closing a file may not cause a disk write.

I-1.5 Computing in a Distributed Environment

When large timesharing machines in the 1970s were shared by many users, data and file sharing was easy. However, CPU cycles were hard to come by.

Today anyone can have a PC, workstation, or other node in a small local area network attached to a mainframe or mini. CPU cycles may be plentiful, but program and data sharing between nodes is complicated, often requiring communications, file transfers, and remote file access. This is tricky to do without a knowledge of file naming conventions and network protocols.

The BiiN™ OS protects users from the complexity of inter-node communication between services. For example, you can type a command at your home node and simultaneously run programs at other machines that are accessing files from still other machines. The combined file space of all nodes looks like a single file space.

- When a program runs, it sees the *same* current directory and home directory regardless of the node it runs on.
- There is no special naming for remote files. A file stored in your directory with the name `suppliers` might be on any node on a distributed system. The program that accesses the file (regardless of the node the program executes on) sees the same interface to the OS file service.
- You control where your program is run, but your system administrator controls which nodes you can run on, and the quality-of-service you'll get on each of these nodes.
- You can control the location of your files (and other programming resources) and find out where files needed by your program are located. For a program with lots of I/O, it is often more efficient to run the program on the same node as the data, rather than bring the data to the node running the program.

I-1.6 Support for UNIX and ISO Standards

The BiiN™ OS supports many industry standards, including:

- System V Interface Definition for UNIX systems
- Communication Protocols:
  - ISO Transport Class 4
  - ISO FTAM
  - X.25
  - HDLC
  - LAN 802.3
- IEEE Floating Point

These standards allow you to easily integrate your existing hardware and software into BiiN™ systems. Since UNIX System V-compatible calls are supported, you can port your existing UNIX applications easily.

I-1.7 Services for High-Function Applications

Many systems provide *two* address spaces within a process—one for an application and one for the OS. Filing, program execution, and other supervisory routines reside in a monolithic kernel. This is known as the *two-space* view.

As applications increase in function, they are becoming more complex. To build these high-function applications, supporting services such as database, forms, and communications are required. The dilemma faced by traditional two-space operating systems is how to fit these supporting services to the OS.

There are three approaches to adding a service to a two-space operating system:

- *Put the supporting service in the address space of the OS.* The result? The OS increases in size and complexity. The introduction of a new service (from which the rest of the OS can't be protected) results in lower OS reliability, and therefore reduced system reliability. Typically, this is how communications is implemented.
- *Put the supporting service in the address space of the application.* This is often impossible because the supporting service needs to access data and operations that the application isn't allowed to access. It becomes difficult to track down errors: an apparent malfunction in a supporting service could be caused by a bug in the supporting service or the application program. The bug might be as simple as using an uninitialized index variable to store into an array.
- *Put the supporting service in its own address space.* This is the approach often taken by mainframe DBMS software—it means putting the DBMS in its own process. The result?
  - Invoking the supporting service from application programs can be awkward. Instead of a simple call/return mechanism, costly inter-process communication must be used.
  - Processing bottlenecks occur when different applications make requests at the same time.
  - There are problems in accounting, resource control, and protection. How does the requesting application get charged for its use of the service? How does the service know the identity of the requestor? How does the service prevent one application from swamping it with requests, at the expense of other applications?

In contrast, the BiiN™ solution gives each supporting service its own address space within a process. This is known as the *n-space* view. Calls to a service are synchronously executed by the user process itself. However, the data and operations of the called service are protected from the caller by using a separate address space for the service.

Today's high-function applications often need to use many supporting services. This demands more than a simple *2-space* view of the world. The BiiN™ OS provides a uniform call/return mechanism that can be used by *all* services in the system—from supervisory routines to applications. Each service can have its own call stack, and can be used by the application the same way you use an existing service (such as an OS filing service). This increases system speed, reliability, and ease-of-use. Basically, the key to understanding the *n-space* view is protection.

A traditional "onion skin" view of the BiiN™ OS doesn't convey how programs and data are protected by the system. Each system service can have an invincible boundary of protection built around the address space it occupies.

Compared to other operating systems:

- *The BiiN™ OS is fast.* Within a single process, each service can execute in its own address space. A single call instruction takes care of switching address spaces. This form of call is faster than most supervisor call instructions on other machines.
- *The BiiN™ OS is reliable.* An invoked service's access to the caller's address space is limited to just the parameters passed by the caller. A service is protected from an application, and an application is protected from a service. One service can invoke another service using the same calling conventions.
- *The BiiN™ OS is easy-to-use.* A service executes in a user-invoked process. It does not have to provide its own protection, resource control, and accounting mechanisms. With less code, and fewer primitives to learn, you can concentrate on the service's operations.

I-1.8 Transparent Resource Management for Easy Programming

Many operations of the BiiN™ OS are executed transparently using virtual memory and file buffering (with a little help from hardware):

- You can invoke and run several programs simultaneously. Each program runs as a job and can appear as a window on your terminal screen.
- Job scheduling, memory space allocation, and file buffer space are handled automatically.

For example, the total of all address spaces for a job might be 2 MB, but your job really only needs 500 KB of primary memory to run. Here's what happens:

- When a job accesses a page not present in primary memory, a fault (invisible to the job) is generated and the page gets swapped in.
- When the hardware reports that a page in primary memory has not been recently accessed, the page is swapped out if changed previously.

I-1.9 Getting Real Time Data

Suppose you need to monitor the movement of robot vehicles on the floor of a factory. You need a way to sense their movement, perform computations, and tell them what to do next—in real time. You don't want a lot of memory and I/O overhead to do this; the robots would be crashing into each other because of the time delays.

The BiiN™ OS stays out of the way and lets the hardware do much of the work:

- As your program executes, application-defined interrupt handlers are invoked without OS intervention.
- Low-level scheduling and synchronization is handled by the CPU.
- Low-level scheduling is priority-based with preemption.

Consider an interrupt procedure that gets invoked due to a signal from a robot vehicle. The procedure might do some processing and then signal a semaphore to cause a suspended real-time process to run. The OS just stays out of the way and lets the hardware do the work.

The BiiN™ OS supports real-time programming:

- Your system administrator can define different real-time scheduling levels and grant access to specific users/programs for a particular level.
  - A real-time program can run at a high-priority level.
  - The job remains in the dispatching mix (managed by hardware), and bypasses the OS scheduler.
- A real-time program can spawn multiple processes within the same job. Each process can run at a different priority.
  - Processes can communicate information using shared memory and can synchronize using semaphores.
  - The hardware also provides a message-passing mechanism (ports) that uses `Send` and `Receive` instructions. Like semaphores, the port mechanism is integrated with hardware dispatching.
- A real-time job can run with its entire address space in primary memory (that is, in *frozen*, non-relocatable memory). When this occurs, the job will not encounter any virtual memory faults.
- Real-time data collection programs can quickly stream large amounts of data to and from disk, with minimal disk head movement.
  - The filing service lays out files contiguously on disk using extents.
  - The file buffer management strategy is read-ahead and write-behind.
  - Indexes and file records can be placed on different physical disks.

I-1.10 System Administration and the Clearinghouse

What is a "Clearinghouse?" Basically, it's a location-server database that lets a system administrator easily control and administer a network of nodes. The Clearinghouse maintains a record of which objects and IDs are at which nodes.

With most distributed computer systems, it's difficult to administer networks that consist of more than a handful of nodes:

- To add a new node to a network, the system administrator has to modify configuration data for *all* nodes.
- Additional modifications are required if a node is moved from one local network to another.
- Often, a file name previously used to access a file on a node must be changed when the node is relocated.
- If a node goes down with a bad board, there is no way to easily move the node's disk to another node.

To remedy this situation, the BiiN™ OS maintains a database of information about nodes, users, volume sets, and distributed services—the Clearinghouse.

- The system administrator adds information using a Clearinghouse utility.
- The information is duplicated on a few key nodes, and is available to all nodes.
- When node 1 needs information about node 2, it asks the Clearinghouse.
- The system administrator can select which nodes have Clearinghouse data, and which portion of the total Clearinghouse database they contain.

Suppose, for example, that node 1 contains a volume set (a logical disk). Your system administrator can move the volume set from node 1 to node 2 (perhaps on a different local network) by changing the I/O configurations of nodes 1 and 2. This does not affect users and programs that previously accessed files on the volume set. *All file names remain the same, and appear as if they are on your home node.*

I-1.11 BiiN™ OS Architecture

System Services consist of several distinct *service areas* such as I/O Services. These service areas consist of one or more system *services* such as the filing service. Each service controls a certain part of the system, and all services interact.

Figure I-1-4 illustrates the relationship between services, packages, and calls.

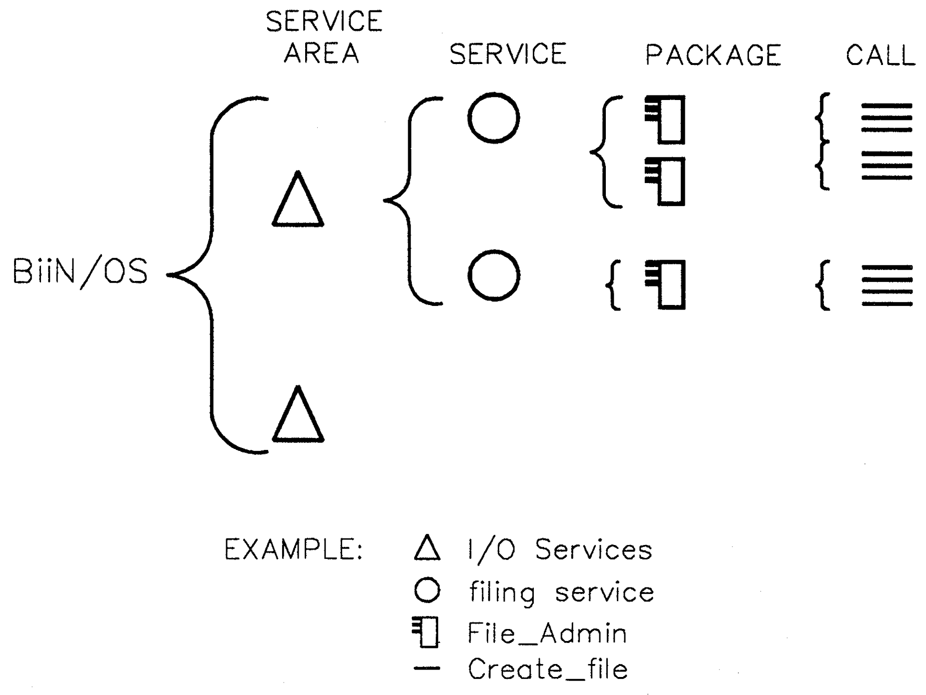


Figure I-1-4. The OS Interface is Made up of Services, Packages, and Calls

I-1.12 Some Basics

Each service executes in its own address space. This space is broken up into individual protected segments of memory called *objects*. One way to think of this decomposition is to imagine a box of building blocks. The blocks—triangles, squares, rectangles, and so forth—can be combined to form different structures such as bridges and houses.

One structure (for example, a bridge) is independent from another structure (for example, a house), yet all structures are composed of the same basic blocks. In this analogy the differently shaped blocks are *objects*, and the structures correspond to *services*. All blocks of the same shape have the same characteristics. Similarly, every object has a "type", and all objects of the same type have the same characteristics.

An object is sometimes referred to as a *system object* to distinguish it from a *BiiN™ Ada object.* (A BiiN™ Ada object is a variable or construct—see the *BiiN™ Ada Language Reference Manual.*) However, when you see the term *object* used in this manual, it refers to *system object* unless otherwise specified.

Think of an object as a resource managed by a service. For example, a file is represented as an object of type "file" that is managed by the filing service. Each individual service controls access to its objects.

Existing operating systems provide two mechanisms to name files, I/O channels, users, processes, and nodes (that is, their "objects"):

names	Symbolic names that you assign to objects, and
identifiers	Binary digits that provide an efficient means for a program to identify an object.

Each "object" in existing systems usually can be referenced by an identifier. In general, the format for identifiers of each object type is different—an I/O channel identifier has a different format than a user identifier. The mapping of names to identifiers is also different for each object type.

In contrast, the BiiN™ OS supports one form of identifier (actually in hardware) for its objects. This allows identifiers for different object types to be distinguished. It also permits universal name mapping.

BiiN™ OS identifiers serve several functions:

- They contain the addresses of the objects they correspond to (that is, they function as conventional *pointers*).
- They specify the rights of the calling program to use the objects managed by a service.
- They can be used to find out the type of object they reference.

This manual uses the term *access descriptor* or *AD* to refer to the object used by the BiiN™ OS. Figure I-1-5 shows how ADs specify the calls you are allowed to make.

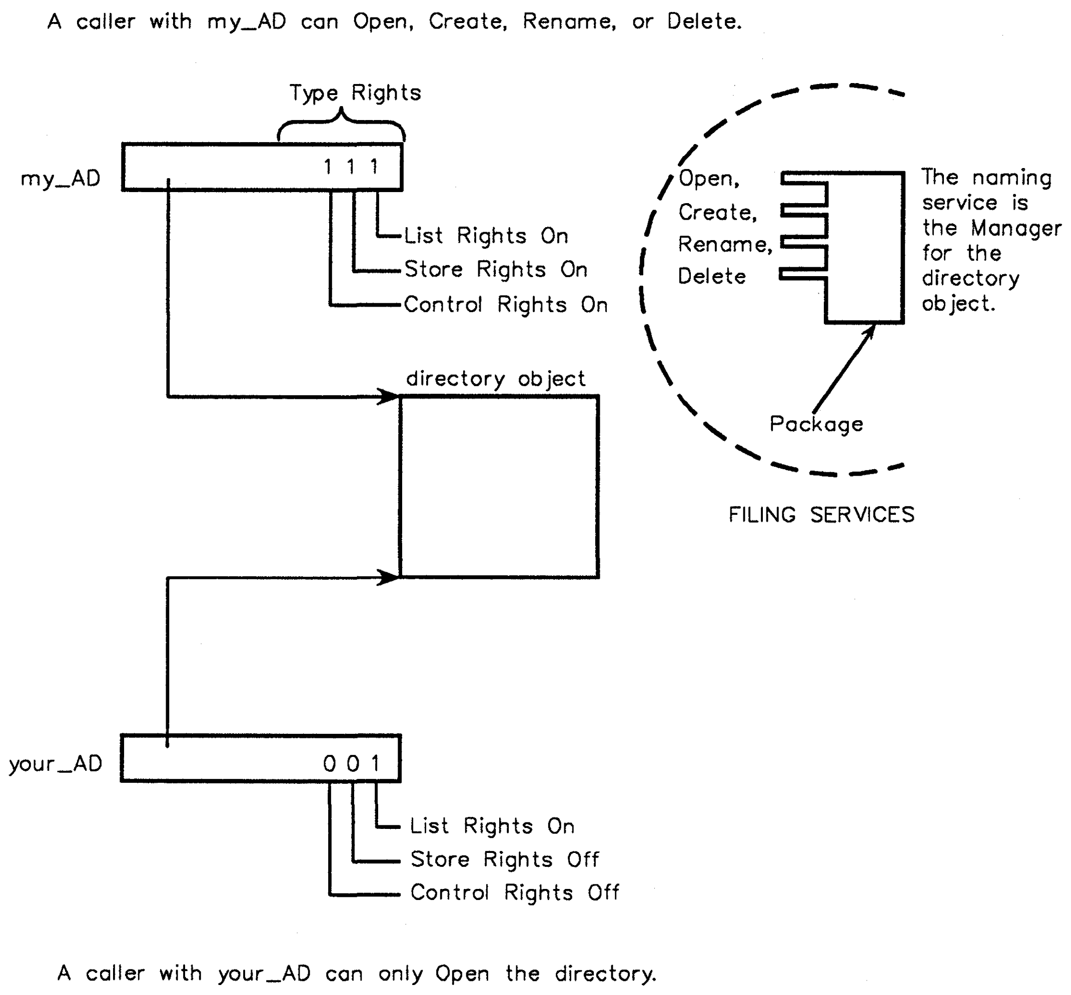


Figure I-1-5. ADs Provide Access and Protection to Services

The caller with `my_AD` can make the calls `Open`, `Create`, `Rename`, and `Delete`. The directory service processes those calls. The identifier `my_AD` controls the caller's *access*. Similarly, the caller with `your_AD` can only `Open` the directory (for reading).

I-1.12.0.1 What Is A System Object?

A *system object* is a protected segment of memory.

A *system object* is distinct from a *BiiN™ Ada object* (an entity that contains a value of a certain type). Whenever you see the unqualified use of the term *object* in this manual, it means *system object.*

There are many types of objects, including file, directory, and pipe objects. Each type of object can have multiple *instances* (for example, there might be several instances of an object of type "pipe" in memory at any particular moment).

I-1.12.0.2 How Are System Objects Protected?

System objects are protected from unauthorized reference. Access to an object is restricted to software with a "need to know" about the object. Access can be controlled at the level of individual data structures and procedures through the use of an AD. *Each call requires the use of one or more ADs.*

Each memory word has a tag to indicate whether or not it is an AD. ADs can only be manipulated in controlled ways and with special instructions, all designed to make accidental or malicious violations of the object protection mechanism impossible.

ADs are also protected pointers to data structures and correspond to the pointer values supported by some programming languages. For example, a BiiN™ Ada `access` value can be represented as an AD. ADs are also synonymous with the protected pointers called *capabilities* provided by some object-oriented computer architectures.

There can be multiple ADs for an object, and different ADs can grant different *access rights.* There are two classes of access rights:

type rights	There are three type rights. Each right corresponds to a set of operations that manipulate an object. The type rights used by the operating system are usually mapped to *use*, *modify*, or *control*. A caller can have any combination of the three type rights.
rep rights	There are two representation ("rep") rights. They are used to control access to the contents of an object (using CPU instructions directly). These are only important to you if you're creating your own service. See "Type Manager Services."

An understanding of type rights is helpful for most OS programming. Figure I-1-6 shows an AD with type rights.

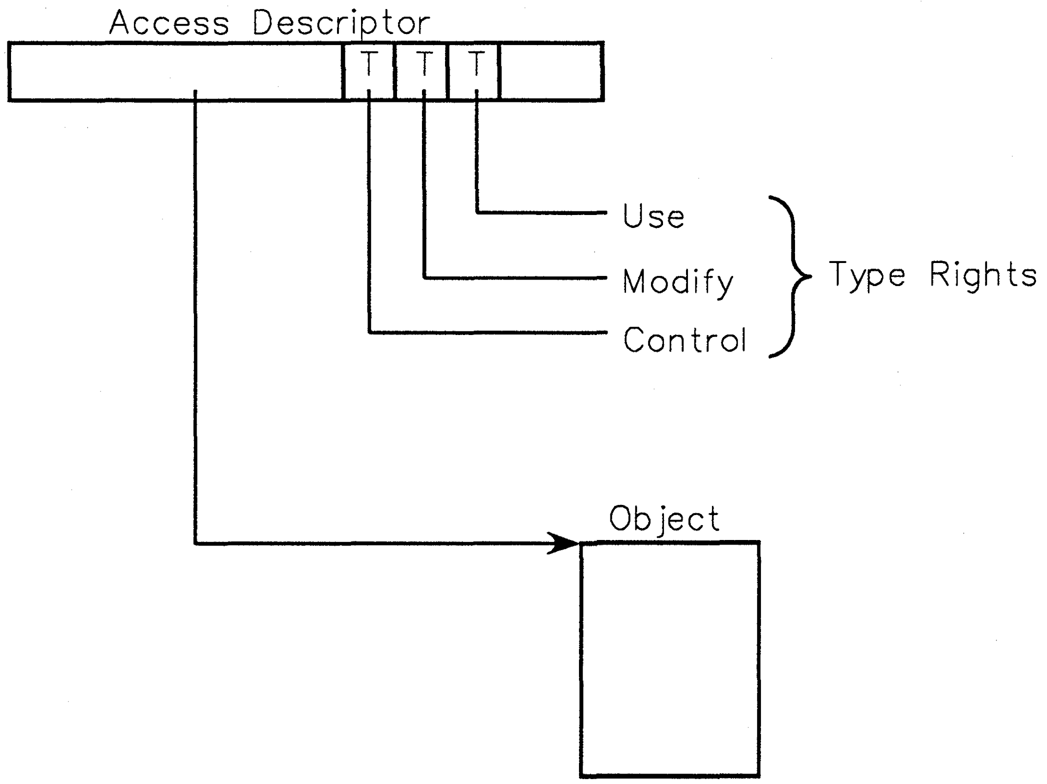


Figure I-1-6. An AD Showing Type Rights

Type rights are:

use	Required to retrieve information from an object, without changing it. Correspond to a set of operations provided by the service that manages the object.
modify	Required to modify an object, without destroying it or changing its basic nature. Correspond to a set of operations provided by the service that manages the object. When compared with *use* rights, *modify* rights give a user additional operations to manipulate the object.
control	Required to destroy or restructure an object.

Different services sometimes map their rights to variant names. For example, *use* rights—needed to read a file—in the filing service corresponds to the same level of access as *list* rights—needed to list the contents of a directory—in the directory service.

Two programs can have ADs for a shared object, with one having only *use* rights and the other having only *modify* rights. An AD can also be *null*, indicating that it references no object. For example, objects can be linked together in a list, with each object containing an AD that references the next list element. The last object in the list would contain a null AD in the link field, indicating that there are no more list elements.

ADs can be freely copied. It is normal to pass a copy of an AD to a called subprogram to specify an object as a parameter. The rights on an AD are often *restricted* when it is copied: some rights are removed from the copy, leaving only those rights needed by the subprogram that receives the copy.

Adding rights to an AD is called *amplifying* those rights. Only the service that manages the object is allowed to amplify its rights. See "Type Manager Services" for details.

SERVICE AREAS AND SERVICES 2

Contents

This chapter briefly describes the OS as a collection of *services* and *service areas*. A *service* is simply a logical collection of packages. A *service area* is a logical collection of services. Services and service areas define a logical organization of the OS, for documentation and learning purposes.

Often, a service manages one or more closely related object types. For example, the naming service manages directories, open directories, name spaces, open name spaces, and symbolic links.

Packages listed in this chapter can be found in the *BiiN™/OS Reference Manual.*

I-2.1 Service Areas

There are eight service areas:

Support Services	Often-used basic services, including system definitions, utility packages, object management, transactions, and messages.
Directory Services	Manages directories, directory lists (name spaces), symbolic links, and the authority lists and IDs used to protect directory entries.
I/O Services	Provides byte-stream, record, and character display I/O. Manages files, character terminals, character terminal windows, printers, spool queues, and other I/O devices.
Human Interface Services	Provides commands, forms, and reports used to interact with users.
Program Services	Provides various program execution services including concurrent programming, scheduling, timing, resource control and accounting, and program monitoring.
Type Manager Services	Provides special OS interfaces for trusted type managers, including access to global memory and participation in system configuration.
Distribution Services	Provides services used to build distributed applications that execute transparently in a distributed BiiN™ system.
Device Services	Provides services used to build new device drivers and device managers.

I-2.2 Support Services

Support Services contains:

utility service
object service
transaction service
message service.

I-2.2.1 Utility Service

Manages system definitions, text strings, and long integers.

`Long_Integer_Defs`
Defines types and calls for 64-bit long integers.

`Machine_Code_Insertion`	Provides useful operations that map to inline CPU instructions.
`String_List_Mgt`	Provides operations on string lists.
`System`	Provides implementation-defined (as opposed to Ada-defined) types and constants.
`System_Defs`	Provides common definitions used throughout the OS.
`System_Exceptions`	Defines common exceptions.
`Text_Mgt`	Provides operations on text records.

I-2.2.2 Object Service

Manages objects, access to objects, and storage of objects.

`Access_Mgt`	Interface for checking or changing rights in access descriptors.
`Attribute_Mgt`	Provides a way to define general-purpose operations supported by multiple object types or objects, with different type-specific or object-specific implementations.
`Object_Mgt`	Provides basic calls for object allocation, typing, and storage management. Defines access rights in ADs.
`Passive_Store_Mgt`	Provides a distributed object filing system.

I-2.2.3 Transaction Service

Manages transactions.

`Transaction_Mgt`	Provides *transactions* used to group a series of related changes to objects so that either all the changes succeed or all are rolled back.

I-2.2.4 Message Service

Manages system and application errors and messages.

`History_Services`	Contains calls for using a job's *history log files.* See also the built-in " `.history_log`" commands, and the `::history` control option, in the *Command Language Executive Guide.*
`Incident_Defs`	Defines incident and message types.
`Message_Adm`	Manages message files used by Message Services.
`Message_Services`	Provides calls to write messages from message files, message stacks, or message blocks.
`Message_Stack_Mgt`	Manages a process's message stack.
`Msg_Object_Defs`	Defines the four message objects used by the operating system.

`System_Error_Recording`	Provides calls to record errors in a *system log file.*

I-2.3 Directory Services

Directory Services contains:

naming service
protection service.

I-2.3.1 Naming Service

Manages directories, lists of directories, and symbolic links.

`Directory_Mgt`	Manages directories and directory entries.
`Name_Space_Mgt`	Provides calls to manage name spaces (lists of directories).
`Symbolic_Link_Mgt`	Provides calls to create, list, and identify symbolic links.

I-2.3.2 Protection Service

Manages authority lists, IDs, and user profiles.

`Authority_List_Mgt`	Provides calls to manage authority lists and to evaluate a caller's access rights to objects protected by authority lists.
`Identification_Admin`	Provides calls to create and modify IDs, and to modify an ID's user profile.
`Identification_Mgt`	Provides operations to manage IDs and ID lists.
`User_Mgt`	Provides calls to manage a user's protection set and user profile.

I-2.4 I/O Services

I/O Services contains:

basic I/O service
character terminal service
print service
spool service
filing service
database support service
data definition service
volume set service
basic disk service
basic streamer service
null device service.

I-2.4.1 Basic I/O Service

Manages byte stream I/O, common I/O definitions, and byte stream files.

`Byte_Stream_AM`	Provides device-independent I/O using streams of bytes.
`Device_Defs`	Declares common I/O types, constants, and exceptions.
`Simple_File_Admin`	Manages stream files.

I-2.4.2 Character Terminal Service

Manages character terminals and character terminal windows.

`Character_Display_AM`	Provides device-independent I/O to character display devices such as printers, plotters, and windows on character and graphics terminals.
`Character_Terminal_Mgt`	Manages character terminals.
`Terminal_Admin`	Provides administrative operations for terminals.
`Terminal_Defs`	Defines constants, types, and exceptions used by the terminal service packages.
`Terminal_Info`	Manages *terminfo* entries.
`Window_Services`	Provides windows on character and graphics terminals, including pull-down menus.

I-2.4.3 Print Service

Manages printers.

`Printer_Admin`	Provides administrative operations for printers.

I-2.4.4 Spool Service

Manages spool queues.

`Spool_Defs`	Declares types and constants used by spooling packages.
`Spool_Device_Mgt`	Manages spool devices.
`Spool_Queue_Admin`	Provides administrative calls for spool queues.

I-2.4.5 Filing Service

Manages files and records.

`File_Admin`	Administers files.

`File_Defs`	Provides declarations used for filing and indexing.
`Record_AM`	Provides device-independent record I/O.

I-2.4.6 Database Support Service

Provides advanced or trusted interfaces to support DBMSs (database management systems).

`Join_Interface`	Provides support for block joins of records from multiple indexed files or record stream devices.
`Record_Processing_Support`	Provides specialized support for processing collections of records.
`Sort_Merge_Interface`	Sorts and merges records from one or more input devices into a single ordered record stream.
`Trusted_Record_Processing_Support`	Provides specialized support for processing collections of records using user-supplied routines.

I-2.4.7 Data Definition Service

Manages data definitions.

`Data_Definition_Mgt`	Manages data definitions (DDefs). This interface is a symbol table for the development of a DDef compiler.
`DDF_Utility_Support`	Defines DDef properties used by services other than the data definition service.
`Field_Access`	Provides buffer access to fields in records that reference data definitions (DDefs).

I-2.4.8 Volume Set Service

Manages volume sets.

`Volume_Set_Admin`	Manages volume sets.
`Volume_Set_Defs`	Defines types, constants, and type-checking for volume sets and volume set disks.
`VSM_Disk_Admin`	Provides administrative and information calls for volume set disks.
`VSM_Disk_Support`	Provides calls to initialize a volume set disk, verify the allocated space on a disk, and remove a volume from a volume set disk.

I-2.4.9 Basic Disk Service

Manages basic disks.

`Basic_Disk_Mgt`	Manages basic disks.

I-2.4.10 Basic Streamer Service

Manages basic streamers, representing streaming tape drives.

`Basic_Streamer_Mgt`	Manages basic streamer devices.

I-2.4.11 Null Device Service

Manages null devices, used as "bit buckets" that discard all output and provide an immediate end-of-file for input.

`Nuldev_Mgt`	Manages null devices. Null devices support byte stream I/O and record I/O.

I-2.5 Human Interface Services

Human Interface Services contains:

command service
form service
report service.

I-2.5.1 Command Service

Supports application-defined commands and command sets, and manipulation of command language variables and of command help texts.

`CL_Defs`	Contains declarations used by the command service, for processing command language (`CL`) arguments and variables.
`Command_Execution`	Contains a procedural interface to command execution.
`Command_Handler`	Contains operations for reading and processing program commands and arguments.
`Environment_Mgt`	Contains operations to get, set, or remove local and global environment variables.
`Help_Text_Adm`	Manages command and form help texts.

I-2.5.2 Form Service

Manages forms.

`Form_Defs`	Defines types and constants used by the `Form_Handler` package.
`Form_Handler`	Provides calls to process, control, and change forms.

I-2.5.3 Report Service

Manages reports.

`Report_Handler`	Provides calls for initializing and printing a report.

I-2.6 Program Services

Program Services contains:

concurrent programming service
scheduling service
timing service
resource service
program building service
monitor service.

I-2.6.1 Concurrent Programming Service

Supports concurrent programs, programs with multiple processes or jobs executing together.

`Event_Admin`	Provides `Establish_event_handler` and `Change_event_state` calls for administrative users, more powerful than the corresponding calls in `Event_Mgt`.
`Event_Mgt`	Manages event clusters. Event clusters provide distributed communications and software interrupts for processes.
`Job_Admin`	Provide a more powerful `Invoke_job` call for administrative users.
`Job_Mgt`	Provides public operations on jobs.
`Job_Types`	Declares types and type rights for jobs.
`Pipe_Mgt`	Manages pipes. A *pipe* is a one-way interprocess or interjob I/O channel. Pipes support byte stream I/O and record I/O.
`Process_Admin`	Provides more powerful `Spawn_process` and `Set_process_globals` calls for administrative users.
`Process_Mgt`	Provides public operations on processes.
`Process_Mgt_Types`	Declares types and type rights for processes.
`Semaphore_Mgt`	Manages semaphores. Semaphores can be used to synchronize concurrent access to shared data structures or resources.
`Session_Admin`	Provides administrative operations on sessions.
`Session_Mgt`	Provides public operations on sessions.
`Session_Types`	Declares types and type rights for sessions.

I-2.6.2 Scheduling Service

Manages scheduling of jobs and processes.

`SSO_Admin`	Provides calls to create and modify Scheduling Service Objects (SSOs).
`SSO_Types`	Defines job scheduling classes, Scheduling Service Objects (SSOs), and SSO messages. Also provides a function to determine whether an AD references an SSO.

I-2.6.3 Timing Service

Manages system time, timed requests, time computations, and time format conversions.

`Clock_Mgt`	Manages a node's system clock.
`Protection_Key_Mgt`	Manages protection keys.
`Time_Zone_Map`	Provides calls to map between time zones and time zone names.
`Timed_Requests_Mgt`	Supports the scheduling of timed requests at a node and provides access to the node's system clock.
`Timing_Admin`	Provides calls for manipulating timed request queues and setting the local time zone.
`Timing_Conversions`	Provides calls for converting between numeric representations of time and other representations, and for obtaining the local time zone.
`Timing_String_Conversions`	Provides calls for converting between string representations of time and other representations.
`Timing_Utilities`	Provides calls to inquire about timed requests.

I-2.6.4 Resource Service

Supports resource control and accounting.

`Resource_Mgt`	Provides distributed resource management.
`Resource_Mgt_AM`	Provides the type manager's interface to resource management, including the resource administration attribute.
`Resource_Types`	Defines constants and types used for resource management.
`Resource_Utilities`	Implements resource accounting.

I-2.6.5 Program Building Service

Supports programs that build or manipulate programs, such as compilers, linkers, and debuggers.

`Control_Types`	Defines a process's arithmetic controls, process controls, and trace controls.
`Debug_Support`	Supports the debugger by providing access to a process's domain object, static data object, instruction object, control stack, process controls, and other structures.
`Domain_Mgt`	Provides calls to check whether an AD references a domain object, an instruction object, a static data object, or a stack object.

`Execution_Support`	Supports the execution of executable objects.
`Link_By_Call`	Supports finding and calling an arbitrary subprogram at runtime. The subprogram must be in an image module or view and be interdomain-callable.
`Program_Mgt`	Supports program invocation and the retrieval of program-related information.
`RTS_Support`	Supports language-defined runtime systems (RTSs).

I-2.6.6 Monitor Service

Supports monitoring of program execution.

`Monitor_Defs`	Defines types used by `Monitor_Mgt`.
`Monitor_Mgt`	Manages monitors used to record information about program execution.

I-2.7 Type Manager Services

Type Manager Services contains:

TM object service
TM transaction service
TM concurrent programming service
configuration service
custom naming service
backup service.

I-2.7.1 TM Object Service

Provides object and memory operations for building advanced type managers.

`Countable_Object_Mgt`	Supports type managers of countable global objects.
`Global_SRO_Defs`	Provides access to the global SROs used to allocate global objects.
`Lifetime_Control`	Provides a trusted interface for creating and managing lifetime violations.
`PSM_Trusted_Attributes`	Defines the passive store trusted attribute.
`SRO_Mgt`	Provides memory management information and control of local garbage collection for local *storage resource objects* (SROs).
`Unsafe_Object_Mgt`	Provides special object allocation and deallocation calls.

I-2.7.2 TM Transaction Service

Manages transactions within a type manager.

`Local_Transaction_Defs`	Defines the per-object record used by instantiations of the `Local_ Transaction_Mgt` generic package.
`Local_Transaction_Mgt`	Provides transaction-oriented locking for type managers of local objects.

`TM_Transaction_Mgt`	Supports global transaction-oriented type managers that customize their participation in transactions. See `Transaction_Mgt` for a general description of the transaction service.

I-2.7.3 TM Concurrent Programming Service

Provides concurrent programming support for advanced type managers.

`Job_Resource_Reclamation`	Supports type managers that reclaim resources when a job is terminating.
`Port_Mgt`	Provides fast interprocess communication within a job.
`Typemgr_Support`	Supports masked type managers implementing blocking operations.
`Unsafe_Port_Mgt`	Provides unsafe deallocation for ports.
`Unsafe_Semaphore_Mgt`	Provides unsafe deallocation for semaphores.

I-2.7.4 Configuration Service

Defines how type managers configure themselves during system initialization, or dynamically reconfigure themselves at runtime.

`Configuration`	Provides operations for creating and modifying a system configuration.

I-2.7.5 Custom Naming Service

Supports custom directories and custom links.

`Customized_Name_Mgt`	Provides a call to retrieve the *name mapper* attribute and an `.Ops` interface package for implementing customized name mappers.
`Link_Mgt`	Provides a call to return the *link* attribute ID.
`Standalone_Directory_Mgt`	Provides the `Create_standalone_directory` call.

I-2.7.6 Backup Service

Supports type managers that participate in backup, logging, restore, or rollforward operations.

`Backup_Support`	Defines the backup attribute that trusted type managers can implement to support backup and recovery. *This package is not implemented in this release. Its specification is included to provide design information about the backup service.*
`Trusted_Log_Mgt`	Provides trusted type managers with calls for writing to the system logs. *This package is not implemented in this release. Its specification is included to provide design information about the backup service.*

I-2.7.7 Distribution Services

Distribution Services contain:

clearinghouse service
RPC service
transport service.

I-2.7.8 Clearinghouse Service

Manages the Clearinghouse, used to find network addresses, volume sets, and other information in a distributed system.

`CH_Admin`	Manages Clearinghouse administrators, and provides calls for administrators to add and delete servers, organizations, domains, and environments.
`CH_Client`	Manages Clearinghouse entries and entry properties.
`CH_Support`	Provides calls to find out about the Clearinghouse structure, to modify client key, and to check access to environments. Also provides calls to administer an environment.
`Node_ID_Mapping`	Provides calls for getting node ID and naming information.

I-2.7.9 RPC Service

Provides a *remote procedure call* (RPC) mechanism for communicating between instances of a service at different nodes in a distributed system.

`RPC_Admin`	Provides administrative RPC calls.
`RPC_Call_Support`	Provides functions for remote procedure calling.
`RPC_Mgt`	Provides a *remote procedure call* (RPC) facility used to implement distributed services.

I-2.7.10 Transport Service

Provides network-independent communication between nodes.

`Comm_Defs`	Contains common addressing and buffer definitions for the communication services. See also `TM_Comm_Defs` and `Subnet_Defs`.
`Datagram_AM`	Provides service-independent datagram communications.
`DG_Filter_Mgt`	Defines datagram I/O filter functions. See also `VC_Filter_Mgt` for virtual circuit I/O filter functions.
`Distributed_Service_Admin`	Provides calls for building universal distributed services.
`Distributed_Service_Mgt`	Provides features needed by distributed service implementers.
`ISO_Adr_Defs`	Defines ISO-specific NSAP, TSAP and TCP addresses, and several functions for converting between `Comm_Defs` byte addresses and ISO addresses.

`ISO_Config_Defs`	Defines parameter records and incident codes for creating ISO transport services, including direct and indirect subnetworks, and gateways. See also the `ISO_Adr_Defs` package.
`ISO_TM_Admin`	Provides operations to enable and disable communications tracing, and to get ISO-specific status information.
`TM_Comm_Defs`	Contains common definitions for transport services. See also the `Comm_Defs` and `Subnet_Defs` packages.
`VC_Filter_Mgt`	Defines virtual circuit I/O filter functions. See also `DG_Filter_Mgt` for datagram I/O filter functions.
`Virtual_Circuit_AM`	Provides service-independent virtual circuit communications.

I-2.8 Device Services

Device Services contains:

device driver service
shared queue service
asynchronous communication service
mass storage service
SCSI service
subnet service
HDLC service
LAN service.

I-2.8.1 Device Driver Service

Provides interfaces used to build device drivers.

`CP_IO_Defs`	Contains declarations used for communicating with Channel Processors (CPs).
`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.
`CP_Resources`	Defines the CP resources attribute.
`DD_Support`	Supports directly-connected device drivers.
`Handling_Support`	Provides calls to save and restore global registers.
`Interrupt_Handling_Support`	Manages interrupt handlers.
`IO_Messages_Defs`	Defines the I/O messages mechanism interface.
`IO_Messages_Ops`	Provides driver-independent I/O message calls for device drivers.
`Region_3_Support`	Provides a call for installing macrocode in Region 3.

I-2.8.2 Shared Queue Service

Supports device drivers using the shared queues mechanism for low-speed I/O.

Cluster_Service	Manages cluster servers.
IO_Shared_Queues	Defines the shared queues I/O mechanism.

I-2.8.3 Asynchronous Communication Service

Defines the OS asynchronous device driver.

Async_Defs	Defines the asynchronous device class.

I-2.8.4 Mass Storage Service

Defines the OS interface to mass storage drivers (various SCSI devices and future IPI devices).

Bus_Independent_Disk_Defs	Defines disk command and reply codes that are independent of any particular I/O bus, such as SCSI or IPI.
Bus_Independent_Streamer_Defs	Defines command and reply codes for streaming tape drives that are independent of any particular I/O bus, such as SCSI or IPI.
Bus_Independent_Tape_Defs	Defines command and reply codes for start/stop tape drives that are independent of any particular I/O bus, such as SCSI or IPI.
Mass_Store_Reply_Codes	Defines I/O message reply classes and reply codes for mass storage devices.
MS_Configuration_Defs	Defines I/O message command codes and reply codes used to configure mass storage device.

I-2.8.5 SCSI Service

Defines the bus-specific interface to the SCSI bus.

CP_SCSI_Defs	Defines CP resources and data structures used to communicate with an SCSI mass storage I/O subsystem.
CP_SCSI_Mgt	Defines type-checking calls for SCSI buses, controllers, and devices.
SCSI_Bus_Dependent_Defs	Defines bus-specific commands and replies for the SCSI (Small Computer System Interface) I/O bus.

I-2.8.6 Subnet Service

Supports network-independent communication between nodes within a subnet.

Carrier_Mgt	Defines communication *carriers* and functions for manipulating carriers. Communication carriers are used for carrying user data, protocol data, and local control information.

`Subnet_CL_AM`	Defines connectionless (CL) subnetwork I/O calls. For connection-oriented subnet calls, see the `Subnet_CO_AM` package.
`Subnet_CO_AM`	Common interface to connection-oriented (CO) subnet I/O calls.
`Subnet_Defs`	Contains definitions used by other subnetwork (subnet) communication packages. See also the `Comm_Defs` and `TM_Comm_Defs` packages.
`Trace_Defs`	Contains types and definitions for tracing communications. Each communications service provider defines the procedural interface for tracing its own communications; for an example, see the `ISO_TM_Admin` package.
`Trace_Support`	Manages tracing of network communications.

I-2.8.7 HDLC Service

Defines the OS interface to the HDLC protocol.

`HDLC_Mgt`	Manages HDLC subnetworks.

I-2.8.8 LAN Service

Defines the OS interface to Local Area Network protocols.

`CSMA_CO_Defs`	Defines parameters and codes for the Intel 82588 Local Area Network controller.
`Ethernet_LAN_Mgt`	Manages Ethernet subnetworks.
`IEEE8023_LAN_Mgt`	Manages IEEE 802.2/802.3 subnetworks.

ADA PROGRAMMING TECHNIQUES 3

Contents

This chapter shows you common Ada programming techniques used with system calls. (A future release of this manual will include a chapter on C programming techniques as well.) You should read this chapter before reading any subsequent chapters in this manual, because many examples throughout the manual depend on these concepts and techniques.

I-3.1 Concepts

This section introduces several concepts that are prerequisites to understanding the programming techniques presented.

I-3.1.1 Working with Pointers

Many system calls require or return pointers to objects managed by the OS. A pointer to an object is called an *access descriptor* (AD) and contains rights bits that control access to the object. An AD is defined for each OS object type. The AD is actually a BiiN™ Ada *access type*. For example, an AD for a job is defined in `Job_Types`:

```
type job_AD is access job_object;
```

By convention, access types that consist of ADs to a particular type of system object are named with the suffix `_AD`.

Only the OS can access the internals, or *representation*, of objects managed by the OS. Your application can only perform those operations allowed by the system calls defined for a particular object type.

Another type of pointer, besides an AD, is a virtual address. Some system calls require that the caller supply the virtual address of a buffer or record to be used by the call. Such parameters may have the type `System.address`.

A virtual address represents an AD to an object and a 32-bit byte offset within the object. By convention, access types or parameters that are virtual addresses are named with the suffix `_VA`. For example, in the package `Record_AM`:

```
type operation_status_VA is
    access operation_status_record;
  -- Virtual address of an operation status record.
  pragma access_kind(operation_status_VA, virtual);
```

I-3.1.2 Common Types in the `System` and `System_Defs` Packages

The built-in BiiN™ Ada `System` package and the `System_Defs` support package define types used for many different system calls. In the `System` package:

- `System.untyped_word` is the type used to represent any 32-bit quantity. It may be a data value, or more typically an AD to any object.
- `System.address` is the type used to contain any virtual address.
- `System.subprogram_type` is the type used to contain a pointer to a procedure or function.
- `System.null_word`, `System.null_address`, and `System.null_subprogram` are the null values for the preceding three types.
- `System.ordinal`, `System.short_ordinal`, and `System.byte_ordinal` are 32-bit, 16-bit, and 8-bit unsigned integer types respectively. All integer operators are supported for these types, but without overflow checking.

In the `System_Defs` package:

- `System_Defs.text` is the type used as a container for strings passed to and from system calls. `System_Defs.null_text` is a zero-length text.
- `System_Defs.system_time_units` is the type used by the OS to measure times and durations.

I-3.1.3 Standard System Exceptions

The `System_Exceptions` support package defines exceptions commonly encountered when making system calls. You should read and understand the exception descriptions in `System_Exceptions` in the *BiiN™/OS Reference Manual.*

I-3.1.4 Package-level and Subprogram-level Variables

Variables can be declared as either *package-level* or *subprogram-level* variables.

A package-level variable is declared inside a package, but outside any subprogram. A subprogram-level variable is declared within a procedure or function.

The lifetime of package-level variables is independent of the invocation of any subprogram inside the package. If the code is shared by multiple processes, these variables are visible to all processes. On the other hand, subprogram-level variables exist only for the duration of the particular subprogram call.

It is recommended that you avoid package-level variables for cases where multiple processes will access the code. Using such variables without careful synchronization between processes can corrupt the variables.

I-3.2 Techniques

This section shows you how to:

- Use unchecked type conversion
- Use overlays as an alternative to unchecked type conversion
- Import operators
- Allocate a buffer
- Recover from record overflow
- Handle recoverable exceptions
- Use paired calls.

All techniques are illustrated with excerpts from compiled examples. Appendix X-A contains complete listings for all examples.

I-3.2.1 Using Unchecked Type Conversion

In different contexts, the OS may require different BiiN™ Ada types to be used for the same value. If these BiiN™ Ada types are not compatible according to BiiN™ Ada's type conversion rules, then an *unchecked conversion* between types is required.

Declarations Used: `Unchecked_conversion`
BiiN™ Ada generic function that does compile-time conversion between otherwise incompatible types.

For example, the BiiN™ Ada type used to contain any AD is `System.untyped_word`. Some system calls that can operate on any AD use `untyped_word` as the BiiN™ Ada type of their parameters or their returned value. Unchecked conversion can convert between other AD types and `untyped_word`.

To convert between two types with unchecked conversion:

1. The BiiN™ Ada `Unchecked_conversion` unit must be in your unit's `with` clause.
2. Your program must instantiate `Unchecked_conversion` to create a new type conversion function.
3. Your program uses the new type conversion function to do the conversion.

In the following example, which changes the current directory in a caller's process globals, the value needs to be converted from type `directory_AD` to type `untyped_word`. These steps are shown in the following excerpts from the `List_current_directory_cmd_ex` example:

```
  1  with Byte_Stream_AM,
 . . .
  9       Unchecked_Conversion;
 . . .
 11  procedure List_current_directory_cmd_ex
 . . .
 37  is
 . . .
 41    function Directory_AD_from_untyped_word is
 42        new Unchecked_conversion(
 43            source => System.untyped_word,
 44            target => Directory_Mgt.directory_AD);
 . . .
 79  begin
 . . .
 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);
 . . .
126  end List_current_directory_cmd_ex;
```

Line 9 includes `Unchecked_conversion` in the unit's `with` clause.

Lines 44-47 create the `Untyped_word_from_directory` function. The function created by specifying `Unchecked_conversion` accepts one parameter, of the source type, and returns a result of the target type.

Line 96 shows a call to the new function, required because `Process_Mgt.Get_process_globals_entry` uses the `untyped_word` type for the value returned, while `Directory_Mgt.Open_directory` requires a value of type `directory_AD`. Such calls are evaluated at compile-time and have no run-time cost.

I-3.2.2 Using Overlays as an Alternative to Unchecked Type Conversion

BiiN™ Ada provides an *overlay* feature that allows a programmer to specify the memory address of a variable, rather than relying on the compiler to allocate storage and determine the variable's address. A variable can be given the same address as a previously declared parameter or variable, providing different BiiN™ Ada types for the same value. The different names can be used as an alternative to performing unchecked type conversions where different BiiN™ Ada types are needed.

CAUTION

Overlays can be dangerous if used in an unstructured manner because this technique voids all of the strong typing of BiiN™ Ada. Serious programming errors can result.

This excerpt is from the `Show_current_directory_cmd_ex` example:

```
10  procedure Show_current_directory_cmd_ex
. . .
29  is
. . .
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.
. . .
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.
. . .
51  begin
52
53    -- Get current directory's pathname:
54    --
55    Directory_Mgt.Get_name(
56        obj  => current_dir_untyped,
57        name => dir_name);
. . .
73  end Show_current_directory_cmd_ex;
```

Lines 43—45 show how an overlay with a different type is declared. The local variables `current_dir` and `current_dir_untyped` name the same word in memory, but with different types. The name `current_dir` is used wherever the type `Directory_Mgt.directory_AD` is required. The name `current_dir_untyped` is used wherever the type `System.untyped_word` is required.

Line 56 shows `dir_untyped` used in the call to `Directory_Mgt.Get_name`.

I-3.2.3 Importing Operators

Some BiiN™ Ada operators (`"="`, `"+"`, and so on) are defined for many types declared in System Services packages or the BiiN™ Ada `System` package. The following rules indicate where operators are defined:

1. The package that defines an access type also defines `"="` and `"/="` for that type.
2. The package that defines an enumeration type also defines all relational operators for that type. If a subtype of the enumeration type is declared in another package, the subtype still uses the operators defined in the first package for the base enumeration type.

3. The package that defines a record type also defines "=" and "/=" for that type.
4. The System package defines for all ordinal types all operators allowed for integers.
5. The Long_Integer_Defs package defines for long integers all operators allowed for integers.
6. The Text_Mgt package defines the operators "<", "<=", ">", and ">=" for the System_Defs.text record type. The System_Defs package implicitly defines "=" and "/=" for texts.
7. Other OS packages may define additional operators for their types.

Your program can use such operators in two ways:

1. Explicitly qualify each use with the package name.
2. Import the package that defines the operators with a use clause, and then use the operators normally.

In either case, the packages must be listed in your program's with clause.

If A, B, and C are long integer variables, the following code fragments show how to write A := B + C; using the two techniques. First, using explicit qualification:

```
A := Long_Integer_Defs."+"(B, C);
```

Note that BiiN™ Ada syntax requires that a qualified operator be quoted and does not allow an operator qualified with a package name to be used as an infix operator. The following code is WRONG:

```
A := B Long_Integer_Defs."+" C;
```

The next code fragment shows importing Long_Integer_Defs with a use clause and then doing the computation:

```
use Long_Integer_Defs;
...
A := B + C;
```

Recommended coding practice is to import packages if operators from the package must be used, with these restrictions:

1. References to any other calls or declarations in the package should still be fully qualified. For example, write Long_Integer_Defs.zero instead of just zero .
2. If operators from a particular package are used only in a particular call or code block, then use the same scope for the use clause.

use clauses should be used only where necessary. If names of entities declared in other packages are not fully qualified with their package names, then your program is harder to understand and harder to maintain.

I-3.2.4 Allocating a Buffer

When you need to set up a buffer to process the results of a system call, there are two main options:

Allocate a buffer as a local variable.
This is the recommended option, because allocation is fast and the buffer can be reclaimed as soon as control exits the subprogram or block.

Create an object for the buffer.

This is useful when the buffer may need to be resized, as it might be when reading variable-length records. A disadvantage is the overhead required to allocate and deallocate an object.

The following excerpt from the example `List_current_directory_cmd_ex` shows how to allocate a buffer as a local variable. In this example, the buffer holds each entry name between reading it and writing it.

```
procedure List_current_directory_cmd_ex
. . .
is
. . .
  name_buffer:  array(1 .. 250) of character;
    -- Each entry name is read into this buffer
    -- and then written from it.
. . .
begin
. . .
  -- Get and write each entry name:
  --
  loop

    length := Byte_Stream_AM.Ops.Read(
        opened_dev => opened_dir,
        buffer_VA  => name_buffer'address,
        length     => name_buffer'size/8);

    Byte_Stream_AM.Ops.Write(
        opened_dev => standard_output,
        buffer_VA  => name_buffer'address,
        length     => length);

  end loop;
. . .
end List_current_directory_cmd_ex;
```

See `Output_records_ex` in Appendix X-A for an example of a dynamically sized buffer contained in a separate object.

I-3.2.5 Recovering from Record Overflow

Some system calls assign array values to fixed-length records supplied by callers as `out` parameters. If an array value is too large, these system calls simply assign all the values that will fit in the record and assign the total array's length to a field in the record.

Checking for such "record overflows" is the caller's responsibility. Any system call that assigns a text record behaves in this way; record overflow assigns an invalid text with `length` > `max_length`. "Information" calls that return lists of processes in a job, jobs in a session, entries in an authority list, or other varying-length arrays can overflow in the same way.

The following excerpt from the `Stored_Account_Mgt_ex` example package body shows how an application checks for text record overflow and retries if necessary. The same technique can be used to handle similar overflows for other record types.

The text record is declared and used in a nested block. To retry, the variable controlling the text's size is increased and control jumps back to the beginning of the block, reentering the block and reallocating the text with the new size. Because the text record is declared in a nested block, it can only be used within that block.

```
procedure Destroy_account(
. . .
is
. . .
begin
. . .
  loop
. . .
      path_length:  integer := 60;
        -- Initial text length for name assigned
        -- by "Directory_Mgt.Get_name".  If
        -- insufficient, then the value is
        -- increased and the operation is
        -- repeated.
. . .
      loop
        declare
          path_text:  System_Defs.text(path_length);
        begin
          Directory_Mgt.Get_name(
              obj  => account_untyped,
              name => path_text);  -- out.
          if path_text.length >
             path_text.max_length then
            -- Text was lost.  Retry:
            path_length := path_text.length;
          else
            Directory_Mgt.Delete(path_text);
            EXIT;

          end if;
        exception
          when Directory_Mgt.no_name =>
            EXIT;

        end;
      end loop;
. . .
  EXIT;
  end loop;
. . .
end Destroy_account;
```

I-3.2.6 Handling Recoverable Exceptions

Most exceptions raised by system calls cannot be recovered from. When an exception *can* be recovered from, this section describes a specific coding technique for recovery.

If an exception occurs, execution of the surrounding block is abandoned and any exception handler is entered. An exception handler cannot jump back into the abandoned code to retry an operation. Thus, if an operation needs to be retried in some cases, then the operation should be placed in a nested block within a loop. The nested block can then handle the possible cases:

- **The operation is successful.** In this case, a `return` or `exit` statement can exit the loop.
- **The operation is not successful and a recoverable exception is raised.** In this case, the nested block contains a handler for the exception. After handling the exception, control loops back and the operation can be retried.
- **The operation is not successful and a non-recoverable exception is raised.** In this case, the exception is simply propagated.

Using a nested block to retry an operation and handle a recoverable exception is illustrated by the following excerpt from the `Stored_Account_Mgt` example package body:

```
loop
  if Transaction_Mgt.Get_default_transaction =
      null then
    Transaction_Mgt.Start_transaction;
    trans := true;
  end if;
  begin
. . .
      Passive_Store_Mgt.Update(account_untyped);
      if trans then
        Transaction_Mgt.Commit_transaction;
      end if;
      RETURN account_rep.balance;
. . .
  exception
    when System_Exceptions.
         transaction_timestamp_conflict =>
      if trans then
        Transaction_Mgt.Abort_transaction;
      else
        RAISE;

      end if;
    when others =>
      if trans then
        Transaction_Mgt.Abort_transaction;
      end if;
      RAISE;
  end;
end loop;
```

I-3.2.7 Using Paired Calls

Some important system calls must be carefully paired for your application to work properly. Some common pairings are:

- In `Semaphore_Mgt`: `P` (`Lock`) with `V` (`Unlock`).
- In `Transaction_Mgt`: `Start_transaction` with either of `Commit_transaction` (if successful) or `Abort_transaction` (if unsuccessful).
- In each I/O access method package: `Open` with `Close`. If `Close` is omitted, an opened device is closed when all jobs using it terminate.

Pairing system calls is complicated by exceptions, which can cause unexpected transfers of control out of a code block. If a matching system call *must* be executed before leaving a block, use a `when others` exception handler, as in the following excerpt from the `Symbol_Table_Ex` example:

```
begin
  Semaphore_Mgt.P(symbol_table.lock);
. . .
    exception
      when others =>
        Semaphore_Mgt.V(symbol_table.lock);
        RAISE;
        -- Reraise exception that entered handler.
    end;

  Semaphore_Mgt.V(symbol_table.lock);
```

I-3.3 Summary

- Many system calls require or return pointers to objects managed by the OS.
- A pointer to an object is called an *access descriptor* (AD) and contains rights bits that control access to the object.
- By convention, access types that consist of ADs to a particular type of object are named with the suffix `_AD` .
- By convention, access types or parameters that are virtual addresses are named with the suffix `_VA` .
- The built-in BiiN™ Ada `System` package and the `System_Defs` support package define types used for many different system calls.
- The `System_Exceptions` support package defines exceptions commonly encountered when making system calls.

Part II
Support Services

This part of the *BiiN™/OS Guide* shows you how to use Support Services, needed throughout all other service areas for basic tasks. The chapters in this part are:

Using Utility Packages	Gives data structures and examples for common system types and operations.
Using Objects and ADs	Shows basic techniques for using objects and ADs.
Storing Objects	Shows how to use the system's distributed storage system to store objects on disk.
Starting and Resolving Transactions	Shows how to use transactions to group operations so that either all operations in the group succeed or all are rolled back.
Writing Messages	Shows how to use the system's facilities for writing messages. The message service allows messages to be expressed in different languages and edited without access to source code.

Support Services contains the following services and packages:

utility service:
```
Long_Integer_Defs
Machine_Code_Insertion
String_List_Mgt
System
System_Defs
System_Exceptions
Text_Mgt
```

object service:
```
Access_Mgt
Attribute_Mgt
Object_Mgt
Passive_Store_Mgt
```

transaction service:
```
Transaction_Mgt
```

message service:
```
History_Services
Incident_Defs
Message_Adm
Message_Services
Message_Stack_Mgt
Msg_Object_Defs
System_Error_Recording
```

Part II Overview

USING UTILITY PACKAGES 1

Contents

Utility packages are used throughout the system by services in all the services areas. Many system calls require parameters of the types in these packages, such as `text`, `string list`, and `long integer`. This chapter shows how to perform common operations using these types.

Packages Used:

`String_List_Mgt`	Provides operations on string lists.
`Text_Mgt`	Provides operations on text records.
`Long_Integer_Defs`	Defines types and calls for 64-bit long integers.
`System_Defs`	Provides common definitions used throughout the OS.

Figure II-1-1 shows the data structures for the objects discussed in this chapter.

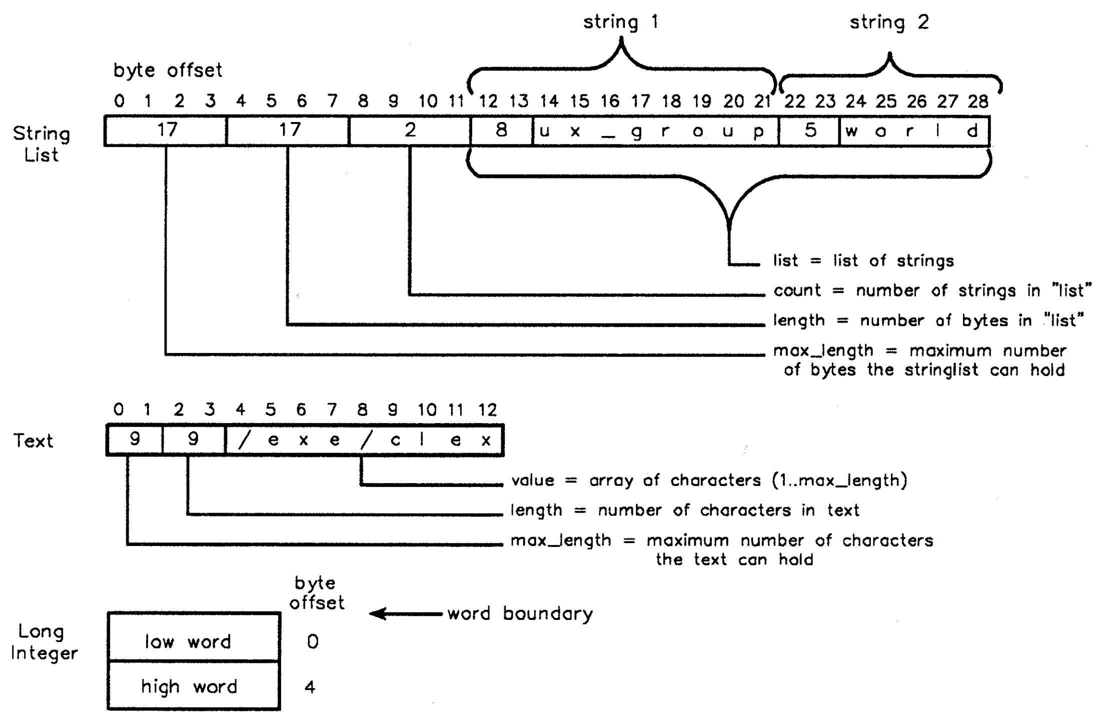


Figure II-1-1. Data Structures for String List, Text, and Long Integer

II-1.1 Concepts

II-1.1.1 String Lists

A *string list* is a standard container for a list of strings. String lists are often used with text records.

String lists have the `System_Defs.string_list` type. A particular string list has a fixed size and can contain any string values that will fit. An individual string in a string list is preceded by a two-byte length field and can have from 0 to 32,767 characters.

A string list contains these fields:

`max_length`	A discriminant, specifying the maximum number of bytes that the `list` field can hold.
`length`	The number of bytes used in the `list` field.
`count`	The number of strings in the string list.
`list`	An array of `max_length` characters, indexed from 1 to `max_length`.

`list(1 .. length)` contains the strings in the list. Each string is contained in a record of type `System_Defs.var_text`, a two-byte length followed by the specified number of characters. Successive strings are packed with no unused bytes and no alignment requirements.

`list(length + 1 .. max_length)` is the free space available in the string list.

System calls that retrieve string lists do so via `out` parameters. If a string list is larger than the space in a particular string list record, then such calls assign the actual length of the string list to the `length` field, the actual number of strings to the `count` field, and the strings that will actually fit (without breaking in mid-string) to the `list` field. This sort of overflowing string list is an *invalid string list*. It is the application's responsibility to check for invalid string lists where they can occur.

See also `String_List_Mgt`.

II-1.1.2 Texts

A *text* is the standard container for a string.

Texts have the `System_Defs.text` type. A particular text can hold a fixed-size string and contains a value of any length up to that size. Text sizes range from 0 to 32,767 characters.

A text contains these fields:

`max_length`	A discriminant, specifying the maximum number of characters that the text can hold.
`length`	Number of characters actually used, *or* if greater than `max_length`, the number of characters needed in the text.
`value`	An array of `max_length` characters, indexed from 1 to `max_length`.

Many system calls assign strings to text `out` parameters. If the string is larger than the space in the text (overflow), such calls assign the actual size of the string to the `length` field and assign the first `max_length` characters of the string to the `value` field. Such a text value, with `length` > `max_length`, is an *invalid text*.

See Chapter I-3 for a technique to recover from such a text overflow.

See also `Text_Mgt`.

II-1.1.3 Long Integers

A *long integer* is represented as a record of two ordinals.

Long integers are of type `Long_Integer_Defs.long_integer` and range from -(2**63) to (2**63)-1.

The range of long integers in decimal is:

```
-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
```

Note that the record's representation reverses the order of the `h` and `l` fields, so that the low word is first in memory followed by the high word. This representation is consistent with the representation used for all other multi-byte integer and ordinal types: the least significant byte is at the lowest memory address, followed by the next most significant byte, etc. The most significant byte is always at the highest memory address used.

See also `Long_Integer_Defs`.

II-1.2 Techniques

After reading this section, you will be able to:

Use a literal text

Declare a constant text

Call a procedure with a text result

Create a string list

Read elements from a string list

Use a literal long integer

Compute with long integers

Convert between strings and long integers.

Complete listings of the programs used in the following examples can be found in Appendix X-A.

II-1.2.1 Using a Literal Text

The following example from the inventory example program (module `Inventory_Messages`) shows the use of a literal text:

```
message_file:  constant System_Defs.text_AD :=
    new System_Defs.text'(
        31,31,"/example/inventory/message_file");
  -- AD to message file text name.
```

II-1.2.2 Declaring a Constant Text

The following example from the inventory example program (module `Inventory_Reports`) shows the declaration of a constant text:

```
report_by_part_DDef_str:  constant string :=
    "/example/inventory/DDefs/report_by_part";
  -- String constant for "report by part"
  -- report DDef's pathname.

report_by_part_DDef_pathname:
    System_Defs.text(
        report_by_part_DDef_str'length) := (
            report_by_part_DDef_str'length,
            report_by_part_DDef_str'length,
            report_by_part_DDef_str);
  -- Text constant from "report by part"
  -- DDef's pathname string.
```

II-1.2.3 Calling a Procedure with a Text Result

After calling a procedure that retrieves a text result, be sure to check for an invalid text (not enough space to accommodate the desired text). There are many system calls that return a result of type `text`; `Directory_Mgt.Get_name` is just one.

Calls Used:

`Directory_Mgt.Get_name`
Gets the full pathname of an object's master AD.

The following example is from the `Stored_Account_Mgt_Ex` example package body in Appendix X-A.

```
loop
  declare
    path_text:  System_Defs.text(path_length);
  begin
    Directory_Mgt.Get_name(
        obj  => account_untyped,
        name => path_text);  -- out.
    if path_text.length >
       path_text.max_length then
      -- Text was lost.  Retry:
      path_length := path_text.length;
    else
      Directory_Mgt.Delete(path_text);
      EXIT;

    end if;
  exception
    when Directory_Mgt.no_name =>
      EXIT;

  end;
end loop;
```

In the above example, note how the developer enclosed the call within a nested block to check if all the characters in the desired text actually fit into the destination text.

The parameter to receive the text is called `path_text`. After the pathname is received from `Directory_Mgt.Get_name`, the values `path_text.length` and `path_text.max_length` are compared to see if the number of bytes in the text was greater than the maximum specified for the text. If so, then the text is resized to the higher size and `Directory_Mgt.Get_name` is repeated.

II-1.2.4 Creating a String List

The simplest way to create a string list is to use `String_List_Mgt` and build the string list from texts.

Calls Used:

`String_List_Mgt.Set`
Copies a text to a string list.

`String_List_Mgt.Append`
Appends a text to a string list.

The following example shows how to create a string list.

```
with String_List_Mgt,
     System_Defs;

procedure String_list_ex
  --
  -- Function:
  --   Create string list with following entries:
  --     1. "ux_group"
  --     2. "world"
is
  string_list:  System_Defs.string_list(255);
begin

  -- 1) "ux_group"
  String_List_Mgt.Set(string_list,
      System_Defs.text'(8, 8, "ux_group"));

  -- 2) "world"
  String_List_Mgt.Append(string_list,
      System_Defs.text'(5, 5, "world"));

end String_list_ex;
```

II-1.2.5 Reading Elements from a String List

The package `String_List_Mgt` provides calls to access the strings in a string list.

`Get_element` retrieves a string from a string list, given its position number.

`Get_element_by_index` retrieves a string from a string list given an index variable, which it updates. Both can be used to loop through all strings in a string list. `Get_element` is simpler to use. `Get_element_by_index` executes more quickly, especially for large string lists.

`Locate` finds a string and returns its position number.

`Locate_index` finds a string and returns its index.

Calls Used:

`String_List_Mgt.Get_element_by_index`
Gets the string with a specified index from a string list, and updates the index variable to reference the next string.

The following excerpt from the `Create_name_space_cmd_ex` example shows how `Get_element_by_index` can loop through all strings in a string list:

```
i := 1;

loop

  String_List_Mgt.Get_element_by_index(
      from       => directory_list,
      list_index => i,
      element    => directory_path);

  -- Exit after last string:
  --
  EXIT when i = 0;
...
end loop;
```

II-1.2.6 Using a Literal Long Integer

The following example from the `Long_Integer_Ex` example package shows the use of a literal long integer:

```
-- Declaring a negative long integer constant,
-- the easy way and the hard way:
--
negative_twenty:  constant long_integer :=
    - long_integer'(0, 20);

another_negative_twenty:  constant long_integer :=
    (16#ffff_ffff#, 16#ffff_ffec#);
  -- Use the hard way when you want a declaration
  -- elaborated at compile-time instead of
  -- at run-time.
```

II-1.2.7 Computing with Long Integers

All standard Ada arithmetic and relational operators are defined for long integers. The = and /= operators are implemented using Ada record comparison. All other long integer operators are defined in the package `Long_Integer_Defs`.

To use long integer operators, a program unit must explicitly `use Long_Integer_Defs`.

Long integers do not support Ada attributes of integers. A long integer is a record type, hence the usual Ada attributes defined for integers cannot be applied. The following list gives long integer alternatives to Ada attributes of integers:

`'first`	Use `Long_Integer_Defs.min_int`.
`'last`	Use `Long_Integer_Defs.max_int`.
`'width`	At most 20 characters are required to represent a long integer as a string. `Long_Integer_Defs` uses 31 characters, which allows ample space for embedded underscores.
`'pos`	Not needed; `x'pos = x` for integer types.
`'val`	Not needed; `x'val = x` for integer types.
`'succ`	Add one.
`'pred`	Subtract one.
`'image`	Call `Long_Integer_Defs.Long_integer_image`.

`'value`	Call `Long_Integer_Defs.Long_integer_value`.

II-1.2.8 Converting Between Strings and Long Integers

To convert between a string type and a long integer type, `Long_Integer_Defs` provides two straightforward calls:

Calls Used:

`Long_Integer_Defs.Long_integer_image`	Converts a long integer to a string image.
`Long_Integer_Defs.Long_integer_value`	Converts a string image to a long integer.

See the `Long_integer_value` function in the `Long_Integer_Ex` package in Appendix X-A for an example of code that converts a string image to a long integer. (The code is too long to include here.)

II-1.2.9 Summary

- Many system calls require parameters of the types `text`, `string list`, and `long integer`.
- A *text* is the standard container for a string.
- A *string list* is a standard container for a list of strings.
- A *long integer* is represented as a record of two ordinals. All standard Ada arithmetic and relational operators are defined for long integers.

USING OBJECTS AND ADS 2

Contents

II-2.1 Concepts

This chapter presents an overview of objects and access descriptors. Objects are the fundamental units of object-oriented programming. Access descriptors are pointers that reference these objects.

Packages Used:

`Access_Mgt`	Interface for checking or changing rights in access descriptors.
`Object_Mgt`	Provides basic calls for object allocation, typing, and storage management. Defines access rights in ADs.

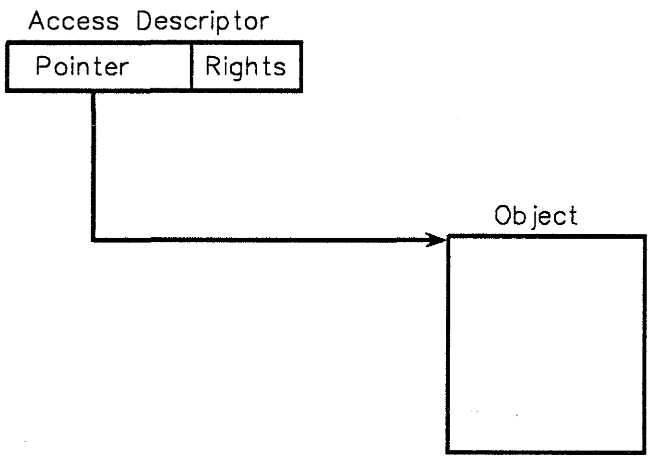


Figure II-2-1. AD and Object

II-2.1.1 What is an Object?

An *object* is a typed and protected memory segment. An object has the following characteristics:

A unique identity that cannot be forged, and is guaranteed to exist as long as references to the object exist

A *type*, determined by a Type Definition Object (TDO)

A *representation*, an area of active memory or passive store that holds the object's contents

A *Storage Resource Object* (SRO) from which the object is allocated and to which its memory space is returned when deallocated

A *lifetime* that determines whether an object's existence is limited to the lifetime of a single job or is indefinite

A *memory type* that determines whether the object will reside in normal (swappable) memory or frozen (non-swappable) memory.

The size of an object may be from 0 to 2^{32} bytes, and can be dynamically changed. Up to 2^{26} objects are possible in a node's four gigabytes of active memory.

A *type manager* is a routine that provides basic operations for all objects of its type. A TDO defines a type manager's *type*. Object types are specified when objects are created, and cannot be changed.

Only a type manager can create objects of its type, or read or write to the object representation. Other services can reference objects of a type manager's type, but must call the type manager to read or change the objects. Therefore, the integrity of type-managed objects depends only on the type manager, and not on other services that use the object.

II-2.1.2 What is an Access Descriptor?

An *access descriptor* is a protected pointer to an object, with rights describing how the object can be used. Figure II-2-2 shows an access descriptor layout.

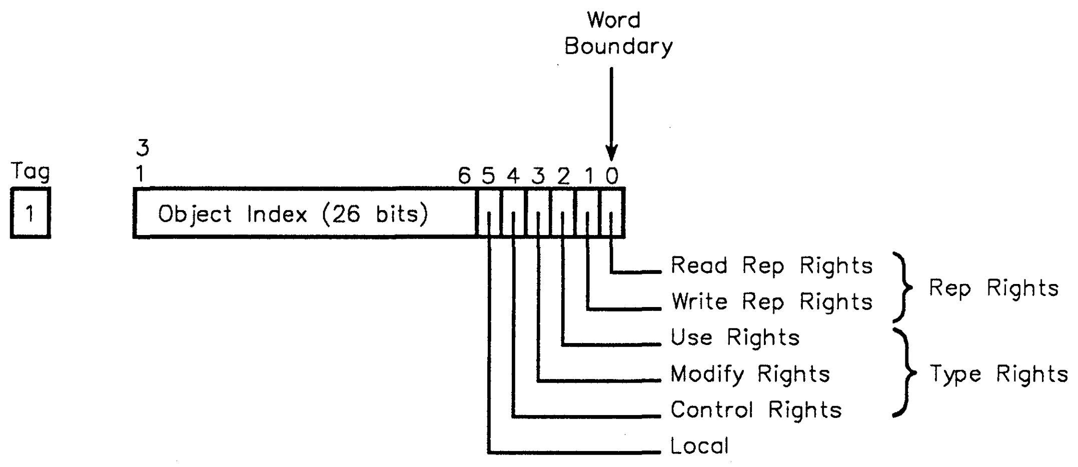


Figure II-2-2. A Valid Access Descriptor

An AD is represented by a memory word with Ada type `System.untyped_word`. An `untyped_word` is a one-word, word-aligned value corresponding to one 33-bit memory word, 32 bits of information and a *tag bit*. (The tag bit indicates whether the word is a valid AD.) An untyped word can be interpreted as either an access descriptor word or as a non-AD word.

If interpreted as an AD, an `untyped_word` contains:

- A valid AD that references an object and provides rights to the object when the tag bit is 1
- A null AD when the tag bit is 0, regardless of the value of the first 32 bits.

If interpreted as a non-AD, an `untyped_word` contains 32 bits of data and the tag bit is ignored.

By convention, access descriptor names end with `_AD`. For example, `directory_AD` is an access descriptor that references a directory object.

An AD contains five rights bits: two representation rights and three type rights. These rights are described in the following sections.

II-2.1.3 Rep Rights Control Access to an Object's Representation

Representation rights are required to read or write the object's representation. These rights grant access to an object's physical layout in memory. Rep rights are checked by the CPU whenever a program reads or writes memory. If needed rep rights are not present, then `System_Exceptions.insufficient_rep_rights` is raised. If an object has a type manager, then the type manager normally removes rep rights on any ADs it exports for the object. A type manager can turn on ("amplify") rep rights on ADs for objects that it manages.

II-2.1.4 Type Rights Control What Type-Specific Operations are Allowed

Type rights are specific to the object type and provide access to an object's logical structure by determining what type manager calls are allowed on that object. The three rights are designated *use*, *modify* and *control* by convention to ensure that they are interpreted consistently regardless of object type.

Use - To get information about the object

Modify - To change the contents of an object but retain its existence and representation

Control - To destroy the object, or perform other privileged operations.

Use and *modify* rights correspond to *read* and *write* rights for files. *Control* rights give the user maximum control over the object. The actual functions of these rights is determined by the type manager. Usually, these types are renamed to reflect the particular usage of the rights. For example, the naming service defines *list* and *store* rights for directories, which correspond to *use* and *modify* rights.

These rights do not form a hierarchy in that a type manager may provide any one or more of them. For example, a type manager could interpret modify rights and not interpret use or control rights. rights.

Checking and enforcing type rights is done by type managers and not by the CPU. A type manager raises `System_Exceptions.insufficient_type_rights` if needed type rights are missing.

II-2.1.5 Generic Objects

When an object is created without specifying a particular TDO, the new object is generic and is associated with the generic TDO. A *generic* object is used only as a memory segment. It does not have a type manager, and has the same TDO as all other generic objects. The generic TDO is held by the BiiN™ Operating System. Applications can create and manage generic objects.

II-2.1.6 Building Type Managers That Define New Object Types

Type managers can be built to support new object types. A new type is defined by creating a new TDO. See Chapter VII-3 for instructions for building a type manager for a new type.

II-2.2 Techniques

After reading this section, you will be able to:

- Check an object's type
- Check rights on an AD
- Remove rights from an AD
- Create a generic object
- Resize an object
- Get an object's size
- Deallocate an object.

II-2.2.1 Checking an Object's Type

An application can check that an object is of a particular type before attempting to perform type-specific operations on it. Each BiiN™ Operating System type manager provides an `Is_` call that checks whether an AD points to an object of the managed type. The following code is from the `Process_Globals_Support_Ex` example. After the process global entry for the home directory is retrieved, the following code checks to verify that it is a directory object.

```
if not Directory_Mgt.Is_directory(dir_untyped) then
  RAISE System_Exceptions.type_mismatch;
```

II-2.2.2 Checking Rights on an AD

The access rights of an AD can be examined. The rights to be checked for are set in `Object_Mgt.rights_mask`. Then `Access_Mgt.Permits` is called. True is returned if the AD has the rights which were set in the mask.

II-2.2.3 Removing Rights From an AD

The access rights of an AD can be removed. The rights to be removed are set in a rights mask. Then `Access_Mgt.Remove` is called. An AD to the same object is returned without the rights set in the mask. The following code is from the `Account_Mgt_Ex` example. While creating a new account, all rep rights are removed from the returned AD thus requiring all reading and writing of the account object to be performed via the account type manager.

```
account_untyped := Access_Mgt.Remove(
    AD     => account_untyped,
    rights => Object_Mgt.read_write_rights);
```

II-2.2.4 Creating a Generic Object

A new object can be allocated with `Object_Mgt.Allocate`. If no TDO is specified in the call, the new object is generic. Another method for creating objects is to use the Ada `new` allocator. The following example allocates a symbol table object with space for `table_size` entries and no entries in use.

```
symbol_table := new symbol_table_object(
    table_size);
symbol_table.length := 0;
  -- Symbol table initially has space for 100
  -- entries with 0 in use.
```

II-2.2.5 Resizing an Object

`Object_Mgt.Allocate` allocates an object at a specified size. That size can be determined with `Object_Mgt.Get_object_size`. (Object size is specified in words.) The size can also be changed with `Object_Mgt.Resize` as shown in the following example:

```
Object_Mgt.Resize(
    obj  => symbol_table_untyped,
    size => 3 + (2 * symbol_table.max_length * (
        symbol_entry'size/32)));

max_length_access := 2 * symbol_table.max_length;
```

In this example, a symbol table is expanded so that it will hold twice as many entries and changes the maximum length of a symbol table entry.

II-2.2.6 Deallocating an Object

Objects can be dynamically deallocated. The following excerpt from `Account_Mgt_Ex` shows an account object checked for a zero balance and then destroyed (deallocated).

```
begin
  account_untyped :=  Access_Mgt.Import(
        AD     => account_untyped,
        rights => destroy_rights,
        tdo    => account_TDO);

  if account_rep.balance /= Long_Integer_Defs.zero then
    RAISE balance_not_zero;

  else
    Object_Mgt.Deallocate(account_untyped);

  end if;
```

II-2.3 Summary

- An *object* is a typed and protected memory segment.
- An *access descriptor* is a protected pointer to an object, with rights describing how the pointer can be used.
- An object's *representation* is an area of active memory or passive store that holds the object's contents.
- A *type manager* is a routine that provides basic operations for all objects of its type.
- A *TDO* defines a type manager's type.
- *Type rights* are specific to the object type and provide access to an object's logical structure by determining what type manager calls are allowed on that object.
- *Representation rights* are required to read or write the object's representation.
- A *generic* object is used only as a memory segment. It does not have a type manager, and has the same TDO as all other generic objects.

STORING OBJECTS 3

Contents

Passive store is the collection of objects stored on disk in a BiiN™ system. This chapter shows you how to store objects on disk.

Packages Used:

`Directory_Mgt`	Manages directories and directory entries.
`Passive_Store_Mgt`	Provides a distributed object filing system.

Passive store is distributed--spread over multiple nodes and transparently accessible from any node (Figure II-3-1). Of course passive store is equally usable on a single-node system.

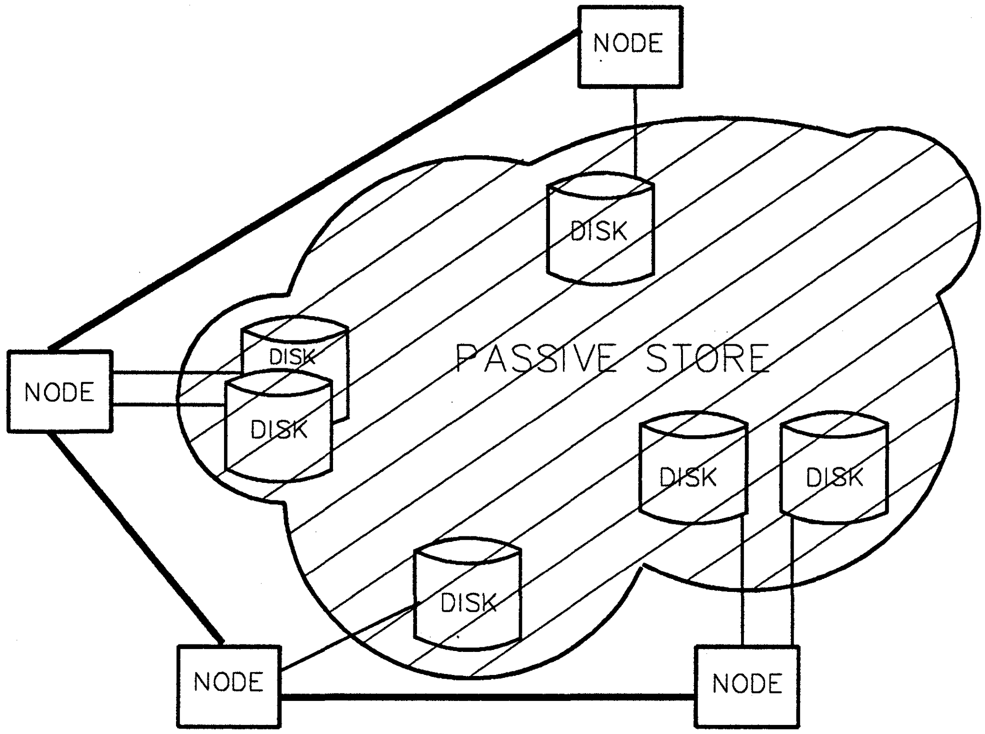


Figure II-3-1. Passive Store is a Distributed Object Filing Service that Unifies all Nodes in a BiiN™ System.

Passive store is reliable--stored objects survive system crashes and changes to stored objects are transaction-oriented.

In many ways passive store is the "glue" that holds together a distributed BiiN™ system. Many system objects are stored there, such as files, directories, programs, and TDOs.

The use of passive store is typically hidden by the services that use it. For example, the filing service and directory service handle all needed passive store operations for files and directories.

II-3.1 Concepts

II-3.1.1 Comparing Passive Store, Files, and Directories

All objects stored on disk are "in" passive store, including file objects and directory objects. However, an application that only uses files and directories can ignore most aspects of passive store and use files and directories in a familiar way. Even for an application that uses other object types, files and directories can be appropriate containers for much of the application's stored data. The application designer should consider these points:

- Files support byte stream I/O and record I/O. Several record-structured file organizations are supported and a rich set of file management, file indexing, and record access operations is provided.
- Files cannot contain ADs.
- Directories can contain ADs but have a fixed structure: a set of <name, AD> pairs, such that each name is unique within its directory.
- If the application wants to store its own typed objects on disk, objects that can contain ADs and have an arbitrary structure, then it must use passive store directly to store those objects.

II-3.1.2 Using Passive Store at Different Levels

Passive store can be used at three different levels:

1. At the conventional application level, an application can use files and directories. Passive store is transparent, but the application may still benefit from the distributed file system and the flexible protection model.
2. At the sophisticated application level, an application can:
    - Request that system objects, such as TDOs or authority lists, be stored.
    - Create its own network of generic objects and store it.
3. At the object-oriented application level, an application can define new object types and type manager modules for those types. Each type manager uses the `Passive_Store_Mgt` package to customize the passive store behavior of its object type.

`Passive_Store_Mgt` is directly used by the second- and third-level applications. Two different groups of calls are provided for the two different levels. The sophisticated application can use `Request_` calls, such as `Request_update`, which do not require any rights. However, an object's type manager can specify type-specific handling of such calls or refuse them by raising `System_Exceptions.operation_not_supported`. A type manager in an object-oriented application can use direct calls, such as `Update`, which require rep rights. A type manager only uses the direct calls for the objects that it manages. The two groups of calls appear in the same package because one module will often use both groups of calls: the direct calls for objects that it manages and the `Request_` calls for objects that other modules manage. Chapter VII-6 shows how to build a type manager for a stored object type.

II-3.1.3 Object Versions

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 II-3-2).

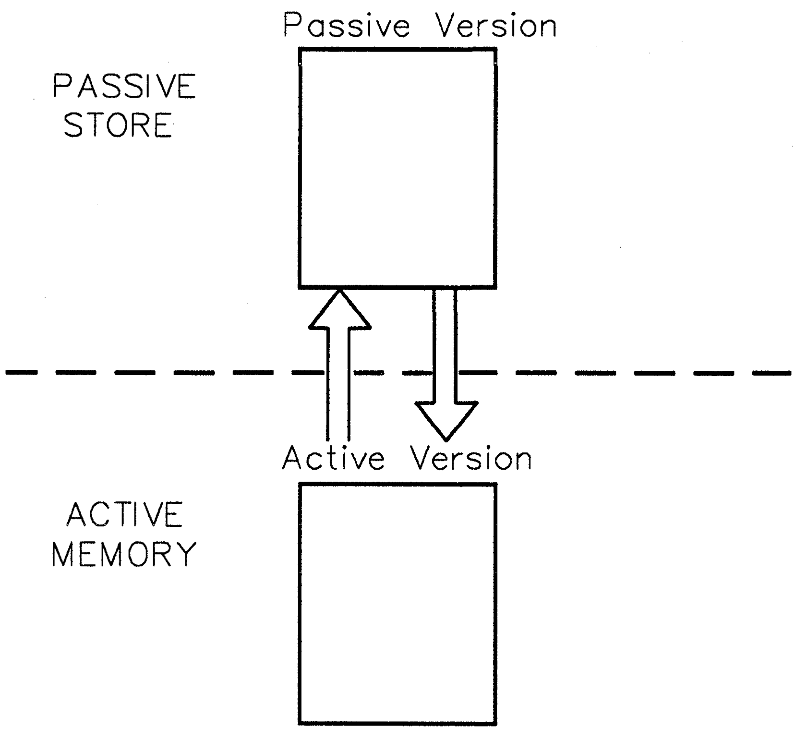


Figure II-3-2. A Single Object can have Passive and Active Versions.

An object can have multiple active versions, in use by different jobs or nodes, but can have only one passive version. Though the term "version" is used, passive store is not a "version control" or "revision control" system and cannot store or reconstruct any but the most current passive version. The passive store does ensure that out-of-date active versions cannot corrupt an object's passive version.

II-3.1.4 Object Activation

Only active versions can be directly read or written. Reading or writing a stored object with no active version causes *activation* of the object, creation of an active version that is then read or written.

Objects are activated when needed in the same way that 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 passive version.

An active version of an object can only be created from the object's passive version. There is no way to create an active version from another active version.

Local objects are activated on a per-job basis. If jobs A and B both use an object, then they get separate active versions. Activating a local object consumes storage in the job's local SRO (storage resource object).

Global objects are activated on a per-node basis. If jobs A and B both use an object and are on the same node, then they share an active version. If jobs A and B are on different nodes, then they use different active versions. A global object is activated in either the normal or frozen global SRO, depending on its memory type.

II-3.1.5 Activation as Reincarnation

A passive object can "live" multiple times, in the form of successive activations in different jobs. For example, job A may create and passivate an object, and then terminate. Some time later, job B references the object and a second active version is created, a second active "life" for the object. Whether an object remembers its "past lives" depends on whether or not each job passivates any changes that it makes.

II-3.1.6 AD Activation

Activating an object activates all ADs in the object. However, activating an object does not activate the objects that it references. For example, a program accesses object A, triggering A's activation. A contains an AD for object B. The AD for B is activated, but not B itself. B will subsequently be activated if the program accesses it.

AD activation is the point at which protection of passive objects is implemented. Chapter III-3 describes how passive objects are protected. When an AD is activated as part of object activation, then a null AD is activated in its place if the owner of the object that contains the AD is not allowed access to the object referenced by the AD.

ADs are frequently activated by being retrieved from directory entries. Protection of ADs in directory entries is handled somewhat differently, as described in Chapter III-3. If the caller is not allowed access to an AD in a directory entry, then `Directory_Mgt.no_access` is raised.

II-3.1.7 Object Passivation

An object's passive version is modified only when a program or type manager explicitly *passivates* or *updates* the object. Passivating an object copies a particular active version to the passive version.

These `Passive_Store_Mgt` calls create or update an object's passive version:

```
  Request_update
  Update
  Update_tree
  Update_with_alternate_rep
```

`Update` updates a single object, for which the caller has rep rights. `Update_tree` updates a tree of objects; the caller must have rep rights for the root object. `Request_update` is used to update type-managed objects from outside their type managers; it requests that the object's type manager update an object. A type manager can refuse an update request by raising `System_Exceptions.operation_not_supported`.

When an object is passivated from a particular active version, then all other active versions are marked as being obsolete. Any attempt to update an object from an obsolete version is rejected, with the `Passive_Store_Mgt.outdate_object_version` exception.

An object's passive version can have a different size than its active version. For example, passive versions that contain ADs are larger than the corresponding active versions.

II-3.1.8 Passivation Dependencies

A passive object should normally only refer to other passive objects. This rule includes implicit references, such as the AD that every object has for its TDO. Thus:

- If an object is passivated, its TDO should be passivated.
- If an object is passivated, then its attribute list should be passivated.
- If a TDO or an attribute list is passivated, then all attribute IDs and attribute value objects that it references should be passivated.
- If an object is passivated, then any authority list protecting the object should be passivated.
- If an object is passivated, then objects for which it contains ADs are normally passivated.

As a general rule, if *A* depends on (references) *B*, then *B* should live as long or longer than *A*. Note that this rule should also influence when destroying a passive version is allowed.

If your application attempts to access an object that has no active or passive version, then `System_Exceptions.object_has_no_representation` is raised.

II-3.1.9 Active-Only Objects

Some types of object cannot be passivated; such objects are called *active-only*. For example, objects that exist only during program execution are active-only: sessions, jobs, processes, stacks, and transactions.

Paradoxically, an active-only object type must have the passive store attribute! This is because the default behavior of objects is to be passivatable. The `Passive_Store_Mgt.Set_refuse_filters` call assigns fields within a type's passive store attribute record so that the type's objects are active-only.

II-3.1.10 Passive Store Behavior of OS Object Types

This section summarizes the passive store behavior of some common types of objects. The lists in this section are selective; see the "OS Object Types" appendix in the *BiiN™/OS Reference Manual* for a list of object types.

The following OS object types are kept in passive store by their type managers:

normal directories
files.

The following OS object types can be passivated, but the application must handle creating and updating passive versions:

attribute ID
attribute list
authority list
basic disk
basic streamer
data definition (DDef) object
domain
event cluster
generic object
instruction object
name space
pipe
static data object
type definition object (TDO).

The following OS object types are active-only and cannot be passivated:

job
all opened device types
certain system directories
process
session
stack
storage resource object (SRO)
transaction
windows (character or graphics).

The following OS object type cannot be passivated but is permanently stored in the Clearinghouse instead of in passive store:

ID.

II-3.1.11 Passive ADs

Like any other object, a stored object can be referenced by many ADs, located in active versions or in other stored objects. Before creating an object's passive version, you must store at least one AD to the object. The first passive AD for an object is the object's *master AD*. All other passive ADs for an object are *alias ADs*. In certain circumstances, described in Section II-3.1.11.6, an AD is converted from an alias AD to a master AD, but there is never more than one master AD.

II-3.1.11.1 Referencing Between Active Memory and Passive Store

ADs can freely cross the boundary between active memory and passive store. A passive version can contain an AD for an active-only object. An active-only object can contain an AD for a passivated object.

II-3.1.11.2 Master ADs

The master AD determines the stored object's:

- volume set
- owner
- authority list (if any).

Master ADs can be stored in directories or any other passive objects.

A stored object is always on the same volume set as the object containing its master AD. Because master ADs cannot reference across volume sets, any passive object can be reached by a chain of master ADs from its volume set's root directory. The volume set containing an object's passive version *cannot* be changed; however the volume set may be moved to another node or even another BiiN™ system. Also the backup service provides techniques to archive volume sets or collections of objects within volume sets and then restore them on other volume sets (but with new object identities).

A stored object's initial owner is the user ID for the process that stores the master AD.

A stored object's initial authority list is determined as described in Chapter III-3.

Authority list evaluation uses whatever rights are on the master AD, even when evaluating rights for an alias AD. Normally a master AD should have all type rights and no rep rights, as rep rights should only be granted within the type manager. For a passivated generic object, the master AD should normally have rep rights as well.

II-3.1.11.3 Alias ADs

All passive ADs besides the master AD are *aliases* or *alias ADs*. Aliases can be freely created and deleted without restriction. Aliases can reference objects on other volume sets or other nodes. An alias may even reference an object on a volume set that is not mounted in the system.

Alias ADs can be used for any object operations. While some operations require that an object have a master AD, no operation distinguishes between a master AD and alias when specifying an object to operate on.

II-3.1.11.4 Restrictions on Storing Master ADs

There are these restrictions on storing master ADs:

1. If the object will use authority list protection, then its authority list must be set *before* the master AD is stored.
2. A master AD cannot be stored after an AD for the object is stored in an active-only directory.
3. A master AD cannot be stored after an AD for the object is transmitted to another job via any of these techniques:
    - Job invocation parameter buffer
    - Event cluster signal
    - Remote procedure call
    - Datagram
    - Virtual circuit.
4. A master AD cannot be stored after an application tries to store a master AD within a transaction, and then aborts the transaction.

These restrictions will cause no problem if you create a passive object as follows:

1. Allocate the object's initial active version.
2. If needed, explicitly set the object's authority list.
3. Store the master AD.
4. Passivate the object.
5. If done within a transaction and the transaction aborts, recover all the way to the first step, allocating a *new* object if the code retries. This avoids the problem of being unable to store a master AD for the previously allocated object.

II-3.1.11.5 Master ADs and Passive Object Lifetimes

In active memory there is no concept of a "master" AD, so why is one AD for each object singled out in passive store? There are several reasons, and one of the best reasons is that this solves the problem of knowing when to delete passive versions. Active objects are reclaimed either by job termination (for local objects) or by garbage collection. Garbage collection can reclaim objects with indefinite lifetimes by detecting when those objects can never again be accessed by any chain of ADs. However garbage collection is impractical in passive store, because ADs can reference between nodes in a network and even reference between mounted and dismounted volume sets! An exhaustive scan of all filing volume sets at all nodes would consume too much network and disk bandwidth, but a scan that includes dismounted volume sets is completely impractical. Instead the master AD is singled out, and constrained to be on the same volume set as the object it references. If the master AD is deleted, then the object is deleted and reclaimed. The object can also be explicitly destroyed while the master AD still exists, but no further passive version of the object can be created.

A passive object exists until its master AD is deleted, or until the object's passive version is explicitly destroyed.

II-3.1.11.6 Transferring Mastership

In two cases, deleting a master AD does not destroy a passive object *O*, but instead converts an alias AD to become the new master AD:

- If the master AD is stored in a directory entry and other directory entries on the same volume set reference *O*, then "mastership" is transferred to one of the other entries.
- If the master AD is stored in a non-directory object and other ADs in that object reference *O*, then mastership is transferred to one of those other ADs.

Note that the master AD always remains on the same volume set as the object it references. Note also that mastership is never transferred from a directory entry to a non-directory object.

II-3.1.11.7 Object Trees

An *object tree* is defined by a stored *root object* and all objects reached from it via a chain of master ADs. For example, if A contains a master AD for B which contains a master AD for C, then the object tree rooted in A contains C.

All objects in an object tree have the same volume set.

Several passive store calls operate on object trees: `Copy`, `Destroy`, `Request_update` (in its default version), and `Update_tree`.

II-3.1.11.8 Passive ADs as Universal Identifiers

A passive AD contains enough information to uniquely identify a passive object among all the passive objects ever created on any BiiN™ system anywhere at any time. Passive objects created on different volume sets, nodes, or systems can never be confused. A passive AD references one object in a universal address space, an address space that spans all objects ever created on any BiiN™ system anywhere.

II-3.1.12 Passive Store Behavior of Generic Objects

Generic objects support all passive store calls. All `Request_` passive store calls on generic objects require read rep rights.

II-3.1.13 Passive Object Characteristics

Figure II-3-3 shows a passive object and some of its associated characteristics.

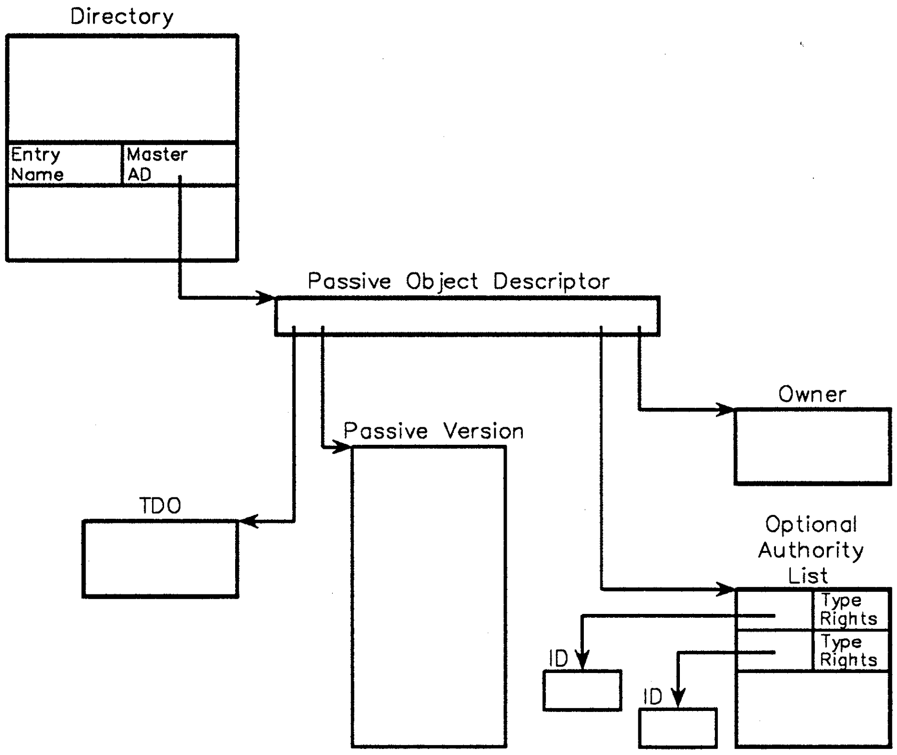


Figure II-3-3. A Stored Object

A passive object information record is also maintained for every passive object:

```
length
create_time
read_time
write_time
change_status_time
owner
auth_list
volume_set
node
```

II-3.1.14 The Life History of a Passivated Object

Figure II-3-4 shows a job creating and using a stored object:

1. Creating the active version
2. Storing the master AD
3. Storing the passive version
4. Changing the active version
5. Updating the passive version.

When the job terminates, the object's active version is deallocated but the passive version still exists.

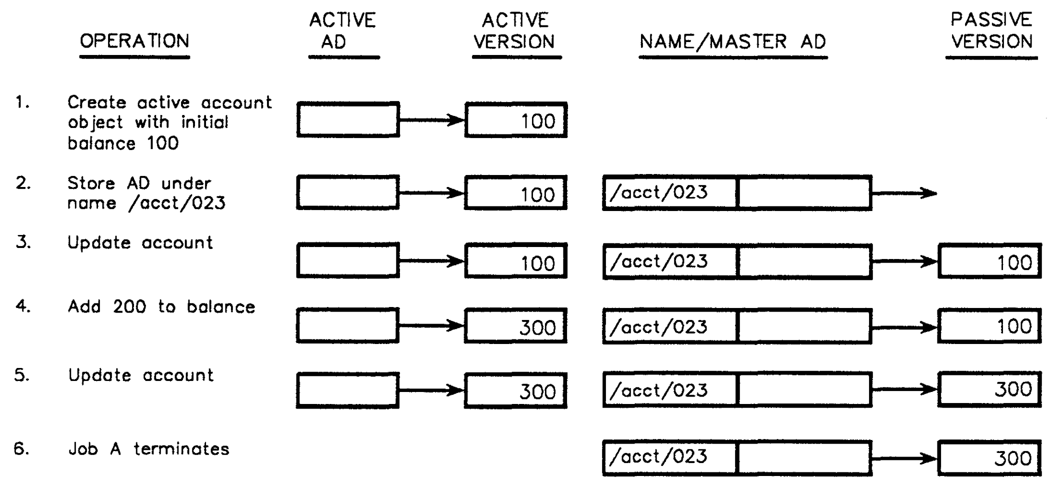


Figure II-3-4. Life History of a Stored Object Part I

Figure II-3-5 shows another job using and then destroying the same stored object:

1. Retrieving an AD from the directory entry
2. Accessing the object (transparently creates an active version)
3. Destroying the passive version, the directory entry, and the active version.

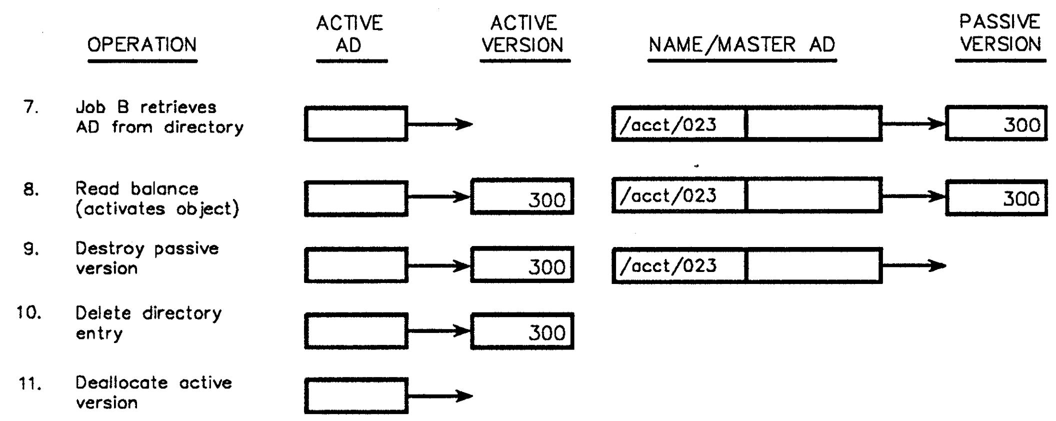

OPERATION
ACTIVE AD
ACTIVE VERSION
NAME/MASTER AD
PASSIVE VERSION
7. Job B retrieves AD from directory
8. Read balance (activates object)
9. Destroy passive version
10. Delete directory entry
11. Deallocate active version
/acct/023
300

Figure II-3-5. Life History of a Stored Object Part II

II-3.1.15 Activation Models

Passive store supports two models of object activation: the *multiple activation model* and the *single activation model.* The choice of an activation model can be concealed within an object's type manager, and only the type manager implementer needs to be concerned with the choice.

II-3.1.15.1 Multiple Activation

In the multiple activation model, a single stored object can have multiple active versions in different jobs (Figure II-3-6).

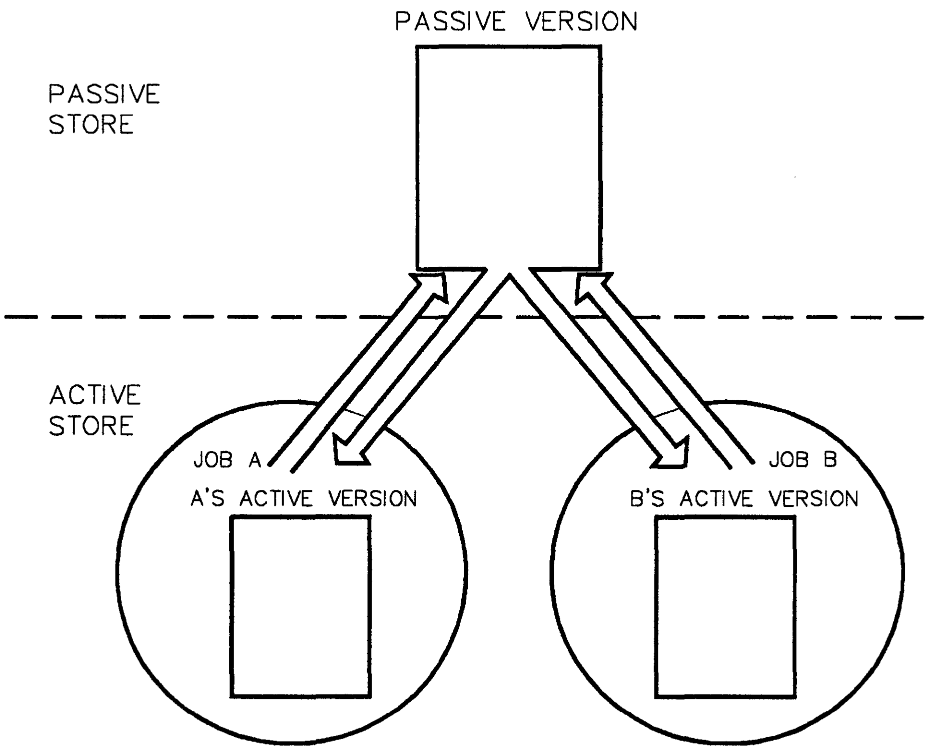


Figure II-3-6. A Single Object can have Multiple Active Versions.

In Figure II-3-6, if Job A updates the object, then Job B's active version is obsolete. Passive store keeps track of object versions and refuses updates from obsolete active versions, raising the `outdated_object_version` exception in `Passive_Store_Mgt`.

An application can handle the `outdated_active_version` exception by:

1. Calling `Reset_active_version` to make its active version current
2. Redoing whatever changes it made to the active version
3. Attempting the update again.

A type manager can also define a version-out-of-date flag in the type's objects. Passive store then sets the flag in any obsolete active versions of the type. An application can check the flag before using an active version, resetting the active version if needed. Note that using the flag does not eliminate the `outdated_object_version` exception; there can be communication delays in setting the version-out-of-date flags.

A job using a transaction can avoid any problems with an obsolete object version by *reserving* the object for the transaction. The object is reserved until the transaction is resolved. While a stored object is reserved, only updates associated with the reserving transaction are allowed. `Passive_Store_Mgt.Reserve` resets the caller's active version if it is obsolete, ensuring that subsequent code begins with the current version.

Reserving frequently accessed objects for long periods can cause performance problems by delaying other jobs.

For global objects, there is one active version of the object per node where the object is accessed rather than per job.

II-3.1.15.2 Single Activation

In the single activation model, an object is only activated in one *home job*. Other jobs that activate the object receive a token active version called a *homomorph* in place of the object. Those other jobs communicate with the object's home job to request operations on the real object.

The type manager conceals the use of homomorphs. When an application requests an operation on a homomorph, the type manager handles all the communication needed to perform the operation and return results.

A type manager can distinguish between homomorph and real active versions by defining an `is_homomorph` boolean that is true in the homomorph template and that is false in all real passive and active versions.

II-3.1.15.3 Choosing an Activation Model

In the single activation model, operations come to the object. In the multiple activation model, the object goes to the operations. In either case, the type manager's public interface should be the same. The choice of an activation model is an implementation decision, important only to designers of type managers, not to outside users.

The multiple activation model:

- Is often easier to implement
- Brings the object to the operation (good for repeated operations on smaller objects)
- Often requires code to handle clashes between concurrent and incompatible versions of the same object.

The single activation model:

- Is often more difficult to implement
- Brings the operation to the object (good for larger objects such as files)
- Does not cause clashes between multiple active versions of the same object.

II-3.1.16 Transaction Support

Most passive store calls that change passive versions are transaction-oriented. A transaction-oriented call participates in any default transaction. If there is no default transaction, then a transaction is created for the duration of the call.

Object activation and `Reset_active_version` calls normally can do *dirty reads*, reading a passive version written by another transaction before that transaction commits. An application can ensure that a committed version of an object is used by calling `Reserve` on the object.

An object can reserve an object using either a write lock or a read lock. Write locks are exclusive and are not released until the enclosing root transaction is resolved. Read locks can be shared with other read lockers, and can be released early with explicit `Release` calls. Update and destroy calls assert write locks if the affected objects are not already write-locked by the calling transaction.

Applications that use passive store may choose to maintain its transaction orientation. To do this, applications should group any sequence of logically dependent passive store/file/directory calls within a transaction. Chapter II-4 shows you how to use transactions.

Stored objects can be concurrently accessed by multiple jobs using multiple transactions. A passive store call on an object can be refused because it is being used by a transaction with a more recent timestamp than the transaction enclosing the refused operation. Chapter II-4 shows you how to handle timestamp conflicts.

II-3.1.17 The Passive Store Attribute

The behavior of passive store is customized for a particular object type by supplying a passive store attribute for the type. Each instance of this attribute is an object with the `Passive_Store_Mgt.PSM_attributes_object` Ada type.

II-3.1.18 Default Passive Store Behavior

If an object type does not have the passive store attribute, then by default it supports all passive store calls. The default allows copying of passive objects of the type and maps all `Request_` calls to the corresponding direct calls, regardless of what rights are on the AD supplied.

II-3.1.19 Type Manager Support

All the features described in this section are controlled by fields in a type's passive store attributes object.

A type manager can supply type-specific subprograms to be called in response to any or all of the following `Passive_Store_Mgt` calls:

```
Request_passive_object_info
Request_release
Request_reserve
Request_reset_active_version
Request_set_timestamps
Request_update
```

A type-specific subprogram that refuses a request should raise `System_Exceptions.operation_not_supported`.

A type-specific subprogram that makes multiple updates to passive objects, files, or directories can enclose its updates in a transaction, so that the type-specific subprogram is transaction-oriented.

A type manager can refuse to allow copying of the type's stored objects by setting the `copy_permitted` boolean to false in the PSM attributes object. For example, if an account balance represents an amount of money, then it may be wise to prohibit copying of account objects!

A type manager can define a *locking area* in its type's objects, using fields in the passive store attribute object. When an object is activated, a semaphore is created and an AD for the semaphore is placed in the first word of the locking area. Other words of the locking area are initially zero. When an active version is reset, the locking area is preserved and copied over from the preceding active version.

A locking area is needed when multiple processes in a job share an active version. The active version must contain a semaphore and possibly other information to synchronize access to the active version. Without a locking area, resetting the active version would lose the reference to needed synchronization information, such as a semaphore with a blocked process waiting to use the active version.

II-3.2 Techniques

After reading this section, you will be able to:

- Create a passive object
- Update a passive object
- Request an update
- Destroy a passive object
- Copy a stored object tree
- Get passive object information.

Many of the examples are excerpted from the non-transaction-oriented body of the `Stored_Account_Mgt_Ex` example. This package extends the type manager for accounts, developed in Chapter VII-3, to use passive store. Appendix X-A contains complete listings for `Stored_Account_Mgt_Ex` and other units excerpted in this chapter.

II-3.2.1 Creating a Passive Object

Calls Used:

`Directory_Mgt.Store`
Stores a new directory entry.

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

To create a stored object:

1. Create and initialize the active object (as described in Chapter VII-3).
2. Store a master AD for the object.
3. Update the object, creating the passive version.

The following excerpt from the `Stored_Account_Mgt_Ex` package body (non-transaction-oriented) shows how to create a stored object:

```
function Create_stored_account(
    starting_balance:
        Long_Integer_Defs.long_integer :=
        Long_Integer_Defs.zero;
    master:     System_Defs.text;
    authority:
        Authority_List_Mgt.authority_list_AD := null)
  return account_AD
  --
  -- Logic:
  --   1. Check the initial balance.
  --
  --   2. Allocate and initialize the account object.
  --
  --   3. Remove rep rights for the exported and master
  --      AD.
  --
  --   4. Store the master AD.
  --      Use "authority" as authority list to store the
  --      account. If "authority" is null, the default
  --      authority list of the target directory is used.
  --      If there is none the caller's authority list in
  --      the process globals is used.
  --
  --   5. Passivate the account object itself.
  --
  --   6. Return the AD without rep rights.
is
  account:          account_AD;
  account_untyped:  System.untyped_word;
  FOR account_untyped USE AT account'address;
    -- Account with no rep rights, viewed with
    -- either of two types.

  account_rep:      account_rep_AD;
  account_rep_untyped:  System.untyped_word;
  FOR account_rep_untyped USE AT
      account_rep'address;
    -- Account with rep rights, viewed with
    -- either of two types.

```

```
begin
  -- 1. Check the initial balance:
  --
  if starting_balance < Long_Integer_Defs.zero then
    RAISE insufficient_balance;

  else
    -- 2. Allocate and initialize the account object:
    --
    account_rep_untyped := Object_Mgt.Allocate(
        size => (account_rep_object'size + 31)/32,
        tdo  => account_TDO);
    account_rep.all := account_rep_object'(
        balance => starting_balance);

    -- 3. Remove rep rights for the exported and
    --    master AD:
    --
    account_untyped := Access_Mgt.Remove(
        AD     => account_rep_untyped,
        rights => Object_Mgt.read_write_rights);

    -- 4. Store the master AD:
    --
    Directory_Mgt.Store(
        name    => master,
        object  => account_untyped,
        aut     => authority);

    -- 5. Passivate the account object itself:
    --
    Passive_Store_Mgt.Update(account_rep_untyped);

    -- 6. Return the account AD with no rep rights:
    --
    RETURN account;

  end if;
end Create_stored_account;
```

II-3.2.2 Updating a Passive Object

Calls Used:

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

Any operation that changes a stored object normally updates it. For example, the `Change_balance` call in `Stored_Account_Mgt_Ex` updates the affected account.

```
account_rep.balance :=
    account_rep.balance + amount;
Passive_Store_Mgt.Update(account_untyped);
```

II-3.2.3 Requesting an Update

Calls Used:

`Passive_Store_Mgt.Request_update`
Requests a type-specific update. Defaults to `Update_tree`.

The following excerpt from the `Make_object_public_ex` example shows storing a master AD and then requesting an update for an authority list object:

```
aut_list:  constant
    Authority_List_Mgt.authority_list_AD :=
    Authority_List_Mgt.Create_authority(entries);
. . .
  Directory_Mgt.Store(aut_list_path, aut_untyped);
  Passive_Store_Mgt.Request_update(aut_untyped);
```

The excerpt uses `Request_update` instead of `Update` because the caller is not the type manager for authority lists and does not have rep rights.

II-3.2.4 Destroying a Stored Object

Calls Used:

`Directory_Mgt.Get_name`
Gets an object's full pathname (if any).

`Directory_Mgt.Delete`
Deletes a directory entry.

`Passive_Store_Mgt.Destroy`
Destroys a stored object tree.

The `Destroy_account` call in the `Stored_Account_Mgt_Ex` example destroys an account's passive version, deletes one directory entry for the account, and deallocates the account's active version. If additional directory entries reference the account, then those entries become dangling references. (Any attempt to access the object via such a dangling reference raises `System_Exceptions.object_has_no_representation`. Here is the code:

```
  path_length:  integer := 60;
    -- Initial text length for name assigned
    -- by "Directory_Mgt.Get_name".  If
    -- insufficient, then the value is
    -- increased and the operation is
    -- repeated.
begin
  account_untyped := Access_Mgt.Import(
      AD     => account_untyped,
      rights => destroy_rights,
      tdo    => account_TDO);

  if account_rep.balance /=
      Long_Integer_Defs.zero then
    RAISE balance_not_zero;

  else
    Passive_Store_Mgt.Destroy(account_untyped);

    loop
      declare
        path_text:  System_Defs.text(path_length);
      begin
        Directory_Mgt.Get_name(
            obj  => account_untyped,
            name => path_text);  -- out.
        if path_text.length >
           path_text.max_length then
          -- Text was lost.  Retry:
          path_length := path_text.length;
        else
          Directory_Mgt.Delete(path_text);
          EXIT;

        end if;
      exception
        when Directory_Mgt.no_name =>
          EXIT;

      end;
    end loop;

    Object_Mgt.Deallocate(account_untyped);
  end if;
end Destroy_account;
```

NOTE: If an application knows that an object has only a single directory entry on its home volume set, and that the directory entry contains the object's master AD, then destroying the object is simpler: Just delete the directory entry containing the master AD and the passive version is also destroyed.

II-3.2.5 Copying a Passive Object Tree

Calls Used:

`Directory_Mgt.Store`
Stores an AD in a new directory entry.

`Passive_Store_Mgt.Create_copy_stub`
Creates a copy stub, used as the target of a subsequent `Copy` call.

`Passive_Store_Mgt.Copy`
Copies a tree of stored objects.

Suppose that passive store contains a tree of objects that you want to copy, such as a program containing many modules; perhaps you want a copy to create a variation of the program. Objects in the tree, all connected to the root by master ADs, are all the parts of the program. The program also contains alias ADs that reference system services or shared library routines.

When you make a copy of the program, you want to preserve the program's structure in your copy. For example, if object A in the program has a master AD for object B, then you want your copy of object A to contain a master AD for your copy of object B. Thus, copying must not just copy stored objects but sometimes *remap* ADs in the objects. On the other hand, if object B contains an alias AD for an object D that is not in the tree, then the copy of object B should contain an identical AD. This is the case when the program and the copy reference shared services or libraries that are not also copied. Figure II-3-7 shows how master ADs (A to B which maps into E to F) are remapped and alias ADs to objects outside the tree (B and F to D) are unchanged when a tree is copied. Any AD from an object in the tree that references another object in the tree is remapped, even if it is an alias (C to B which maps into G to F).

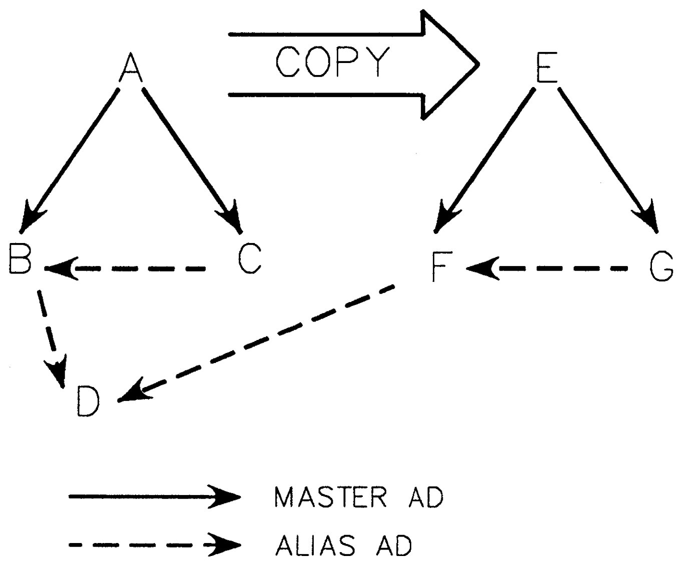


Figure II-3-7. Copying an Object Tree

The passive store copy calls should *not* be used for backing up or restoring stored objects. See the *BiiN™ Systems Administrator's Guide* for information about backing up or restoring stored objects.

Some object types cannot be copied. For example, TDOs and attribute IDs cannot be copied. Objects that correspond to physical devices cannot be copied. (What would it mean to "copy" a printer?) The `copy_permitted` boolean in a type's PSM attributes object determines whether objects of the type can be copied. The `Set_refuse_filters` call assigns `copy_permitted` false. If any objects in a tree cannot be copied, then `Copy` raises `System_Exceptions.operation_not_supported`.

Copying an object tree copies passive versions and does not create active versions of any of the copied objects.

When you make a copy, you create a *new* stored root object and possibly other new objects below it in a tree. But before this root object that does not yet exist can be stored, a master AD must be stored for it! The master AD must be stored to determine the new object tree's volume set, owner, and authority lists. So that a destination master AD can be stored before copying an object tree, `Passive_Store_Mgt` provides the `Create_copy_stub` call, which creates a new "stub" object that is only used to:

1. Store a master AD.
2. Be the destination of a `Copy` operation.

Copying a tree of stored objects thus has three steps:

1. Create a *copy stub*, used as the target of the `Copy` call.
2. Store a master AD for the stub.
3. Copy the object tree to the stub.

The `Named_copy_ex` example procedure copies a source object tree to a destination object tree, given `source` and `dest` pathnames. This excerpt shows the three-step operation:

```
source_AD := Directory_Mgt.Retrieve(source);
dest_AD   := Passive_Store_Mgt.
                 Create_copy_stub(source_AD);
Directory_Mgt.Store(name   => dest,
                    object => dest_AD);
Passive_Store_Mgt.Copy(source_AD, dest_AD);
```

II-3.2.6 Getting Passive Object Information

Calls Used:

`Passive_Store_Mgt.Request_passive_object_info`
Requests information about an object's passive version.

The OS keeps much more information for passive versions than active versions: owner, authority list, volume set, node, size, and time created, last read, last written, and last modified in any way.

The `Older_than_ex` example function compares two stored objects to determine if the first was last written before the second. For example, `Older_than_ex` can be used to determine if a machine instruction object is older than the associated source code object, requiring a recompile.

The function uses `Request_passive_object_info` rather than `Get_passive_object_info` because it may not have rep rights on the objects being checked:

```
  use Long_Integer_Defs;
    -- Import "<" for long integers.

  a_info:  Passive_Store_Mgt.passive_object_info;
  b_info:  Passive_Store_Mgt.passive_object_info;
begin
  a_info := Passive_Store_Mgt.
            Request_passive_object_info(a);
  b_info := Passive_Store_Mgt.
            Request_passive_object_info(b);

  if not a_info.valid or else not b_info.valid then
    RAISE System_Exceptions.bad_parameter;

  else
    RETURN a_info.write_time < b_info.write_time;

  end if;
end Older_than_ex;
```

The `valid` field of the `passive_object_info` record is false if the object does not have a passive version.

II-3.3 Summary

- Passive store is a distributed, reliable object filing service.
- Use files instead of passive store if you do not need to store ADs and object types. Use files to port programs that use conventional file systems. Use files if you need fast random access to record-structured data.
- An object can have zero or one passive versions and zero, one, or multiple active versions. These are all versions of a single object.
- Active versions are created automatically when you try to access an object's representation within a job without an active version of the object.
- Passive versions are created and changed by explicit *update* calls.
- The first AD stored for an object is its *master AD*, and must be stored before the object is stored.
- An *object tree* is a root object and all objects reached from it via a chain of master ADs. Some passive store calls operate on object trees.
- The passive store service detects conflicts between multiple object versions and raises exceptions that can be handled by the callers.
- Passive store calls are normally handled within type managers.
- Type managers can customize the passive store service for their type's objects by defining an instance of the passive store attribute.

STARTING AND RESOLVING TRANSACTIONS 4

Contents

A *transaction* groups related operations so that either all the operations succeed or all are rolled back.

Packages Used:

`Transaction_Mgt`	Provides *transactions* used to group a series of related changes to objects so that either all the changes succeed or all are rolled back.

This chapter introduces transactions and basic techniques for using transactions. Chapter IV-10 describes transaction locking of files, opened files, key ranges, and records.

Transactions are typically used to group changes to files or other objects stored on disk. For example, transferring $100 from one bank account to another could be enclosed in a transaction. If the change to either account failed or if the system crashed before all changes were made, then all changes would be undone.

Transactions are also useful in less obvious ways. Even when a single record is inserted into a file, several disk writes may be needed to update indexes as well as the file's primary data area. The filing service uses a transaction to ensure that a failure within such a group of writes doesn't make the file and its indexes inconsistent.

II-4.1 Concepts

II-4.1.1 What Transactions Provide

Transactions provide several services simultaneously to the developer:

Atomicity	Transactions are *atomic* or indivisible, either completely succeeding or making no changes at all. (Though atomicity only applies to those operations that participate in the transaction, as described below.)
Consistency	Inconsistent and transitory states that your data passes through within a transaction are not visible from outside the transaction. For example, the state when one account has changed but not the other, or the state when one index has changed but not another, are not visible outside the enclosing transactions. (Consistency is enforced by type managers.)
Crash Recovery	Transactions work correctly even if the system hardware or operating system crashes. Transactions in progress when a crash occurs are aborted as part of crash recovery.
Synchronization	Transactions synchronize with each other so that one transaction cannot access data being actively used by another transaction.
Deadlock avoidance	Transaction synchronization is designed so that *deadlocks* are not possible. An example of a deadlock would be if two transactions each blocked waiting for locks held by the other transaction. However, because each is blocked, the locks would never be released. The transaction service defines an ordering scheme that determines whether a particular transaction is allowed to block for another particular transaction. If blocking is not allowed, an exception is raised.

Time limits	When an application creates a transaction, it can specify a *timeout* that limits (aproximately) the total time taken by the transaction.
Distributed service	The transaction service is distributed. One transaction can include changes to objects at multiple nodes.
Extensible service	By default the transaction service can be used with any new type of local passivatable object. A type manager can also customize the transaction service for its object type.

II-4.1.2 Transaction Calls

`Transaction_Mgt` defines three basic calls for transactions:

`Start_transaction`
: Creates a transaction.

`Commit_transaction`
: All changes within a transaction are done successfully. Terminate the transaction.

`Abort_transaction`
: Something went wrong. Undo changes made within the transaction and terminate the transaction.

`Transaction_Mgt` includes several other calls, used for special purposes. The three basic calls are all that many applications need.

II-4.1.3 Transaction Stack

Transactions can be nested, and several transactions may be active at once. Each process has an associated *transaction stack* in its process globals. A process's transaction stack is initially empty. Creating a transaction pushes the new transaction onto the caller's transaction stack.

II-4.1.4 The Default Transaction

The transaction on top of a process's transaction stack is its *default transaction*. Many `Transaction_Mgt` calls have a transaction parameter that can be defaulted, indicating that the caller's default transaction should be used. `Start_transaction` pushes the new transaction onto the caller's transaction stack. By default, `Commit_transaction` and `Abort_transaction` operate on the default transaction and pop it from the stack.

II-4.1.5 Participating in Transactions

A transaction only affects those calls that *participate* in the transaction. A participating call must be implemented in such a way that it can be aborted up to the time it is committed. Some calls only participate in a transaction if the caller has a default transaction. Other calls participate in a transaction even if the caller has none, by creating a transaction for the duration of the call. If a System Services call participates in transactions, then its call description in the *BiiN™/OS Reference Manual* will state that it participates. Calls that affect structured transaction-oriented files, directories, and other passive objects often participate in transactions.

It is important to realize that aborting a transaction does *not* roll back non-transaction-oriented actions! For example, screen I/O, printing a check, or assigning a program variable are not rolled back.

II-4.1.6 The Transaction Service as a Coordinator

The transaction service acts as a coordinator for whatever type managers participate in a particular transaction. Different transaction-oriented type managers may implement transaction locking, commital, and abortion differently.

II-4.1.7 Subtransactions

If `Start_transaction` is called when there is already a default transaction, then the new transaction is a *subtransaction* or *child transaction* of that default transaction. A top-level transaction is a *root transaction.* Subtransactions and root transactions behave somewhat differently:

- Committing a subtransaction does not make changes permanent but simply passes responsibility for the changes up to its parent transaction.
- Committing a root transaction makes permanent any changes made within the root transaction and within any committed subtransactions.
- A transaction cannot be committed until all of its subtransactions are committed or aborted.
- Aborting a subtransaction only aborts changes within the subtransaction and does not abort the parent transaction.
- Aborting a transaction also aborts its subtransactions.

Why use subtransactions? As far as atomicity and rollback, there seems to be no advantage to having transactions within transactions. There are two good reasons that subtransactions are used:

1. To allow transaction-oriented functions to be combined in straightforward ways. A can call B which can call C, each procedure can enclose its code in a transaction, and they will all work.
2. To provide synchronization between concurrent processes within the same transaction. The transaction service uses transactions as the units being synchronized. Any concurrency within an overall transaction must be split into different subtransactions or the needed locking won't happen. (To be precise, subtransactions for synchronization of concurrent processes in a transaction are only needed if there is data that may be used and locked by more than one of the processes.)

II-4.1.8 Avoiding Subtransactions

There is some overhead in creating subtransactions when they are not required. A transaction-oriented module may choose to check whether there is already a default transaction and only start a new transaction if there is not already one on the stack.

II-4.1.9 Rules for Using Transactions

These rules can help you in designing code that uses transactions:

1. Normally the section of code that starts a transaction should also commit it if successful and abort it if any exceptions occur.
2. If possible, code between matching `Start_transaction` and `Commit_transaction` calls should not include operations that don't participate in the transaction.

3. If possible, code between matching `Start_transaction` and `Commit_transaction` calls should not include transfers of control out of that code block, such as `RETURN`, `EXIT`, or `GOTO`. If there are such transfers, then the transaction should be either committed or aborted on every possible path out of the block.
4. Your code should normally not manipulate the transaction stack directly.
5. If you spawn child processes within a transaction *T* and those processes need to participate in *T*, then:
   a. *T* must be passed to each child.
   b. Each child must push *T* on its transaction stack and then start a subtransaction.
   c. Each child must resolve its subtransaction before terminating.
   d. Each child should signal resolution of the subtransaction to the parent process.
   e. The parent process should not attempt to commit *T* until it is signaled that all the subtransactions used by the child processes have been resolved.

II-4.1.10 Transaction Locking

Whenever a transaction-oriented operation reads or writes data, it *locks* the data or some entity containing the data. Locking prevents concurrent changes and also keeps changes within the transaction from being visible to other transactions until such changes are committed. There are two type of locks asserted by a transaction:

read lock	Indicates that a transaction is using an entity and that the entity cannot be changed until the lock is released. A read lock allows other read locks but excludes write locks. A read lock can be explicitly released from within the transaction that asserted the lock.
write lock	Indicates that a transaction is using an entity and may change it. The entity cannot be read or written from outside the locking transaction until the locking transaction commits or aborts (except for *dirty reads*). A write lock is exclusive and does not allow any other concurrent locks. A write lock cannot be explicitly released. Only resolving the locking transaction releases a write lock.

The granularity of locking depends on the application or type manager. Transaction locks can be aquired on entire files, records within files, entries within directories, or entire passive objects.

For reading only, an application can choose to bypass locks and *dirty read* data that may be involved in an uncommitted transaction.

II-4.1.11 Transaction Timeouts

Whenever a transaction is specified, an advisory *timeout* duration is specified. By default a system-supplied value, specified as a node configuration parameter, is used. A transaction that runs out of time is automatically aborted by the transaction service. A timeout is a lower limit on the actual time allowed. For example, if a timeout of 30 seconds is requested, then the actual timeout may occur after one minute. A timeout value is always finite--there is no concept of waiting forever--but can be very large. Timeouts help ensure that files, records, or other data structures don't remain locked indefinitely if a process holding a transaction lock is killed.

II-4.1.12 Transactions and Job Termination

When a transaction is started, it is associated with the calling job. If that job terminates and the transaction is not already committed or aborted, then job termination aborts the transaction.

II-4.1.13 Avoiding Deadlock with Timestamp Conflicts

To avoid circular waiting, the transaction service defines a precedence scheme for transactions. Younger transactions will wait for older transactions but not the reverse. If an older transaction does request a lock held by a younger transaction, then `System_Exceptions.transaction_timestamp_conflict` is raised. The rules are different if the transactions involved are ancestor and descendant; see `Transaction_Mgt.Blocking_permitted` for details.

An application can recover from a timestamp conflict by:

- Aborting its transaction
- Resetting any other state information, such as variables
- Looping back in its code to the point where the transaction is started.

The newly started transaction will be younger than the transaction holding the lock and will be allowed to wait. Multiple loop backs can occur due to (rare) concurrent activity.

II-4.1.14 Independent Transactions

`Transaction_Mgt.Start_independent_transaction` can be called to create a new root transaction even if the caller has a default transaction. Consider a system accounting manager as an example of using independent transactions. Operations that consume or return system resources would update accounts on disk via the system accounting manager. Such updates are independent of whatever the application may be doing, and should be independent of any surrounding transaction. Otherwise, an abort of the surrounding transaction could erase all charges for resources used during the aborted operation.

A process using an independent transaction should not try to get any lock held by an older unresolved transaction in the same process. This will cause the process to block until the older unresolved transaction times out.

II-4.2 Techniques

After reading this section, you will be able to:

- Use a transaction
- Avoid unnecessary subtransactions
- Use a transaction and recover from timestamp conflicts.

II-4.2.1 Using a Transaction

Calls Used:

`Transaction_Mgt.Start_transaction`
Creates a transaction.

`Transaction_Mgt.Commit_transaction`
Indicates that changes within a transaction are done. Makes the changes permanent if the transaction is a root transaction.

`Transaction_Mgt.Abort_transaction`
Undoes all changes made within a transaction.

The following excerpt from the `Make_object_public_ex` example shows how to use a simple transaction:

```
Transaction_Mgt.Start_transaction;
begin
  Directory_Mgt.Store(aut_list_path, aut_untyped);
  Passive_Store_Mgt.Request_update(aut_untyped);
  Transaction_Mgt.Commit_transaction;
exception
  when others =>
    Transaction_Mgt.Abort_transaction;
    RAISE;

end;
```

Note that the block containing the exception handler is only entered if the transaction is successfully started. Any exception causes the transaction to be aborted and the exception to be reraised.

II-4.2.2 Avoiding Unnecessary Subtransactions

Calls Used:

`Transaction_Mgt.Start_transaction`
Creates a transaction.

`Transaction_Mgt.Commit_transaction`
Indicates that changes within a transaction are done. Makes the changes permanent if the transaction is a root transaction.

`Transaction_Mgt.Abort_transaction`
Undoes all changes made within a transaction.

The following excerpt from the `Stored_Account_Mgt_Ex` transaction-oriented body shows how to start and resolve a local transaction only if the caller does not already have a default transaction:

```
      trans:  boolean := false;
        -- True if a local transaction is started.
 . . .
        -- 4. Start a local transaction if there is not
        --    a transaction on the stack:
        --
        if Transaction_Mgt.Get_default_transaction =
            null then
          Transaction_Mgt.Start_transaction;
          trans := true;
        end if;
        begin
 . . .
          -- 7. Commit any local transaction:
          --
          if trans then
            Transaction_Mgt.Commit_transaction;
          end if;
        exception
          -- 8. If any exception occurs, abort any local
          --    transaction, deallocate the account,
          --    and reraise the exception:
          --
          when others =>
            if trans then
              Transaction_Mgt.Abort_transaction;
            end if;
            Object_Mgt.Deallocate(account_untyped);
            RAISE;

        end;
```

Note the use of the `trans` boolean to indicate whether or not a local transaction has been started.

II-4.2.3 Using a Transaction and Recovering from Timestamp Conflicts

Calls Used:

`Transaction_Mgt.Start_transaction`
Creates a transaction.

`Transaction_Mgt.Commit_transaction`
Indicates that changes within a transaction are done. Makes the changes permanent if the transaction is a root transaction.

`Transaction_Mgt.Abort_transaction`
Undoes all changes made within a transaction.

The `System_Exceptions.timestamp_conflict` exception is raised to prevent transaction deadlocks, commonly when an older transaction requests an entity locked by a younger transaction. An application can recover from a timestamp conflict by aborting its transaction, resetting any other state information (such as variables), and looping back in its code to where the transaction is started. The newly started transaction will be younger than the transaction holding the lock and will be allowed to wait. Note that multiple loop backs can occur due to (rare) concurrent activity. The following example of using a transaction and handling timestamp conflicts is excerpted from the `Stored_Account_Mgt_Ex` example's transaction-oriented body:

```
  trans:  boolean := false;
    -- True if a local transaction is started.
begin
  account_untyped := Access_Mgt.Import(
          AD     => account_untyped,
          rights => change_rights,
          tdo    => account_TDO);

  loop
    if Transaction_Mgt.Get_default_transaction =
        null then
      Transaction_Mgt.Start_transaction;
      trans := true;
    end if;
    begin
      Passive_Store_Mgt.Reserve(account_untyped);
      if account_rep.balance + amount < zero then
        RAISE insufficient_balance;

      else
        account_rep.balance :=
            account_rep.balance + amount;
        Passive_Store_Mgt.Update(account_untyped);
        if trans then
          Transaction_Mgt.Commit_transaction;
        end if;
        RETURN account_rep.balance;

      end if;
    exception
      when System_Exceptions.
           transaction_timestamp_conflict =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        else
          RAISE;

        end if;
      when others =>
        if trans then
          Transaction_Mgt.Abort_transaction;
        end if;
        RAISE;
    end;
  end loop;
end Change_balance;
```

II-4.3 Summary

- A transaction groups related operations so that either all succeed or all are rolled back.
- Using a transaction can be done with three simple calls that all use the caller's default transaction, with no explicit parameters.
- Transactions synchronize with each other using read locks and write locks.
- Younger transactions will wait for older transactions at a lock, but not vice versa.
- Transactions can have timeout values.

WRITING MESSAGES 5

Contents

Messages, incidents and exceptions are used to pass error messages between applications, programs, program modules, and users. This chapter discusses messages, incidents and exceptions from a procedural viewpoint.

Packages Used:

`History_Services`	Contains calls for using a job's history log files.
`Incident_Defs`	Defines incident and message types.
`Message_Services`	Provides calls to write messages from message files, message stacks, or message blocks.
`Message_Stack_Mgt`	Manages a process's message stack.
`System_Error_Recording`	Provides calls to record errors in a *system log file.*

Traditionally, a program developer defines errors and exceptional situations and handles the messages that need to be sent when such situations occur. The BiiN™ system offers an efficient and powerful mechanism for reporting errors and sending messages using *incidents* and *messages. Help* messages are similar to error messages but are managed separately. See `Help_Text_Adm` in the *BiiN™/OS Reference Manual.*

An incident can be a normal program error, an Ada exception, an OS error, a test point as defined by a test point monitor, a situation that requires a message to the user or any situation that is reported outside the program. Each incident is assigned an *incident code* which identifies a message and a severity level for the incident. Figure II-5-1 shows how an application developer can associate an error with an appropriate message using incident codes.

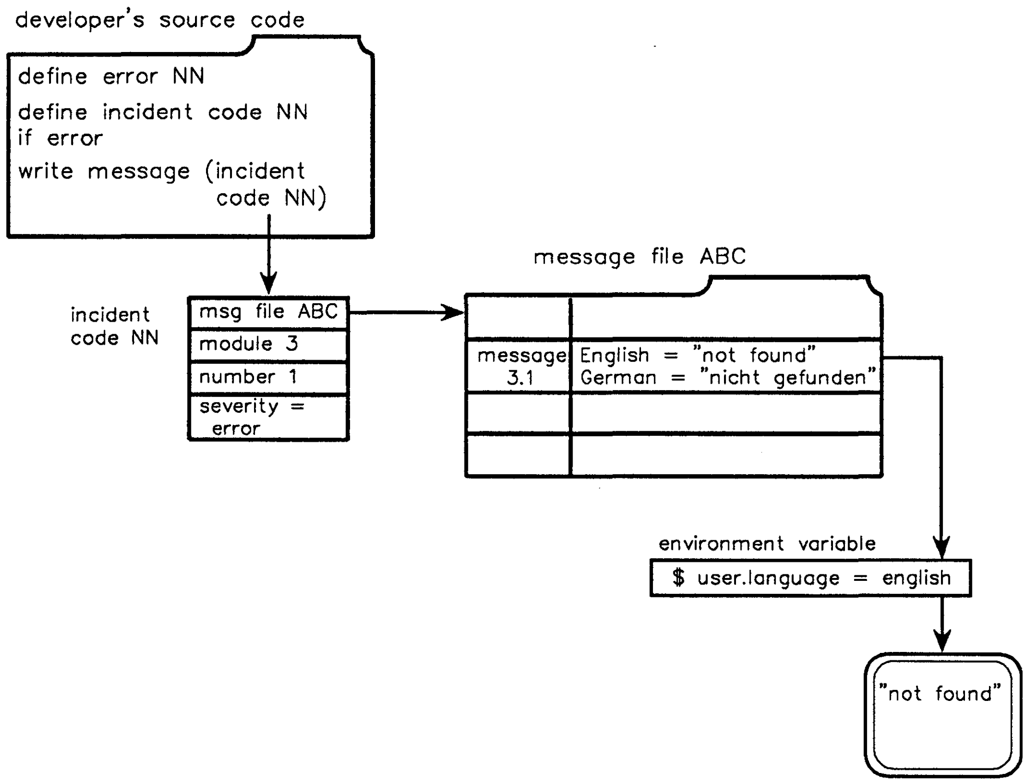


Figure II-5-1. Incidents Associate Errors with Messages

II-5.1 Concepts

Using incidents greatly eases the development of messages and the handling of errors for large projects with coherent user interfaces. For example, once an incident has been defined for the common situation `file not found`, all developers on a large project can reference that incident code when the situation comes up and only the definer needs to maintain the actual message text and severity level for that incident.

II-5.1.1 Messages

A *message* is the human-readable text associated with an incident. The message may contain text in more than one natural language (German, English, etc.) in a *short* or *long* form and contain parameters that can be substituted at the time the message is displayed.

The `Message_Services` package provides the procedural interface for sending messages. The developer indicates specific message by passing an incident code, message block or complete message stack (see subsequent sections for descriptions of these message constructs). The human language and the message level are determined by the user's setting of CL variables.

From the user's point of view, a message contains a *header*, generated by the system, and the message *text*, derived from the possible texts in the message for that incident code according to the user's CL variables.

A header is automatically prepended to messages of `warning`, `error` and `fatal` status but not to messages of `information` status. The heading consists of the time of occurrence, the sender name and a single-letter code for the severity level. The appearance of a header and message is affected by CL variables (defined in a subsequent section).

In the following message example, the time of the incident is `14:30:25`, the sender is `Inventory_Files` and the severity level is `E` (error). The CL variable `msg.time` is set to true, `message.long` is false and `user.language` is `English`.

```
   14:30:25 Inventory_Files - E:  Insufficient access rights to read file.
```

For more information about the contents of a message, see the *BiiN™ Command and Message Guide.*

II-5.1.2 Message Files

A *message file* contains the short and long forms of messages in one or more language variations. The messages are indexed by a message index comprised of a module number and sequential number and, optionally, by a message name. A message file is created for one or more applications by the `manage.messages` runtime command of the `manage.program` utility. The `message_object` field in the incident code references the application's message file.

For more information about creating a message file, see the *BiiN™ Command and Message Guide.*

II-5.1.3 Incident Codes

An *incident* is a BiiN™ construct that assigns a unique identifier, an *incident code*, to each error situation. An incident code references a message file, an individual message within that file and a severity level.

It is recommended that each Ada exception be assigned an incident code. Situations that are not Ada exceptions may also have incident codes. For example, an incident code can be assigned when a user presses a special function key.

An incident code record contains the following fields:

`message_object`

This field references the message file containing the message texts. The message file itself is created with the `manage.messages` utility. The software developer decides how to group messages (in a single file or in multiple files) and how to associate the message file with the software (to explicitly name the message file or to use a default message file). This field takes one of three values:

- A valid AD to a message file.
- A null AD indicating that the message file is the *default message file.* The default message file is created using `manage.messages` and is associated with the program via the OEO (Outside Environment Object, see the *BiiN™ Command Language Executive Guide*) using `store.default_message_file`. (See the *BiiN™ Command and Message Guide* for more information on these utilities.)

- A compiler-generated value (if the programmer did not define an incident code). This value may be an AD to some object other than a message file or a non-AD value. Either shows that a message is not defined for the incident code. In a program where the programmer does not define messages for program exceptions, for example the compiler generates unique exception values.

`module`	A number from 0 to 256K-1, inclusive, assigned to a program or module within a program. Combined with the incident `number` to identify an individual message within a message file. The incident `module` and incident `number` provide an index into the message file.
`number`	A number from 0 to 4095, inclusive, assigned to the incident within the module where it is defined.
`severity`	A level of seriousness for the incident. Four severity levels are recognized: `information`, `warning`, `error` and `fatal` error.

- `information`
  Not related to an error or warning; provides additional or helpful information.
- `warning`
  Indicates an occurrence which deviates from the expected behavior but does not impede the expected outcome of an operation.
- `error`
  Indicates that the operation generating the error cannot complete properly until the condition causing the error is corrected.
- `fatal_error`
  Indicates an error of such severity that further processing is not possible.

II-5.1.4 Message Blocks

A *message block* contains an incident code (which includes the message's module and number) and any message parameters. Calls that accept separate incident code and message parameters (see `Message_Stack_Mgt`) reformat them into message blocks.

II-5.1.5 Message Stacks

Program errors or incidents are frequently propagated through many layers of operations, especially within a large application. The message stack provides a means of keeping a trace of any incidents that have occurred within a process along with specific information about each incident.

The *message stack* is a fixed-length, open-bottomed stack. Each process has its own message stack. The message stack is large enough for two messages of maximum size. The message stack is open-bottomed so that if another message is pushed on a full stack, the bottom message on the stack is lost.

A message stack contains message block entries. Calls that accept separate incident code and message parameters (see `Message_Stack_Mgt`) reformat them into a message block which is then pushed onto a message stack. Entries are retrieved from the message stack as message blocks.

A message block is pushed onto a message stack whenever more specific information than that associated with the exception itself would be useful to someone debugging the program. (Messages are not automatically pushed on the message stack; they must be explicitly pushed on the stack by the exception handler.) Large user-written applications may make use of the message stack in a similar manner.

When a process terminates due to an unhandled error propagating out of its top level procedure, its message stack contains the history of that error's propagation. The first message block on the message stack is the actual error that caused termination. The subsequent message blocks contain information about the various levels of the system through which the error propagated.

It is good practice, although not required, for a program that catches and handles an incident or error to clear the message stack. Otherwise, on a later incident or error, the message stack contains the history of the previous (already-handled) errors as well as the error that caused termination. This can be confusing to someone debugging the program.

See the `Message_Stack_Mgt` package for more information about message stacks.

II-5.1.6 Messages and Exceptions

An *exception* is the Ada construct that signals the occurrence of errors or other exceptional situations that arise during program execution. *Raising* an exception causes normal program execution to be abandoned in order to deal with the error or situation.

Each exception may be, but is not required to be, associated with an *incident*. When a programmer wants to define a message for an exception, an incident code is assigned to that exception.

II-5.1.7 CL Variables That Affect Messages

The following CL variables affect how a message is displayed.

`user.verbose`	Determines whether *information* level messages are displayed. Messages which report job status, for example, may be displayed only in verbose mode. The developer can set `verbose_only` to true in order to make informational messages display when calling `Message_Services.Write_msg` or similar calls. When `user.verbose` is false, only `warning`, `error` and `fatal_error` messagess are displayed.
`user.language`	Controls which language variant of a message is displayed. If the message does not exist in the desired language, the message's default language variant is displayed (the first variant stored in the message file).
`msg.long_text`	Controls which form of the message (short or long) is displayed. If false (the default), the short form is displayed, otherwise the long form. If the selected level does not exist, the other level is displayed.
`msg.time`	Controls whether the time the incident occurred should be displayed as part of the message header.

The built-in command `set.variable` sets or changes the values of user CL variables.

II-5.1.8 How CLEX Handles Messages From Terminated Jobs

An exception for which no exception handler exists will terminate a job. All messages pushed on the message stack (up to the maximum it can hold) prior to termination will be on the stack. A message, if any, associated with the terminating exception will not appear on the stack.

II-5.1.9 Message Utilities

A message file can be created and updated with the `manage.messages` runtime command of the `manage.program` utility. The runtime commands of `manage.messages` include `change`, `list`, `remove`, `set.language` and `store`.

II-5.1.10 History Files

`Message_Services` automatically records messages in a job history log if one is installed. Users can turn off message recording via a boolean parameter in the various `Write_msg` calls. The `History_Services.Record_message` call takes an incident code and sends the corresponding message to the job's `history_log` file. Thus a job can maintain a record of any messages that were sent during the course of the job.

A user can have a history installed for a logon session if the CL variable `logon.install_history_log` is true.

A job can have a history installed if:

- The control option `::history_log` was called in the invocation of the job, or
- The built-in command `start.history_log` was called, or
- The package `History_Services` was used to create, open and set a `history_log` file.

II-5.2 System Error Log

The `System_Error_Recording` package provides calls for recording system errors on a system error log. This log is a record-oriented, sequential file. The error information can be specified as an incident code with from zero to five parameters or as a message block. The record layout is defined by the type `Monitor_Defs.monitor_message`. `System_Error_Recording.Get_event_cluster` provides access to an event cluster that gets signalled whenever an error is recorded to the system error log file. The system error log is only for trusted type managers such as device drivers.

II-5.3 Techniques

After reading this section, you will be able to:

- Define application messages
- Write a message
- Associate an incident code with an exception
- Replace OS exceptions with application messages

- Use predefined OS messages
- Push a message when raising an exception
- Clear the message stack when handling an exception
- Write a message with acknowledgement
- Record history entries.

Code examples in the following sections are excerpts from the `At_Cmd_Ex`, `At_Support_Ex`, `Inventory_file`, `Create_Name_Space_Cmd_Ex`, `Example_Messages` and `Inventory_Files` example programs that are listed in their entirety in Appendix X-A.

II-5.3.1 Defining Application Messages

Declarations Used:

`Incident_Defs.incident_code`
A representation for errors, warnings, information, exceptions and system errors.

The system recognizes four types of messages:

- Those used to identify exceptions
- Those used to identify other messages to be pushed onto a process's message stack
- Those used to identify operating system errors, and
- Those used as test point monitoring codes.

All of these message types may be represented by `Incident_Defs.incident_code`. This incident code contains the `severity` of the incident and a message file reference and index (module and number) which uniquely identifies message text associated with the incident.

To create an incident code, declare a constant of type `Incident_Defs.incident_code` with the following fields:

message object	An AD to the message object.
module	Number of the module in which this incident is defined.
number	A number for the specific incident within the module.
severity	A severity level.

When the application developer assigns a module number and a sequential number to an incident, these numbers must be unique within the environment in which they are visible.

The following example from `At_Support_Ex` defines an incident code:

```
-- Exception Codes:
msg_obj: constant System.untyped_word :=
        System.null_word;    -- use oeo

time_format_error_code: constant Incident_Defs.
    incident_code := (
        module         => 0,
        number         => 1,
        severity       => Incident_Defs.error,
        message_object => msg_obj);
```

The fields of the `time_format_error_code` incident code contain the following values:

`message_object`	Declared as a null AD (uses the default message file specified in the programs Outside Environment Object).
`module`	The module number of `At_Support_Ex`. The value is `0`.
`number`	The number of this incident in this module. The value is `1`.
`severity`	The value `error`.

The actual text of the message associated with an incident can be stored in a message file by one of the following three methods.

II-5.3.1.1 In the Source File

Include the actual text of the message in the source file with tagged comment lines which can be identified and extracted by `manage.messages`. The tag for the comment line in the example code is `*D*` (that is, `*D*` immediately follows the two dashes of an Ada comment line. The `extract.tagged_commands` utility extracts tagged lines and passes them to `manage.messages` which creates the message file. In this method, the text of the message is physically close to the definition of its associated incident, an advantage for small programs with few messages.

In the following code example, tagged comment lines are used to include message text in the program source file:

```
-- Exceptions:
--
--*D* manage.messages
--
time_format_error: exception;
-- Occurs when the time was not input in a proper
-- format
--*D*   store 0 1 time_format_error \
--*D*   :short = "$p1 is an improper time specification
--*D*The correct format is hh[:mm[:ss[.dd]]]"
```

II-5.3.1.2 In a Command File

Include the text of messages in a command file which is passed to `manage.messages` which in turn creates the message file. This method allows all messages for a program to be kept in a single file which can be edited or updated using any text editor.

II-5.3.1.3 Using `manage.messages`

Invoke `manage.program` then run `manage.messages` and use its runtime commands to create and update the text of messages. This method allows easy listing, searching and updating of individual message texts, an advantage for larger applications where consistency and coherence among messages is desirable. The following example shows the declaration for the `not_on_file` message, the tagged lines used by `manage.messages` to include the message in a message file and the declaration of the exception associated with the message.

```
not_on_file_code:  constant
    Incident_Defs.incident_code := (
        message_object =>
            Inventory_Messages.message_object,
        module         => module,
        number         => 5,
        severity       => Incident_Defs.error);

--*D*   store  :module = $module \
--*D*          :number = 5 \
--*D*          :msg_name = not_on_file \
--*D*          :short = "There is no parts
--*D*            record for part ID '$p1<part
--*D*            ID (index value)>' does not
--*D*            exist."

not_on_file:  exception;
  pragma exception_value(not_on_file,
                       not_on_file_code);
  -- Raised by "Read_parts_record" and
  -- "Rewrite_parts_record".
```

For more information on creating message texts, see the *BiiN™ Command and Message Guide.*

II-5.3.2 Writing a Message

Calls Used:

`Message_Services.Write_msg`
Formats and writes a message.

To write a message to the user's message window, specify:

`msg_id`	Incident code for the message.
`param(1...5)`	Parameter(s) to insert into the message text, if any.
`device`	Opened device to which message is sent. The user's opened message window by default.

For example in the `At_Cmd_Ex` example program, the message associated with incident code `prior_time_warning_code` is written as follows:

```
Message_Services.Write_msg(
    msg_id => At_Support_Ex.prior_time_warning_code);
```

For more information on writing message texts to accept message parameters, see the *BiiN™ Command and Message Guide.*

II-5.3.3 Associating an Incident Code With an Exception

Declarations Used:

`pragma exception_value`
Binds the value of an exception with a named incident code.

It is often useful to associate an Ada exception with an incident so that when the exception is raised, the incident code is implicitly available. `pragma exception_value` associates an exception with an incident code. This binding is illustrated with the following example from Inventory_File:

```
not_on_file_code:  constant
    Incident_Defs.incident_code := (
        message_object =>
            Inventory_Messages.message_object,
        module         => module,
        number         => 5,
        severity       => Incident_Defs.error);
not_on_file:  exception;
  pragma exception_value(not_on_file,
                         not_on_file_code);
  -- Raised by "Read_parts_record" and
  -- "Rewrite_parts_record".
```

II-5.3.4 Replacing an OS Exception With an Application Message

When the operating system raises one of its exceptions, that exception can be replaced with a more detailed local message. The following code from the Inventory_Files example program shows an update operation. When it is unsuccessful, Record_AM.invalid_record_address is automatically raised. The package raises its own exception, not_on_file and writes an explanatory message.

```

  -- Rewrite (update) parts record:
  --
  Record_AM.Keyed_Ops.Update_by_key(
      opened_dev => parts_file,
      buffer_VA  => parts_record'address,
      length     => parts_record'size/8,
      index      => part_ID_index_name);

exception

  when Record_AM.invalid_record_address =>
    Message_Services.Write_msg(
        msg_id => not_on_file_code,
        param1 => Incident_Defs.message_parameter(
            typ => Incident_Defs.txt,
            len => part_ID_index_str.length)'(
                typ      => Incident_Defs.txt,
                len     => part_ID_index_str.length,
                txt_val => part_ID_index_name));
    RAISE not_on_file;
```

II-5.3.5 Taking Advantage of Predefined OS Messages

The /msg directory contains predefined message files. These messages may be used by application programs. It is advisable to use the messages in the same context for which they were originally created. These messages may be reviewed with the list command of the manage.messages run-time command of the manage.program utility.

II-5.3.6 Pushing a Message When Raising an Exception

Calls Used:

Message_Stack_Mgt.Push_msg_1_param
: Pushes a message block with one parameter onto the caller's message stack.

The BiiN™ Operating System often pushes a message to the message stack prior to raising an exception. Applications may also push messages on the stack when raising exceptions in order to provide more information concerning the reason for abnormal program termination.

In the following example from `Inventory_Files`, the exception handler for the `Read_parts_record` procedure catches an attempt to read a part that is not on the file and writes the message associated with the `not_on_file_code`. It then pushes the `not_on_file` message on the message stack.

```
Message_Services.Write_msg(
    msg_id => not_on_file_code,
    param1 => Incident_Defs.message_parameter(
        typ => Incident_Defs.txt,
        len => part_ID.length)'(
            typ     => Incident_Defs.txt,
            len     => part_ID.length,
            txt_val => part_ID));
Message_Stack_Mgt.Push_msg_1_param(
    not_on_file_code);
```

II-5.3.7 Clearing the Message Stack When Handling an Exception

Calls Used:

`Message_Stack_Mgt.Clear_messages`
Discards all messages on the caller's message stack.

It is good practice, although not required, for a program that catches and handles an incident or error to clear the message stack. Otherwise on a later incident or error, the message stack contains the history of the previously handled errors as well as the error that caused termination. This can be confusing to people debugging the program.

In the following example from `Inventory_Files`, the exception handler for the `Read_parts_record` procedure catches an incomplete key value and writes the message associated with the `invalid_part_ID_code`. It then clears the message stack before pushing the current message.

```
Message_Services.Write_msg(
    msg_id => invalid_part_ID_code,
    param1 => Incident_Defs.message_parameter(
        typ => Incident_Defs.txt,
        len => part_ID.length)'(
            typ     => Incident_Defs.txt,
            len     => part_ID.length,
            txt_val => part_ID));
Message_Stack_Mgt.Clear_messages;
Message_Stack_Mgt.Push_msg_1_param(
    message_id => invalid_part_ID_code,
    param1     => Incident_Defs.message_parameter(
        typ => Incident_Defs.txt,
        len => part_ID.length)'(
            typ     => Incident_Defs.txt,
            len     => part_ID.length,
            txt_val => part_ID));
```

II-5.3.8 Writing a Message With Acknowledgement

Calls Used:

`Message_Services.Acknowledge_msg`
Writes a message with no <LF>, then reads and parses the user's response.

To write a message and receive a response from the user, use `Message_Services.Acknowledge_msg`.

The message should explain to the user the choices on which his response must be made. A positive acknowledgement currently results from `yes`, `ja`, `true` or `+`. Any other input, including just <CR>, returns false. The words `yes`, `ja` and `true` can be abbreviated to one letter.

If an opened device is specified, it is used both for writing a message and reading the response. Otherwise, the device from the caller's `user_dialog` entry in process globals is used. In any case, the device must be interactive as defined by `Device_Defs.device_info.common_info`. The call does nothing and returns false if the device is noninteractive.

If writing or reading fails for any reason, `false` is returned.

The following code is from `Example_Messages` (the acknowledge message) and `Create_name_space_cmd_ex` (code requesting affirmation from the user before storing a new name space as a directory entry).

```
overwrite_query_code:
    constant Incident_Defs.incident_code :=
    (0, 4, Incident_Defs.information, msg_obj);
--
--*D* store :module=0 :number=4 \
--*D*     :msg_name=overwrite_query_code \
--*D*     :short = \
--*D*     "$p1<pathname> exists.  Overwrite it?"
```

```
  -- Confirm overwrite:
  --
  overwrite :=
      Message_Services.Acknowledge_msg(
          Example_Messages.
            overwrite_query_code,
          Incident_Defs.
            message_parameter(
            typ => Incident_Defs.txt,
            len => name.max_length)'(
              typ     =>
                  Incident_Defs.txt,
              len     =>
                  name.max_length,
              txt_val => name));
end if;
```

II-5.3.9 Recording History Entries

Calls Used:

`History_Services.Record_message`
: Records a message in an opened history file, or in the caller's current job history, returning the record ID.

To record a message in a job's `history_log` file, record an individual message explicitly via `History_Services.Record_message`. This call returns a record ID.

II-5.3.10 Summary

- An *incident* is a BiiN™ construct that assigns a unique identifier, an *incident code*, to each error situation.
- An *incident code* identifies a message and a severity level for an incident.
- An *exception* is the Ada construct that signals the occurrence of errors or other exceptional situations that arise during program execution.
- A *message* is the human-readable text associated with an incident.
- A *message file* contains the short and long forms of messages in one or more language variations. The `message_object` field in the incident code references the application's message file.
- A *message block* contains an incident code (which includes the message's module and number) and any message parameters.
- A *message stack* is a fixed-length, open-bottomed stack that provides a means of keeping a trace of any incidents that have occurred within a process along with specific information about each incident.

Part III
Directory Services

This part of the *BiiN™/OS Guide* gives concepts and techniques for naming objects in directories and for protecting stored objects from unauthorized access.

This part contains these chapters:

Understanding Directories
Explains basic concepts needed to understand the system's directory mechanism.

Using Directories Provides techniques for using directories.

Protecting Stored Objects
Shows how to protect objects using IDs and authority lists.

Using Name Spaces
Shows how to use name spaces (lists of directories).

Creating Symbolic Links
Shows how to use symbolic links between directories or directory entries.

Directory Services contains the following services and packages:

Naming Service
```
Directory_Mgt
Name_Space_Mgt
Symbolic_Link_Mgt
```

Protection Service
```
Authority_List_Mgt
Identification_Admin
Identification_Mgt
User_Mgt
```

Part III Overview

UNDERSTANDING DIRECTORIES 1

Contents

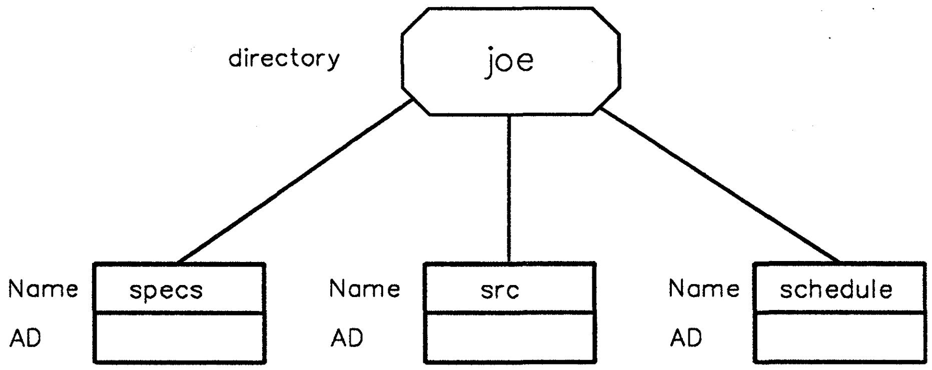


Figure III-1-1. Directories Contain <Name, AD> Pairs

Directories allow you to associate a name with the object's AD and store the <name, AD> pair in a directory (Figure III-1-1). Given a full pathname, you can find the object associated with that name.

This chapter explains in more detail the concepts of BiiN™ directories.

III-1.1 Directory Structure

In other systems, directories map names to files or directories only. By contrast, the directory service allows an AD for *any* type of object to be stored in a directory. This includes files, other directories, devices, programs, IDs, authority lists, data definitions, version groups, form definitions, report definitions, and so on.

All names within a given directory must be unique. The storage of names in directories is case-sensitive; that is, lowercase characters are distinct from uppercase (for example, `My_File` is distinct from `my_file`).

Examples of valid names include:

```
ADA_source      tools
Chapter-I.12    673-59-1257
%_of_cost       #s_per_sq_inch
2
```

A directory can contain another directory (a subdirectory), allowing for conventional tree structures and hierarchies. For example, in Figure III-1-2, `src` is a subdirectory of `joe`. A directory that contains an entry or another directory is the *parent* of that entry or directory.

A directory can also contain an `alias` entry. An alias entry is another name for an AD that already has a name. Because an AD can have any number of aliases, you can set up directory structures that are not limited to trees or strict hierarchies (Figure III-1-2).

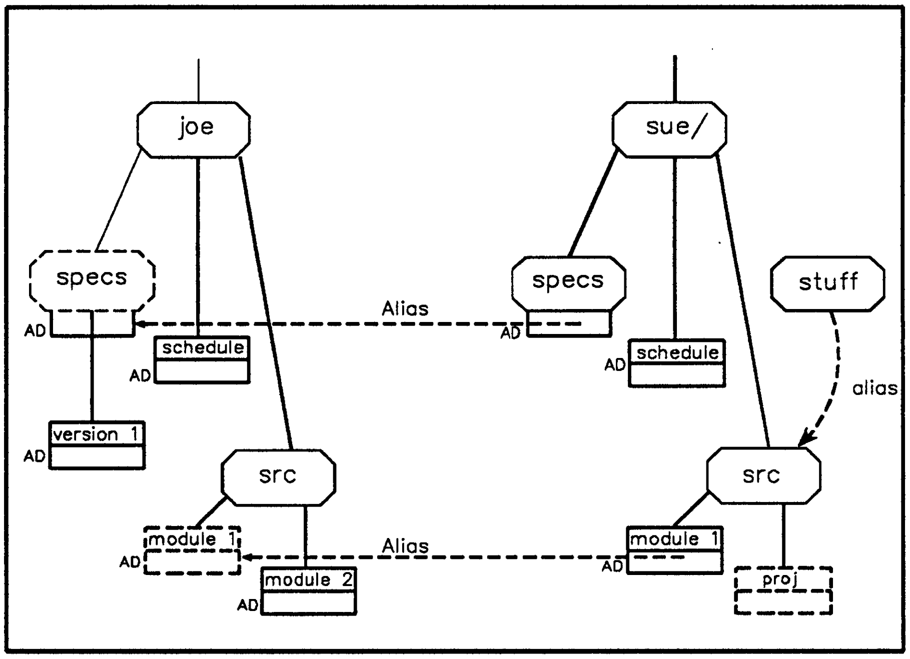


Figure III-1-2. A Directory Structure with Aliases

In Figure III-1-2, an example of an alias is the name `module 1` in directory `joe/src`. Joe has a name for this module as does Sue. Both names reference the same underlying object. Note that an alias can reference an AD on a different node in the distributed system. Deleting an alias has no effect on the referenced object, so Joe can delete his `module_1` without affecting Sue's.

Directories and subdirectories are common in other systems. Using BiiN™ directory services, however, a set of connected directories is not limited to tree structures. A single object can be stored in the same directory under different names (aliases) or in other directories under the same or different names. Directories can be linked together into meaningful, networked structures.

{correct this later. 6/24/88 - stanf} Cycles are allowed; that is, in Figure III-1-2, Sue's file `module_1` can be aliased to directory `sue`, even though `sue` is a parent of `sue/src`.

III-1.1.1 Pathname Syntax

See the "Pathname Syntax" appendix in the *BiiN™/OS Reference Manual* for an explanation of the different kinds of pathnames and their syntax.

III-1.1.2 Alias Entries and Master Entries

Calling `Directory_Mgt.Store` to store an AD with a name for the first time places an entry in the directory and the associated passivated AD is the *master AD*. Subsequent `Stores` of an AD for which a master already exists result in *alias* entries.

Storing an AD to an object doesn't always produce a master AD for the object. Only the first AD to cross the boundary between active and passive space as a result of a `Store` (or

`Create_directory`) or update of an object which contains the AD produces a master. If the first AD to cross the boundary from active to passive space does so as a result of some other operation, then subsequent `Stores` of the AD (or updates of its container) will NOT produce a master AD.

III-1.1.3 Symbolic Links

A symbolic link contains a pathname. Symbolic link evaluation retrieves whatever AD is stored with that pathname. If an AD to a symbolic link is stored in a directory entry, then retrieving from the entry does *not* return the entry's AD. Instead, an AD to the object referenced by the link is returned.

Aliases and symbolic links provide two ways to associate an AD with different names.

Both are useful in that they allow the user flexibility in the naming and symbolic referencing of objects.

Both aliases and symbolic links may be stored in any directory for which the user has store rights.

However, using an alias in a `Directory_Mgt.Delete` causes only the alias to be deleted; the underlying object is not affected. Using a symbolic link in a `Directory_Mgt.Delete` causes the symbolic link object itself (not the object referenced by the link's value) to be deleted.

An alias has the following advantages:

- It references the same object type.
- The alias may inherit "mastership," so that even if the master pathname is deleted, there may still be a named reference to the object (inheritance requires that the alias entry reside on the same volume set as the master AD).

An alias has this disadvantage:

- It references the same object type, i.e, the associated AD is "object instance" specific, so that if the underlying object is deallocated, the alias may be left as a dangling reference. For instance, if you have a program `/joe/prog` that is aliased by `/joe/bin/p` and you replace `/joe/prog` with a revised version of the same program, the `/joe/prog` alias will point to the outdated version. If the alias was `/vs2/joe/prog`, it would be a dangling reference to the old version.

A symbolic link has the following advantages:

- It references an object NAME. Any object can exist under this name at one time or another. This means you can also update an object under that name and not end up with a dangling reference as for aliases (you might want to replace an existing program with a revised version, or some such).
- You can set its value to a CL variable, for example, `$mybin`, which gives you a great deal of flexibility.

A symbolic link has the following disadvantages:

- The symbolic link cannot inherit "mastership" for the object referenced by the link's value.

- The associated link value is "name" specific, so that if a different object is stored under the same name, the user may end up accessing something incompatible with the type needed.

III-1.1.4 Protecting Directories and their Contents

Most OSs determine *who* has access to *what* programs and data using an owner/group/world mechanism. Access to programs and data depends on whether your group (or you, or everyone) has the authority to read, write, or execute a file.

The BiiN™ system extends the familiar three-level *owner / group / world* protection to flexible, multi-level protection. This is done with authority lists and IDs. The associated authority list protects the object. (Chapter III-3 discusses authority lists and IDs in more detail.)

Each caller is represented by a list of IDs which that caller can portray. By default, these IDs are the user ID, `ux_group`, and `world`, which the caller acquires during the logon session. When the caller tries to access a protected object, the caller's IDs are compared with the IDs in the authority list to determine access.

There can be any number of IDs in the authority list (Figure III-1-3), as opposed to the three allowed in *owner / group / world* protection. The authority list contains a list of IDs and associated type rights. For each ID, the type rights specify what the ID holder can do with the object protected by this authority list.

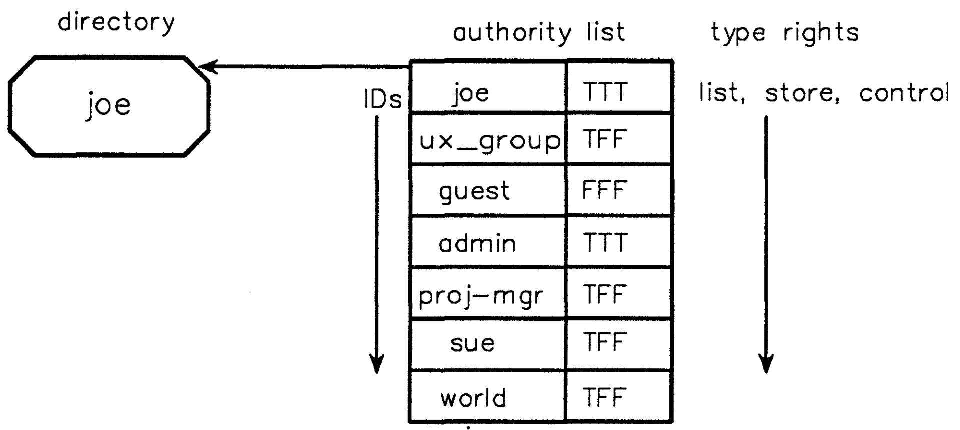


Figure III-1-3. A Directory is Protected with an Authority List

For example, according to the authority list associated with directory `joe` in Figure III-1-3, a caller holding the ID for `proj_manager` has list rights, and can list the contents of the directory. But a caller holding the ID for `guest` has no type rights and cannot access that directory at all.

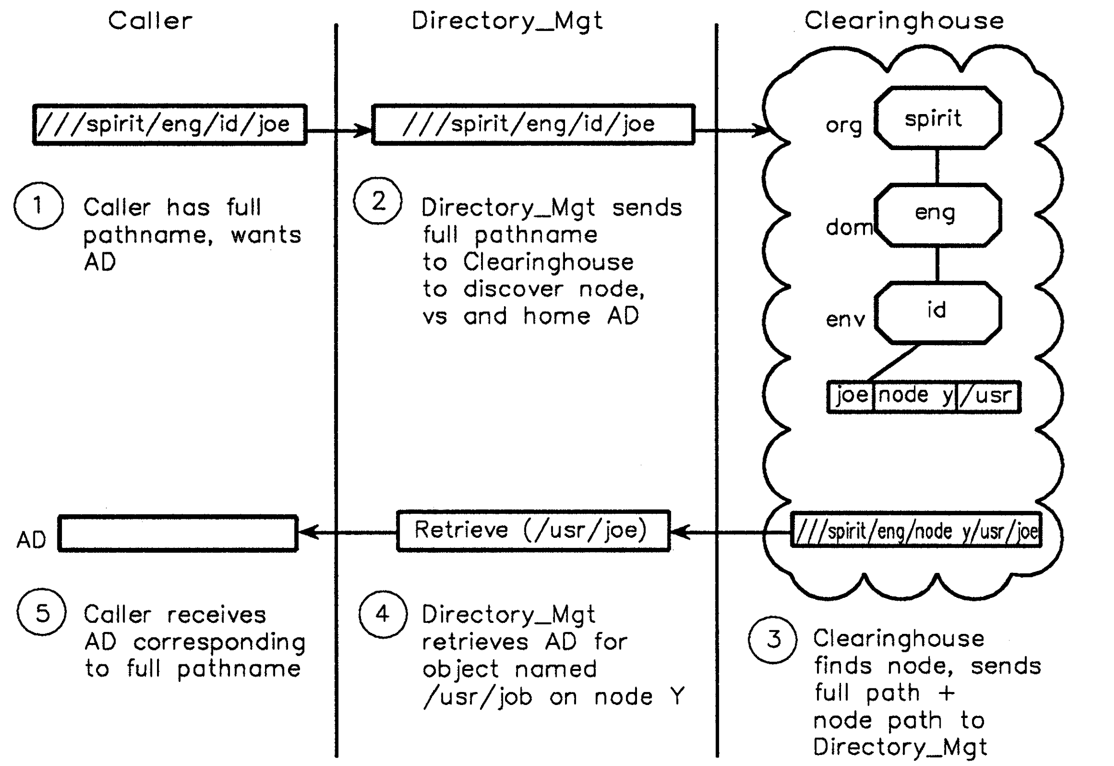
Chapter III-3 discusses authority lists and IDs in more detail.

III-1.2 The Clearinghouse: Naming in a Distributed System

The Clearinghouse maintains the database showing where objects are actually stored in a distributed system, by keeping track of where each volume set is mounted. (A volume set is a logical disk, and contains programs and objects of various types as well as files.)

Directory_Mgt goes to the Clearinghouse to find the node and volume set, then to the node and volume set to find the named object. This process is transparent to the caller, so that the caller does not need to know which node an object is stored on. (The caller also doesn't need to know about the Clearinghouse, beyond understanding its role in finding named objects.)

To illustrate, Figure III-1-4 shows the process Directory_Mgt goes through to return an AD for a full pathname.



Caller
Directory_Mgt
Clearinghouse
///spirit/eng/id/joe
///spirit/eng/id/joe
org
spirit
dom
eng
env
id
joe
node y
/usr
1 Caller has full pathname, wants AD
2 Directory_Mgt sends full pathname to Clearinghouse to discover node, vs and home AD
AD
Retrieve (/usr/joe)
///spirit/eng/node y/usr/joe
5 Caller receives AD corresponding to full pathname
4 Directory_Mgt retrieves AD for object named /usr/job on node Y
3 Clearinghouse finds node, sends full path + node path to Directory_Mgt

Figure III-1-4. Directory_Mgt **uses the Clearinghouse to Resolve Network Names.**

When specifying an organization or full pathname, for example to access an object or service in a different */organization/domain* in the distributed system, it is helpful to understand a little about how objects and services are stored and named over the distributed system. The distributed storage-and-naming system works like this.

Every object that has been Stored with Directory_Mgt.Store exists on a *volume set*, which is a logical disk. There can be many names for any volume set, and many volume sets attached to a particular node.

All the volume sets attached to all the nodes in a distributed system form the *passive space*. The passive space is grouped into *naming domains*, so-called to distinguish them from other kinds of domains in the BiiN™ system. A naming domain is identified by the org/dom part of a pathname; for example, spirit_motors/engineering. Each volume set is assigned to a naming domain, and a naming domain can contain one or more volume sets. All volume sets in a naming domain must have unique names.

Objects in the passive space are identified by *pathnames*. The BiiN™ system needs a way to identify objects anywhere in the distributed system, no matter where they are. To refer to an object in the same naming domain, you can simply use a node or relative pathname. To refer to an object in a different naming domain, you must use a pathname that begins with two or more slashes (organization or full pathname). The rest of this paper describes how full pathnames are built.

In BiiN™ systems, there are many valid pathnames for any object. However, there is one standard pathname, called the *canonical* patname, which uniquely identifies a passive object anywhere in a network of BiiN™ systems by specifying the volume set it is on and its pathname within the volume set. All utilities that result in full pathnames show them in canonical form. For example, the output of `list.current_directory` is always a canonical pathname.

A canonical full pathname looks like this:

///org/dom/`vs`*/vsname/pathname*

The parts of this pathname have the following meanings:

org	A BiiN™ distributed system can be divided into several *organizations*. The organization is the largest division in a distributed system. For example, a system for a large corporation might be divided by division (`systems`, `components`, and `software`) or by site (`portland`, `new_york`, `maui`, and `berlin`).
dom	Each organization can be divided into several *domains*. For example, the organization `systems` could be divided into the domains `engineering`, `doc`, `marketing`, `manufacturing`, and `shipping`, or alternatively `first_floor`, `second_floor`, and `basement`.
`vs`	`vs` is a predefined directory in each naming domain that holds the names of all *volume sets* in the naming domain. The literal word `vs` in a pathname indicates that the rest of the pathname refers to an object on one of the volume sets in the network. (`vs` is actually one of several *environments* in each domain. Two other environments are `home` and `node`, which will be discussed later. However, the `vs` environment is always used in the canonical pathname.)
vsname	This is the name of a particular volume set. Volume sets may have names like `vs1`, `vs2`, and `vs3` or `sys_volset`, `user_volset`, and `temp_volset`, or anything else.
pathname	The pathname that follows the volume set name traces the directories from the top directory of the volume set to the specified object. It uniquely identifies the object in the volume set.

For example, if `usr` is the name of a volume set, then the canonical pathname of the file `~jane/books/ssg` might be

```
///software/doc/vs/usr/jane/books/ssg
```

Note that this pathname does *not* specify the node on which the volume set `usr` is currently mounted. This pathname continues to be valid even if the volume set is moved to another node in the distributed system, as long as it remains in the same naming domain.

The canonical form is not the only way to refer to an object in the distributed system, especially if the object is in your naming domain. Here are some other valid ways of building pathnames:

- *///org/dom/*`home`*/username/pathname*

  This pathname identifies the object relative to the home directory of the specified user. For example, if `/usr/jane` is the home directory of user `jane`, the following full pathnames refer to the same object:

  ```
  ///software/doc/home/jane/books/ssg
  ///software/doc/vs/usr/jane/books/ssg
  ```

- *///org/dom/*`node`*/nodename/pathname*

  This full pathname identifies the object according to the node to which it is attached. For example, if the volume set `usr` in the example is attached to the node named `greedo`, the following full pathname names the same object as the one in the previous example:

  ```
  ///software/doc/node/greedo/usr/jane/books/ssg
  ```

III-1.2.1 A Node's Default Directories

The following directories are installed by the system in a node's `/sys` directory, thereby presenting a common set of directories at all BiiN™ nodes:

`aid`	An alias to the attribute ID directory on the system volume set.
`dev`	An alias to the device directory on the system volume set.
`home`	An active-only directory that provides access to the home Clearinghouse environment of the node's naming domain (see `CH_Client` in the *BiiN™/OS Reference Manual*). References of type `/home/jerry` resolve to an AD for the home directory of ID `jerry`.
`id`	An active-only directory that provides access to the ID Clearinghouse environment of the node's naming domain.
`node`	An active-only directory that provides access to the node Clearinghouse environment of the node's naming domain. A listing of "/node" lists the names of all the nodes belonging to the node's naming domain.
`rid`	An alias to the resource ID directory on the system volume set.
`sso`	An alias to the SSO directory on the system volume set (contains Scheduling Service Objects).
`sys`	An alias to the root directory of the node's system volume set.
`tdo`	An alias to the TDO directory on the system volume set (contains Type Definition Objects).
`vs`	An active-only directory that provides access to the vs Clearinghouse environment of the node's home naming domain.

In addition to these root directory entries, BiiN™/UX reserves the following entries:

```
/bin      /etc    /usr    /tmp
```

III-1.3 Directory Operations

III-1.3.1 Retrieving Entries

The most common directory call is `Directory_Mgt.Retrieve.` `Retrieve` takes a name such as `/usr/joe` as a parameter and returns an AD for the object stored under the specified name.

Storing an object's AD by name in a directory does not necessarily mean the object itself is also stored there. Directory names can reference objects stored in active memory or in passive store.

(Storing of objects in directories is distinct from the *filing service,* which stores data in traditional file structures. See Chapter III-3 for information about storing objects.)

III-1.3.2 Listing a Directory

To list the contents of a directory, you open it as a *read-only* device to be read with `Byte_Stream_AM` or `Record_AM`. The result is a stream of entry names in ASCII collating sequence. Associated ADs are *not* read.

In contrast to `Byte_Stream_AM` or `Record_AM`, the `Open` calls in `Directory_Mgt` allow a *pattern* to be specified that is used to filter the stream of names. Only those entry names in the directory that match the pattern will be read. A pattern is a combination of plain characters which simply match the identical characters in a name, and *pattern operators* each of which matches a sequence of zero or more characters in a name.

The pattern operators are:

?	Matches any single character.
*	Matches zero or more characters.
[*amz*]	Where *amz* denotes zero or more characters. Matches any of the single characters within brackets.
[*a*-*z*]	Where *a* and *z* are single characters. Matches all ASCII characters >= *a* and <= *z*. Match always fails if *z* < *a*.
\	Escape character. Interprets the following character literally and not as a pattern operator. Must precede any of `?`, `*`, `[`, `]` that are to be matched.
!	Not (negation). Makes sure the character immediately following does NOT match. For example, `a[!b]c` matches every 3-character string beginning with `a` and ending with `c`, *except* the string `abc`.

`Directory_Mgt` in the *BiiN™/OS Reference Manual* lists which access method calls are supported and the exceptions that can be raised.

III-1.3.3 Process Globals and Directories

In the BiiN™ system, every process has *process globals* that determine the environment in which the process executes.

Process globals carry the following items pertinent to directories:

home directory	Location after successful login, that is, initial current directory. Set by a system administrator.
current directory	Current location in a directory structure and usual starting directory for evaluating relative pathnames.
command name space	Default directory list to search for commands during name evaluations started, for instance, by Human Interface Services.

authority list	Default authority list, to protect objects for which ADs are being stored with a name for the first time, when the directory in which the ADs are being stored has no default authority list.

III-1.3.4 Directory Operations and Transactions

These calls automatically participate in the caller's transaction:

```
Create_directory
Delete
Get_name
Open_directory_by_name
Rename
Retrieve
Store
```

If there is no caller's transaction, `Create_directory`, `Rename`, and `Delete` start their own transactions.

`Directory_Mgt` calls are atomic; when carried out within a transaction, if the transaction aborts their effects are undone, whether or not the directory call has already successfully completed.

The `Directory_Mgt` package description in the *BiiN™/OS Reference Manual* describes transaction locking.

III-1.3.5 Standalone Directories

A normal directory is integrated into the system's directory structure. Occasionally, however, it's useful to create directories that are independent of the system's directory structure. `Standalone_Directory_Mgt` creates such directories. The entries in a standalone directory are managed with normal `Directory_Mgt` calls.

Standalone directories differ from normal directories in several important ways:

- Normal directories have names, whereas standalone directories are identified only by their ADs (that is, they do not have names).
- Normal directories are created and passivated in an existing parent directory. Standalone directories are created in the active space; it is the caller's responsibility to passivate the standalone before using it. The caller must update the standalone before trying to use it; failure to do so will raise an exception during calls on the standalone.
- A normal directory resides on the same volume set as its parent directory; a standalone directory's home volume set depends on where the caller passivates the standalone's master AD.
- Entries in a normal directory are always protected by an authority list. By default, standalones also protect their entries with an authority list; however, if a standalone is created with the `no_authority` parameter set to `true`, the entries in the standalone are not protected by an authority list.

  Once a standalone is created, the user cannot later add or remove the protecting authority list. (An existing list can be replaced.)
- In normal directories, ownership of the directory is assigned to whomever makes the `Create_directory` call. Similarly, by default, ownership of a standalone directory is assigned to whomever first passivates the directory. That is, ownership is assigned to the `user_ID` of the calling process.

It is possible to create normal directories in standalones. As with normal directories, the caller can also invoke `Directory_Mgt` calls on entries within this structure.

For example, a caller may wish to define a database with two components: a database description and a set of associated ADs to components of the database. One approach would be to define the database using two ADs, one to the descriptor, one to a standalone directory containing the related AD set. In a simplified scenario, the caller would act as follows:

1. Create the database and database descriptor.
2. Call `Create_standalone_directory` to create a standalone in the active space (this operation does not store and update the standalone directory).
3. Copy ADs for the descriptor and standalone ADs into the database.
4. Call `Passive_Store_Mgt.Update` to passivate the database and its embedded objects (that is, the descriptor and standalone).
5. The caller may create and store entries in the standalone, and perform other calls common to directories.

A standalone directory can be deleted from the system by calling `Passive_Store_Mgt.Destroy`, which will destroy the standalone directory and any entries it contains. A standalone may also be deleted implicitly as a result of master AD deletion, for instance, by deleting an object that contains the standalone's master AD.

To prevent unwanted deletion of standalone directory entries, the caller might call `Destroy` from a utility that asks the user for confirmation before completing destructive operations.

III-1.4 Summary

- Directories associate names with objects by storing <name, AD> pairs in the directory.
- `Directory_Mgt.Retrieve` is an important call to obtain an AD for an object in the BiiN™ directory structure.
- Directory entry names can be listed using `Byte_Stream_AM` or `Record_AM`.
- When listing directory contents, the names can be "filtered" so that only names that match a pattern are listed.
- Directories can be set up with hierarchies, subdirectories, and aliases to other directories, across the entire distributed system (crossing node boundaries).

USING DIRECTORIES 2

Contents

Directories allow you to name and organize objects in a BiiN™ system. You can name an object by associating a name with the object's AD and storing the <name, AD> pair in the directory (Figure III-2-1). Given a name, you can then find any object in the system. This chapter gives some specific techniques for using directories.

Packages Used:

`Directory_Mgt` Manages directories and directory entries.

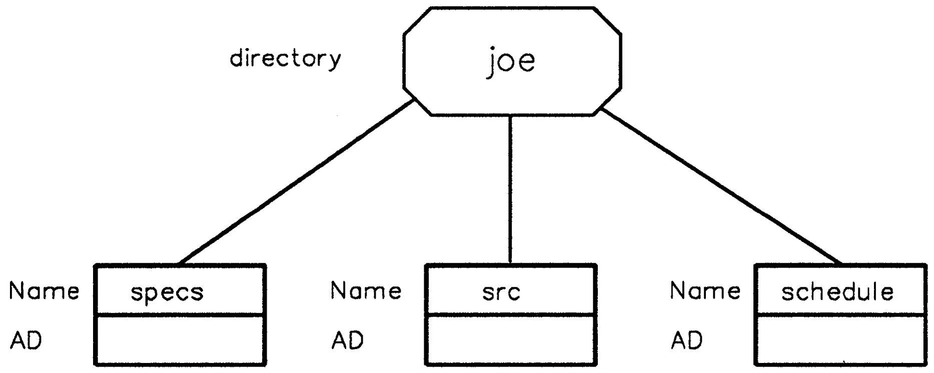


Figure III-2-1. Directories Contain <Name, AD> Pairs.

After reading this chapter, you will be able to:

- Create a directory
- Store a directory entry
- Retrieve a directory entry
- Delete a directory entry
- List a directory
- Use a pattern to filter a directory listing
- Retrieve a directory from process globals.

Complete listings of the following examples can be found in Appendix X-A.

III-2.1 Creating a Directory

The simplest way to create a directory is to call `Directory_Mgt.Create_directory`, specifying the pathname of the new directory and using defaults for the rest of the parameters. The pathname must be a `System_Defs.text` record.

Calls Used:

`Directory_Mgt.Create_directory`
Creates a directory.

The following example from procedure `Create_directory_command_ex` creates a new directory with the name given as input. This excerpt shows just the declarations and statements to create the directory:

```
dir_name:  System_Defs.text(252);
  -- Name of the directory to be created.

dir_AD:  Directory_Mgt.directory_AD;
  -- Newly created directory's AD; returned
  -- but not used by "create.directory".
. . .
Command_Handler.Get_string(
   cmd_odo    => opened_command,
   arg_number => 1,
   arg_value  => dir_name);
. . .
dir_AD := Directory_Mgt.Create_directory(
    name => dir_name);
```

The `Create_directory` call automatically:

- Stores a master AD for the new directory in the parent directory.
- Creates a representation of the new directory in passive store.
- Assigns an authority list to protect the new directory, either the parent directory's default authority list or the default authority list in process globals.
- Sets the owner of the new directory to the caller's ID.
- Returns the new directory's AD to the caller with all type rights.

You then have a new directory ready for use.

III-2.2 Storing an AD in a Directory

The simplest way to create a directory entry is to call `Directory_Mgt.Store`, specifying the new pathname and the object's AD, and using defaults for the other parameters.

Calls Used:

`Directory_Mgt.Store`
Creates a new directory entry: AD and name.

The calling process must have store rights in the parent directory for the call to succeed. The calling process will have store rights if the calling process:

- Created the target directory
- Has become its owner, or
- Is granted store rights by the authority list protecting the parent directory.

`Directory_Mgt.Store` cannot overwrite an existing entry.

If the AD is the first AD stored in passive store for the object, then:

- The directory entry is the object's *master AD*,

- The caller's ID is the object's owner,
- Either the parent directory's authority list or the process globals authority list protects the object, if the authority list parameter is defaulted.

If there are subsequent stores of the same AD under different names, the subsequent entries are alias entries and the object's owner remains the master AD's owner.

Note that storing the AD for the object does *not* store the object itself. To update the object's passive version, you must call `Passive_Store_Mgt.Request_update` after `Directory_Mgt.Store`.

The following example from procedure `Named_copy_ex` stores an AD in a new directory entry:

```
source:  System_Defs.text;
dest:    System_Defs.text)
. . .
    source_AD := Directory_Mgt.Retrieve(source);
    dest_AD   := Passive_Store_Mgt.
                 Create_copy_stub(source_AD);
    Directory_Mgt.Store(name   => dest,
                        object => dest_AD);
```

III-2.3 Retrieving a Directory Entry

Retrieving a directory entry is a common way to obtain an AD for a named object in the BiiN™ system. To retrieve a directory entry, use `Directory_Mgt.Retrieve`. `Retrieve` accepts a name (and optional directory and ID) and returns an AD for a directory entry.

Calls Used:

`Directory_Mgt.Retrieve`
Returns AD associated with pathname.

The following excerpt from the `Make_object_public_ex` example procedure retrieves an AD for the ID `world`.

```
-- Get the world ID AD
world_name:  constant System_Defs.text(9) :=
    (9, 9, "/id/world");
world_untyped:  constant System.untyped_word :=
    Directory_Mgt.Retrieve(world_name);
```

III-2.4 Deleting a Directory Entry

To delete a directory entry, use `Directory_Mgt.Delete`, giving the pathname and using the defaults for the other parameters.

The calling process must have list and store rights in the parent directory for a `Delete` to succeed.

If the AD is the object's master AD (the first AD stored in passive store) and no alias entries exist for this object on the same volume set, then deleting the AD deletes the object's passive version.

If the AD is the master AD and alias entries do exist on the same volume set, then the OS converts one of the alias entries to the master AD, and the object's passive version is not deleted.

III-2.5 Listing a Directory

To list the contents of a directory, open the directory as a device and use `Byte_Stream_AM` or `Record_AM` to read the opened device. The result is a list of entry names.

Remember that ADs are `Retrieved`; names are `Read`.

Calls Used:

`Directory_Mgt.Open_directory`
: Given a directory AD, opens directory for sequential reads.

`Byte_Stream_Am.Ops.Read`
: Reads bytes from opened device.

`Byte_Stream_Am.Ops.Write`
: Writes bytes to opened device.

`Byte_Stream_AM.Ops.Close`
: Closes an opened device.

The following example from the `List_current_directory_cmd_ex` example procedure uses the following steps:

1. Opens directory as an input device.
2. Opens standard output.
3. Sets up a buffer.
4. Sets up a read/write loop: reads bytes from directory into buffer, writes from buffer to standard output.

```
procedure List_current_directory_cmd_ex
  --
  -- Function:
  --   Lists names of entries in user's current
  --   directory.
  --
  --   Each entry name is written to the user's
  --   standard output, on a separate line.
. . .
is
. . .
  opened_dir:  Device_Defs.opened_device;
    -- Opened device for reading stream of names
    -- from user's current directory.

  standard_output:  Device_Defs.opened_device :=
      Device_Defs.opened_device(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.standard_output));
    -- User's standard output.

  name_buffer:  array(1 .. 250) of character;
    -- Each entry name is read into this buffer
    -- and then written from it.

  length:  System.ordinal;
    -- Length in bytes (characters) of last
    -- entry name read.
. . .
begin
. . .
  -- Open directory for reading, filtered by
  -- ":pattern":
  --
  opened_dir := Directory_Mgt.Open_directory(
      dir     => Directory_AD_from_untyped_word(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.current_dir)),
      pattern => pattern);


  -- Get and write each entry name:
  --
  loop

    length := Byte_Stream_AM.Ops.Read(
        opened_dev => opened_dir,
        buffer_VA  => name_buffer'address,
        length     => name_buffer'size/8);

    Byte_Stream_AM.Ops.Write(
        opened_dev => standard_output,
        buffer_VA  => name_buffer'address,
        length     => length);

  end loop;

exception

  when Device_Defs.end_of_file =>

    Byte_Stream_AM.Ops.Close(opened_dir);

    RETURN;

end List_current_directory_cmd_ex;
```

III-2.6 Using a Pattern to Filter a Directory Listing

To filter a directory listing according to a pattern, use `Directory_Mgt.Open` or `Directory_Mgt.Open_directory_by_name`. When you specify a pattern to these calls, only the directory entries that match the pattern are returned by `Reads`.

For example, you could add a `pattern` specification to the call `Open_directory`. The `pattern` must be a text record conforming to `System_Defs.text`. The following example from `List_current_directory_cmd_ex` "filters out" those entries beginning with a period (those that match pattern `!.*`):

```
--*D*   define.argument pattern \
--*D*       :type = string
--*D*       set.lexical_class symbolic_name
--*D*       set.maximum_length 252
--*D*       set.value_default "*"
--*D*     end
. . .
pattern:  System_Defs.text(252) := (252, 252, (others => ' '));
  -- Optional ":pattern" used to select entries
  -- matching the pattern, such as "abc?" or
  -- "m*device".  Default is "!.*", meaning all
  -- entries NOT beginning with a "." (period).
. . .
-- Open directory for reading, filtered by
-- ":pattern":
--
opened_dir := Directory_Mgt.Open_directory(
    dir     => Directory_AD_from_untyped_word(
        Process_Mgt.Get_process_globals_entry(
            Process_Mgt_Types.current_dir)),
    pattern => pattern);

```

III-2.7 Retrieving a Directory from Process Globals

The call `Process_Mgt.Get_process_globals_entry` allows you to retrieve one of the two directory ADs in the process's process globals. A process's globals contain the ADs for two directories: the home directory of the process's `user_ID` and the current directory.

Calls Used:

`Process_Mgt.Get_process_globals_entry`
Retrieves a value from a slot in process globals.
`Directory_Mgt.Get_name`
Gets the full pathname of an object's master AD.

The following example from `Show_current_directory_cmd_ex` retrieves the name of the current directory from process globals with the following calls:

1. `Process_Mgt.Get_process_globals_entry` gets the AD for the current directory.
2. `Directory_Mgt.Get_name` gets the name associated with the AD of the current directory.

```
procedure Show_current_directory_cmd_ex
. . . .
is
. . . .
  current_dir:   Directory_Mgt.directory_AD :=
      Directory_Mgt.directory_AD(
          Process_Mgt.Get_process_globals_entry(
              Process_Mgt_Types.current_dir));
    -- Current directory's AD.

  current_dir_untyped: System.untyped_word;
     FOR current_dir_untyped USE AT
         current_dir'address;
    -- Current directory's AD as an untyped word.

  dir_name:  System_Defs.text(252);
    -- Current directory's name.
. . . .
begin

  -- Get current directory's pathname:
  --
  Directory_Mgt.Get_name(
      obj  => current_dir_untyped,
      name => dir_name);
. . . .
end Show_current_directory_cmd_ex;
```

PROTECTING STORED OBJECTS 3

Contents

This chapter shows you how to protect stored objects from unauthorized access, using IDs and authority lists.

Packages Used:

`Identification_Mgt`	Provides operations to manage IDs and ID lists.
`Authority_List_Mgt`	Provides calls to manage authority lists and to evaluate a caller's access rights to objects protected by authority lists.
`User_Mgt`	Provides calls to manage a user's protection set and user profile.

Objects may be protected with *authority lists* and *IDs*.

An authority list shows *which IDs* can access the object, with what access rights. An ID identifies *what agent* is trying to access the object. A process carries IDs for agents it may represent in an *id list* (Figure III-3-1).

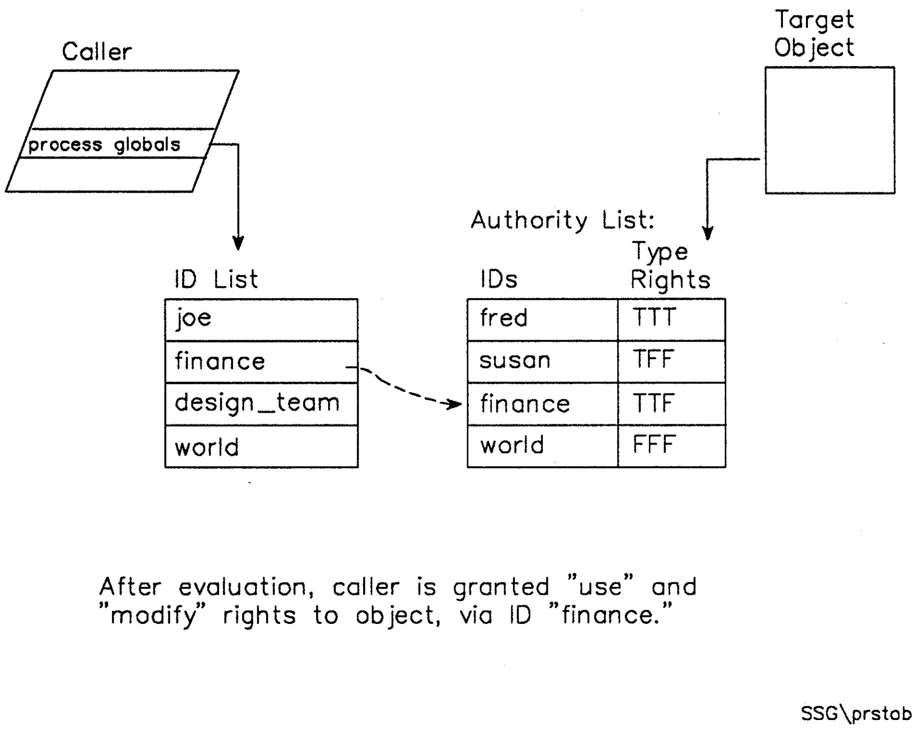


Figure III-3-1. A Caller Accesses a Protected Object

In Figure III-3-1, the caller carries IDs for `joe`, `finance`, `design_team` and `world`. When this caller tries to access an object, all these IDs are used in *evaluating* the caller's access to the object. (Evaluation is discussed in more detail later in this chapter.)

The object itself is protected by an authority list. In the authority list, ID `fred` has all rights, ID `susan` has "use" rights, ID `finance` has "use" and "modify" rights, and ID `world` has no rights. When a caller tries to access this object, these <ID, type rights> pairs are used in evaluating the caller's access to the object.

III-3.1 Concepts

The following concepts present authority-list-based protection from a user standpoint.

III-3.1.1 Why Objects Need Authority-Based Protection

When you store an object, you must protect it with *authority-based protection*. This is distinct from the address space protection mechanism provided by ADs. Basically, authority lists are intended to extend the architecture's capability-based protection (ADs) into passive store.

An object is stored in passive store similar to the way files are stored in a conventional filing system. If there were no authority-list protection, the object would be accessible to any user over an entire distributed BiiN™ system, not just to the caller who stored the object. This presents a problem: how can the object itself be protected from unauthorized access while in passive store, which is accessible from the entire distributed system? Authority lists provide a solution.

You associate the object with an authority list. To oversimplify, the authority list specifies exactly which IDs, with what type rights, can access the object. Thereafter, any caller's ID must appear in the authority list, with the proper type rights, for the caller to access the object. (Evaluation is discussed in full in Section III-3.1.6).

III-3.1.2 IDs Identify the Caller

An *ID* represents an entity, either an individual or an *access class*. An individual is usually a user (`joe`). An access class may represent a collection of users (`design_group`), a program (`database`) or all "outsiders" (`world`).

Typically, each individual has a unique ID, which is created by the system administrator when creating a new user. The system administrator may also define various access classes within the system and create IDs for them, so that users, by holding an ID to one or more access classes, may also portray themselves as members of these classes.

The caller carries IDs in an *ID list* which is stored in the caller's process globals. The first ID in the list is the caller's user ID. The ID list can contain one or more IDs. For example, in addition to the caller's user ID, a single caller might carry IDs for the following:

another user (`joe`)
a group of users (`design_team`)
a program (`db_data_entry`)
a group of programs (`cad_system`)
a generic ID (`world`)

To access an object, one of a caller's IDs must match an ID in the object's authority list, with the proper rights. Access to the object is evaluated according to the rights associated with that ID in the authority list. (This is oversimplified; more on evaluation in Section III-3.1.6).

In addition to the IDs in the caller's process globals, many `Directory_Mgt` calls accept an explicit ID. This is especially useful for system utilities that may require rights for an ID that is not available in the ID list itself.

III-3.1.2.1 What's In an ID?

Figure III-3-2 shows the parts of an ID.

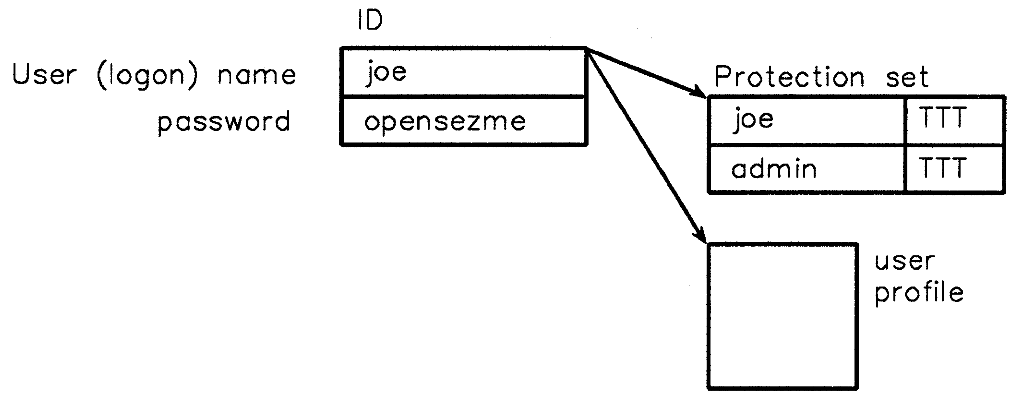


Figure III-3-2. Parts of an ID

User (logon) name	Name for this ID.
Protection set	A *protection set* protects an ID just as an authority list protects a stored object. IDs are protected with protection sets instead of authority lists because IDs are maintained in the Clearinghouse, not in passive store. In Figure III-3-2, the protection set for ID *joe* allows all type rights to callers *joe* and *admin.*
Password	Password for this ID. Originally set by the system administrator, and changeable by anyone with control rights to the ID.
User profile	Originally set by the system administrator, and some parts changeable by anyone with control rights to the ID.

IDs and ID lists are active-memory-only objects, maintained through `Identification_Mgt` and the Clearinghouse. Thus, calls to `Passive_Store_Mgt` on IDs and ID lists will raise exceptions.

IDs are created with the `Identification_Admin` package.

III-3.1.3 A Process's ID List

The caller's ID list is in the caller's process globals. By convention, the OS always interprets the first ID in a process's ID list as the *user ID*. (By default, the second ID in the list is the *group ID* for BiiN™/UX applications.) *BiiN™/UX User's Guide*). See Figure III-3-3 for an illustration of an ID list.

Any caller can obtain an AD to its ID list with `Process_Mgt .Get_process_globals_entry` or can list the contents of an ID list with `Identification_Mgt.List_IDs`, but *setting* the ID list in the process globals can only be done using the `Process_Admin` or `Job_Admin` packages.

The caller's ID list is inherited by child processes, just as other items in process globals are inherited.

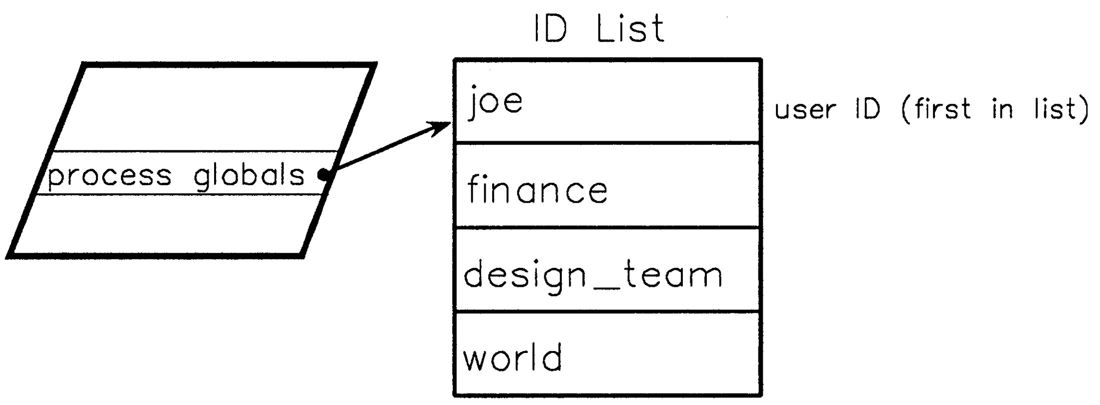


Figure III-3-3. A Process's ID List

III-3.1.4 Type Rights on an ID

The following type rights are defined for IDs:

Portray rights	Needed to enter an ID into a process's ID list.
Control rights	Needed to change an ID's password or to set an object's owner. The user ID in a process's ID list must also have control rights.

By default, users have portray and control rights to their own user IDs.

Portray rights are acquired by being passed an ID AD with such rights, or through rights evaluation. ID rights can be amplified to control and portray rights by providing the correct password to `Identification_Mgt.Portray_ID`.

III-3.1.5 Authority Lists Specify Who Can Access Objects

An authority list is composed of a *protection set*, a record containing <ID, type rights mask> pairs. Normally, the caller who stores an object assigns the authority list, either specifying one or using the default.

An authority list is an object in itself, separate from the object it protects. As objects, authority lists need to be stored using `Directory_Mgt` and updated using `Passive_Store_Mgt`; these calls are *not* done automatically.

Both active and stored objects can be protected by authority lists, and any number of objects can share a single authority list, thus saving storage space (Figure III-3-4).

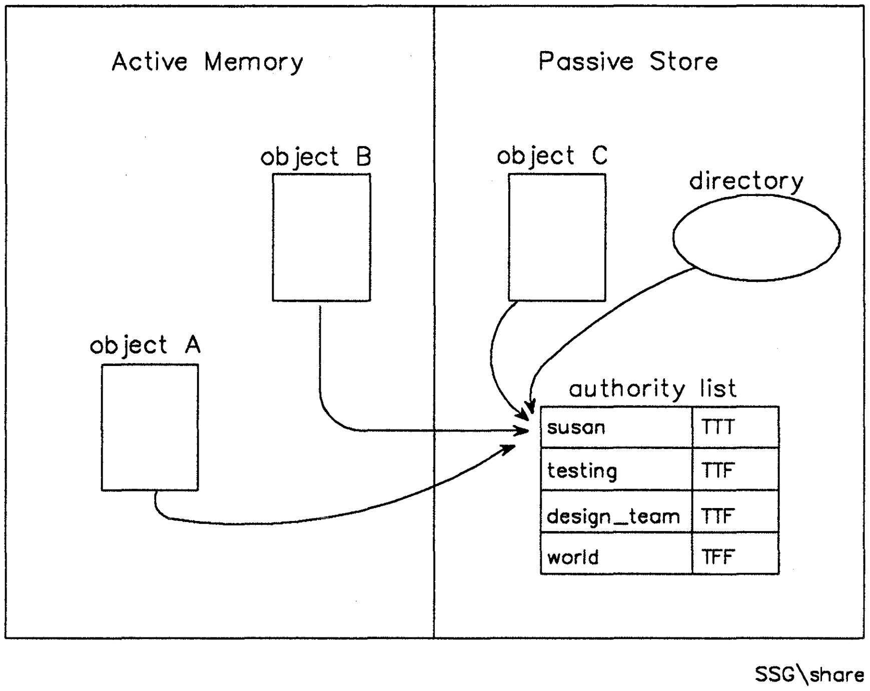


Figure III-3-4. Multiple Objects Sharing an Authority List

III-3.1.6 How a Caller's Access Rights to an Object Are Evaluated

Whenever a caller retrieves or activates an AD, the caller's access rights to that object are *evaluated.* That is, the caller's IDs are checked against the authority list, to return the proper type rights on the underlying object.

`Directory_Mgt.Retrieve` does an implicit `Authority_List_Mgt.Evaluate` against retrieved ADs before returning the result to the caller.

Object activation, which is done transparently by `Passive_Store_Mgt`, also does an `Evaluate`; however, evaluation differs somewhat between a `Retrieve` and activation. The following sections discuss each evaluation process.

III-3.1.6.1 Evaluating Access During a Retrieve

Figure III-3-5 shows the steps in the evaluation during a `Directory_Mgt.Retrieve`.

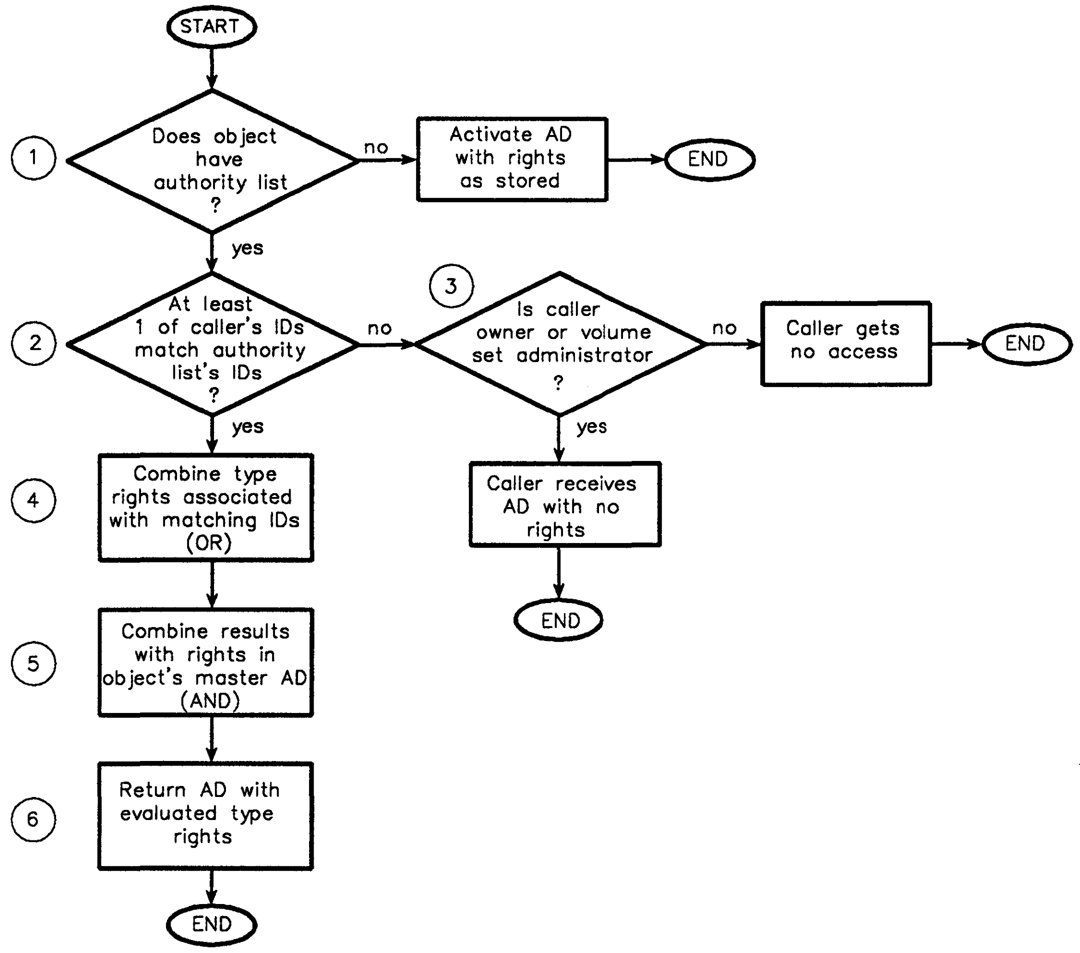

START
1
Does object have authority list ?
no
Activate AD with rights as stored
END
yes
2
At least 1 of caller's IDs match authority list's IDs ?
no
3
Is caller owner or volume set administrator ?
no
Caller gets no access
END
yes
yes
4
Combine type rights associated with matching IDs (OR)
Caller receives AD with no rights
END
5
Combine results with rights in object's master AD (AND)
6
Return AD with evaluated type rights
END

Figure III-3-5. Evaluating Access During a Retrieve

1. If the object has an authority list, proceed to step 2. Otherwise, activate the object, granting the same type rights as when the object was stored, and end evaluation.
2. If at least one of the caller's IDs matches an ID in the object's authority list, evaluation continues at step 4. If the caller's IDs do not match any in the authority list, evaluation proceeds to step 3.
3. If the caller is the object's owner or volume set administrator, the caller receives an AD with no type rights (no authority list access) and evaluation ends. If the caller is not the object's owner or volume set administrator, the caller gets the exception `Directory_Mgt.no_access` and evaluation ends.
4. The type rights associated with the matching IDs in the authority list are combined (logical OR). This results in the maximum type rights for that caller and that authority list.
5. The maximum type rights are then compared (ANDed) with the type rights in the object's master AD. This results in the least type rights for that caller and that object. That is, the caller can never get more rights than are present in the object's master AD.
6. The caller receives an AD with the final evaluated type rights.

For example, consider the caller, object, and authority list in Figure III-3-6.

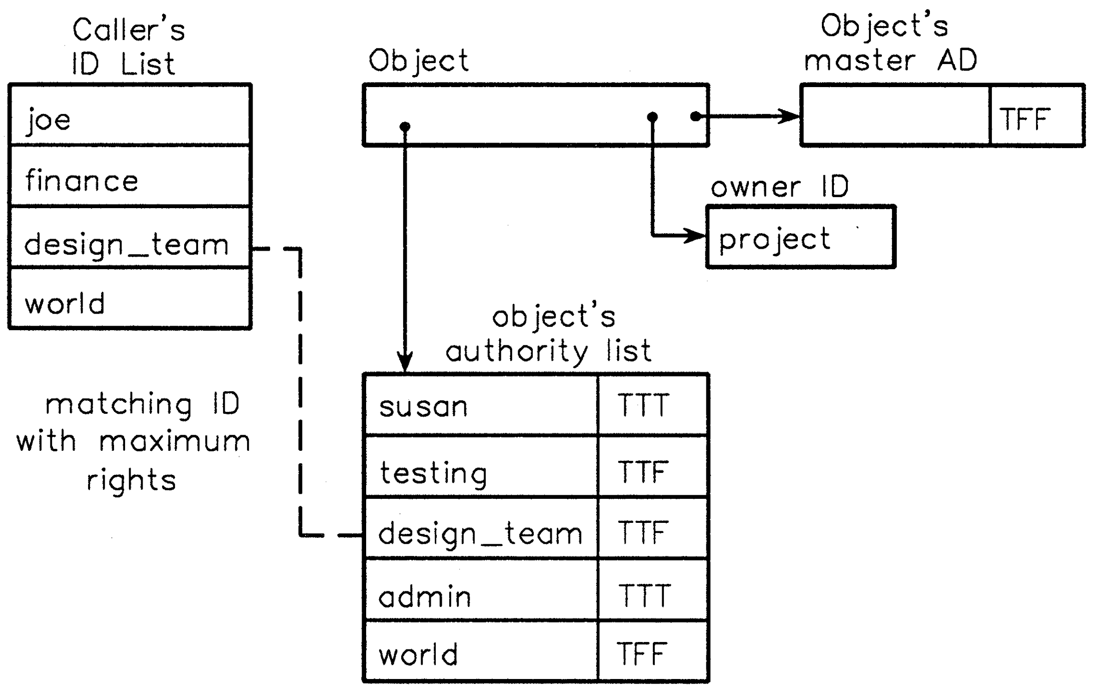


Figure III-3-6. Example: Evaluating Access During a Retrieve

1. The caller's IDs *design_team* and *world* match IDs in the object's authority list.
2. Type rights associated with ID *design_team* are "use" and "modify". Type rights associated with ID *world* are "use". A logical OR between these two results in type rights "use" and "modify".
3. The type rights in the object's master ID are "use" only. A logical AND between these rights, and the results of the OR operation gives "use" rights only.
4. The caller receives an AD for the object with "use" rights.

III-3.1.6.2 Evaluating Access Rights During Activation

A caller's access to an object is also evaluated when activating the object's AD. If access is not granted, a null AD is activated in place of the AD that should be activated, instead of raising `Directory_Mgt.no_access`. See the `Passive_Store_Mgt` package for more information about AD activation.

III-3.2 Techniques

After reading this section, you will be able to:

- Get information about an object's protection
- Use default protection
- Create an authority list

- Change a directory's default authority list
- Change an object's owner and authority list.

Creating IDs is a privileged operation for the system administrator; see the *BiiN™ Systems Administrator's Guide*.

III-3.2.1 Getting Information about an Object's Protection

The following calls are used to get information about an object's ID and authority lists.

Calls Used:

`Identification_Mgt.Get_object_owner`
Returns the owner ID AD of the object.

`Authority_List_Mgt.Get_object_authority`
Returns AD for the object's authority list.

`Authority_List_Mgt.List_authority`
Returns the set of authority list entries.

`Directory_Mgt.Get_default_authority`
Retrieves directory's default authority list.

`Authority_List_Mgt.Evaluate`
Returns type rights on object.

`Identification_Mgt.List_IDs`
Returns the set of IDs from the ID list.

Note that the calls `List_authority` and `List_IDs` require the caller to receive results in an **out** variable.

III-3.2.2 Using Default Protection

Normally, what happens by default is all the protection you need. The usual way to store an object with authority list protection is to use `Directory_Mgt.Store`, accepting the target directory's default authority list as the object's protecting authority list.

III-3.2.3 Creating an Authority List

In general, to avoid unexpected results, an authority list should be stored and updated before being assigned to protect objects.

Calls Used:

`Authority_List_Mgt.Create_authority`
Creates an authority list.

To create an authority list:

1. Create a protection set (list of <ID, type rights> pairs) in the form required by `User_Mgt.protection_set`.

2. Create the new authority list with `Create_authority`, specifying the protection set. You will receive an AD, with control rights, to the new authority list.
3. Store the new authority list AD with `Directory_Mgt.Store`.
4. Passivate the new authority list with `Passive_Store_Mgt.Request_Update`.

The following example from `Make_object_public_ex` shows how to create a simple authority list for an object, allowing all type rights for the *world* ID.

```
procedure Make_object_public_ex(
. . .
is
. . .
  -- Define the protection set
  entries:   constant User_Mgt.protection_set(1) := (
      size  => 1, length => 1,
      entries => (1 => (rights => (true, true, true),
                        id     => world_id)));

  -- Create the authority list
  aut_list:  constant
      Authority_List_Mgt.authority_list_AD :=
      Authority_List_Mgt.Create_authority(entries);
  aut_untyped:  System.untyped_word;
  FOR aut_untyped USE AT aut_list'address;

begin
. . .
    Directory_Mgt.Store(aut_list_path, aut_untyped);
    Passive_Store_Mgt.Request_update(aut_untyped);
. . .
end Make_object_public_ex;
```

Once the authority list has been created, stored, and updated, you can then associate that list with any object.

III-3.2.4 Changing a Directory's Default Authority List

You may want to change a directory's default authority list to another authority list. Note that `Directory_Mgt.Create_directory` sets the default authority list to null; the caller must act to set a directory's default authority list.

Calls Used:

`Directory_Mgt.Set_default_authority`
Sets directory's default authority list.

A directory's default authority list is the first one `Store` looks for when a master AD is stored with default protection.

The default authority list of a directory is not necessarily the authority list that protects the directory itself.

III-3.2.5 Changing an Object's Owner and Authority List

Calls Used:

`Identification_Mgt.Portray_ID`
Returns ID AD with control and portray rights.

`Identification_Mgt.Set_object_owner`
Sets or changes the owner ID of an object.

`Authority_List_Mgt.Set_object_authority`
Associates a new authority list with an object.

To change an object's owner:

1. Obtain the new owner ID AD with `Directory_Mgt.Retrieve`.
2. Obtain control rights to the new owner ID with `Identification_Mgt.Portray_ID`.
3. Replace the object's current owner with a new owner with `Set_object_owner`. The caller's ID (either passed or default user ID) must match the old owner ID, and must have control rights. By default, `Set_object_owner` uses the caller's user ID, which has control rights.

To change an object's authority list:

1. Replace the object's authority list with a new authority list via `Set_object_authority`. The caller's ID (either passed or default user ID) must match the owner ID, and must have portray rights. By default, `Set_object_authority` uses the caller's user ID, which has portray rights.

III-3.3 Summary

- Objects may be protected with *authority lists* and *IDs*.
- An authority list shows *which IDs* can access the object, and what type rights they can acquire.
- An ID identifies *what caller* is trying to access the object.
- A caller carries one or more IDs in an ID list which is stored in the caller's process globals. The first ID in the ID list is the caller's `user ID`.
- A protection set protects an ID just as an authority list protects a stored object.
- Whenever a caller retrieves or activates an AD, the caller's access to that object is *evaluated.*
- During a `Retrieve`, the caller's IDs are compared with the object's authority list and master AD to return the proper rights on the retrieved AD.
- During AD activation by `Passive_Store_Mgt`, the "containing" object's owner ID is compared with the object's authority list to return the proper rights on the activated AD.

USING NAME SPACES 4

Contents

A name space is a list of directories to be searched when looking for an object. This is similar in function to the UNIX-like `path` environment variable or the MS-DOS `PATH` command. This chapter gives concepts and techniques for creating a name space.

Packages Used:

`Name_Space_Mgt`
Provides calls to manage name spaces (lists of directories).

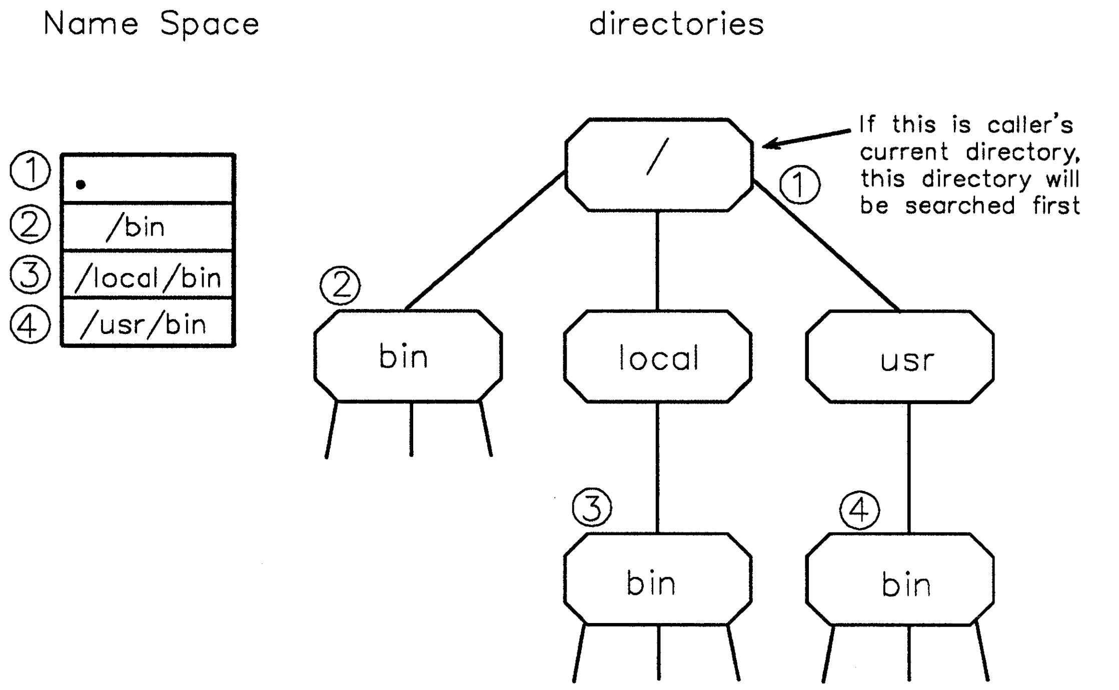

Name Space
directories
① .
② /bin
③ /local/bin
④ /usr/bin
/
If this is caller's current directory, this directory will be searched first
①
②
bin
local
usr
③
④
bin
bin

Figure III-4-1. A Name Space Lists Directories to be Searched

III-4.1 Concepts

A name space contains a string list. Each string list is the name of a directory.

III-4.1.1 A Name Space is a List of Directories

Directories in a name space are searched in the order in which they appear. For example, in Figure III-4-1, `Directory_Mgt` first looks in directory `/bin`, then in directory `/local/bin`, then in directory `/usr/bin`. If the "current directory", represented by dot (.), is in the name space, the directory that is current at the time the call is made will be searched.

Each user's user profile references a *command name space*, used by CLEX when searching for commands.

The directories in the name space are used *only* for retrieving and listing. That is, no `Store`/`Delete`/`Rename` or other `Directory_Mgt` calls are allowed on the listed directories.

Opening a name space does *not* open any directories in the name space. Instead, directories are opened as encountered during `Reads`. Thus, the first directory in the name space is opened at the beginning of the first `Read` request. Rights evaluation is performed against listed directories when they are opened, to make sure list rights are present in the directories.

As name space `Reads` progress, the current opened directory is closed and the next directory in the name space is opened. When the last directory in the list reaches `end_of_file`, the name space is also marked as at `end_of_file`.

A pattern may be specified to select only names which satisfy the pattern (see Chapter III-2 for an example of using a pattern).

Name spaces are constants and cannot be modified once they have been created.

III-4.1.2 How a Name Space References Directories

`Reads` on name spaces return names (not ADs), just like `Reads` on directories.

When read using `Byte_Stream_AM` `Reads`, the names are separated by an ASCII newline character; for `Record_AM` `Reads`, each name is returned as a record.

Note that if a directory's pathname is renamed after the name space is created, the directory cannot be opened in the name space because the name space won't be able to find it.

Relative pathnames are usually avoided in name spaces, because you want to use the same name space regardless of your starting directory. An exception to this is the current directory (.) which is often the first element in a name space.

`Reads` on name spaces do not participate in transactions and the directory currently being read is *not* locked.

III-4.2 Techniques

After reading this section, you will be able to:

- Create a name space
- Change the command name space in the user profile
- Change the command name space in process globals.

III-4.2.1 Creating a Name Space

To create a name space, use `Name_Space_Mgt.Create_name_space`, specifying the list of directories. The list must conform to `System_Defs.string_list`.

Calls Used:

`Name_Space_Mgt.Create_name_space`
Creates a name space containing text entries.

The following is from the example `Create_name_space_cmd_ex` in Appendix X-A. The developer uses the `Command_Handler` package to get the new name space's name and parent directory from user input.

```
procedure Create_name_space_cmd_ex
...
is
...
  directory_list:  System_Defs.string_list(508);
    -- String list containing pathnames of the
    -- directories in the new name space.
...
  Command_Handler.Get_string_list(opened_cmd, 2,
      arg_value => directory_list);
...
    name_space := Name_Space_Mgt.Create_name_space(
        directory_list);
...
        Directory_Mgt.Store(name, name_space_untyped);
...
    Passive_Store_Mgt.Request_update(
        name_space_untyped);
...
end Create_name_space_cmd_ex;
```

III-4.3 Changing a User's Command Name Space

To change a user's command name space in the user profile, use `User_Mgt`:

1. Use `User_Mgt.Get_user_profile` to get the current user profile record.
2. Change the `command_path` component of the `user_profile` record to contain the desired new command path, of type `System_Defs.string_list`.
3. Use `Set_user_profile` with the new `user_profile` record to insert the new command name space in the user's profile.

It is the responsibility of the one modifying a user profile to guarantee the validity of names in the profile.

III-4.4 Changing the Command Name Space within a Job or Process

The call `Process_Mgt.Set_process_globals_entry` allows you to insert a name space into its slot in process globals, to be effective for the duration of the job or process.

Calls Used:

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

Note that as for any object, the name space should be created and passivated before being assigned to process globals.

The following example from `Process_Globals_Support_ex` shows setting the `cmd_name_space` slot in process globals.

```
procedure Set_command_name_space(
  ns:  Name_Space_Mgt.name_space_AD)
  --
  --  Logic:
  --      1. Check that "ns" is a name space.
  --      2. Set the new command name space.
is
  ns_untyped:  System.untyped_word;
    FOR ns_untyped USE AT
        ns'address;
begin
  if not Name_Space_Mgt.
      Is_name_space(ns_untyped) then
    RAISE System_Exceptions.type_mismatch;

  else Process_Mgt.Set_process_globals_entry(
      slot  => Process_Mgt_Types.cmd_name_space,
      value => ns_untyped);
  end if;

end Set_command_name_space;
```

III-4.5 Summary

- A name space is a list of directories to be searched when looking for an object.
- Each user's user profile references a *command name space*, used by CLEX when searching for commands.

CREATING SYMBOLIC LINKS 5

Contents

A symbolic link provides a way to associate another name with an object already stored under a different name. This chapter gives concepts and techniques for creating a symbolic link.

Packages Used:

`Symbolic_Link_Mgt`
Provides calls to create, list, and identify symbolic links.

Figure III-5-1 diagrams a symbolic link.

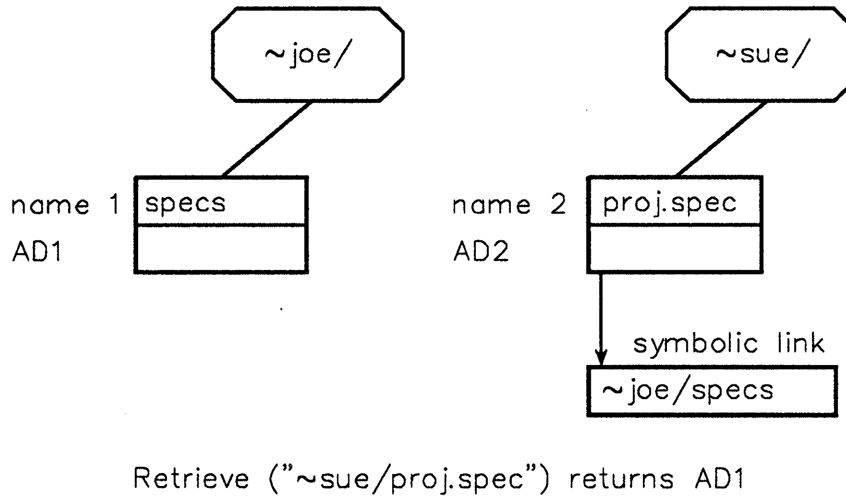


Figure III-5-1. A Symbolic Link

III-5.1 Concepts

A symbolic link contains a pathname. Symbolic link evaluation retrieves whatever AD is stored with that pathname.

If an AD to a symbolic link is stored in a directory entry, then retrieving from the entry does *not* return the entry's AD. Instead, an AD to the object referenced by the link is returned.

For example, in Figure III-5-1, a `Directory_Mgt.Retrieve("~sue/proj.spec")` returns `AD1`.

It is also possible to suppress the link. For example, in Figure III-5-1, you can obtain `AD2` by suppressing link evaluation of `~sue/proj.spec`.

III-5.1.1 Suppressing Link Evaluation

The at sign (`@`) suppresses link evaluation.

If `Directory_Mgt.Retrieve` is called with a pathname that contains an at sign (`@`), the part of the pathname preceding the at sign is evaluated. If the resulting object has the link attribute, the link evaluation is suppressed and the AD of the named object itself is used to complete the evaluation.

For example, in Figure III-5-1, a `Directory_Mgt.Retrieve("~sue/proj.spec@")` returns `AD2`.

The one exception to this rule is `Directory_Mgt.Delete` when the name supplied is the name of the link object itself. In this case, whether or not there is a trailing at sign, the link object itself is deleted.

III-5.1.2 How Symbolic Links Compare with Aliases

Symbolic links and aliases provide two different ways to associate another name with an object that already has a name. A symbolic link can be thought of as a "soft link," and an alias as a "hard link."

A symbolic link is a new name for a new object that contains the name of an existing object.

An alias is a new name for an object that already has a name.

Aliases can become master ADs, whereas symbolic links can't.

A symbolic link has the following advantages:

- It references an object *name*. Any object can exist under this name at one time or another. This means you can also update an object under that name and not end up with a dangling reference as for aliases (you might want to replace an existing program with a revised version, or some such).
- You can set its value to a CL variable, for example `$mybin`, which gives you a great deal of flexibility.

A symbolic link has the following disadvantages:

- The symbolic link cannot inherit "mastership" for the object referenced by the link's value.
- The associated link value is "name" specific, so that if a different object is stored under the same name, the user may end up accessing something incompatible with the type needed.

III-5.1.3 Symbolic Links and Links in General

Symbolic links are one implementation of the OS link attribute as defined by `Link_Mgt`. You may also provide your own implementation of the link attribute, so that a `Directory_Mgt.Retrieve` will execute your implementation when it retrieves your object with the link attribute. See the `Link_Mgt` package for information about implementing the link attribute.

III-5.2 Techniques

After reading this section, you will be able to:

- Create a symbolic link.

III-5.2.1 Creating a Symbolic Link

To create a symbolic link, use `Symbolic_Link_Mgt.Create_symbolic_link`, specifying the pathname within the link. An AD to the link is returned.

Calls Used:

```
Symbolic_Link_Mgt.Create_symbolic_link
```
Creates a symbolic link.

III-5.3 Summary

- A symbolic link provides a way to associate a name with an object stored under a different name.
- Symbolic link evaluation retrieves whatever AD is stored with that pathname.
- The at sign (@) can suppress link evaluation, to allow you to retrieve the AD of the symbolic link.

Part IV
I/O Services

This part of the *BiiN™/OS Guide* gives concepts and techniques for managing files, terminals, windows, printers, and other devices using byte stream, record, and character display I/O.

The chapters in this part are:

I/O Services contains the following services and packages:

basic I/O service:
```
     Byte_Stream_AM
     Device_Defs
     Simple_File_Admin
```

character terminal service:
```
     Character_Display_AM
     Character_Terminal_Mgt
     Terminal_Admin
     Terminal_Defs
     Terminal_Info
     Window_Services
```

print service:

```
    Printer_Admin
```

spool service:
```
    Spool_Defs
    Spool_Device_Mgt
    Spool_Queue_Admin
```

filing service:
```
    File_Admin
    File_Defs
    Record_AM
```

database support service:
```
    Join_Interface
    Record_Processing_Support
    Sort_Merge_Interface
    Trusted_Record_Processing_Support
```

data definition service:
```
    Data_Definition_Mgt
    DDF_Utility_Support
    Field_Access
```

volume set service:
```
    Volume_Set_Admin
    Volume_Set_Defs
    VSM_Disk_Admin
    VSM_Disk_Support
```

basic disk service:
```
    Basic_Disk_Mgt
```

basic streamer service:
```
    Basic_Streamer_Mgt
```

null device service:
```
    Nuldev_Mgt
```

UNDERSTANDING I/O ACCESS METHODS 1

Contents

This chapter describes the I/O *access methods* provided by System Services. An access method is a set of operations for accessing devices. Applications interact with devices through access methods. The I/O access methods provided by the BiiN™ Operating System are:

Packages Used:

`Byte_Stream_AM`	Provides device-independent I/O using streams of bytes.
`Record_AM`	Provides device-independent record I/O.
`Character_Display_AM`	Provides device-independent I/O to character display devices such as printers, plotters, and windows on character and graphics terminals.

The BiiN™ programming languages provide their own calls for I/O. You may prefer to use them when writing code which must be portable between operating systems.

IV-1.1 Devices

A *device* is an object that supports one or more I/O access methods, and which represents a hardware or software system device. `Device_Defs.device` is the Ada type for ADs to devices.

Devices may be implemented in hardware or software. Device types provided by the BiiN™ Operating System include:

basic disk
basic streamer
command input device
directory
file
form
name space
pipe
print device
report
window

Terminals and disks are normally not accessed directly as devices, but are accessed through windows and files, respectively.

IV-1.2 Opened Devices

When a device is opened, an *opened device* is created. Opened devices are I/O channels to devices. Each access method `Open` call creates a new opened device. `Device_Defs.opened_device` is the Ada type for ADs to opened devices.

When performing I/O, an application selects an access method and passes a device to be opened to its `Open` call. An opened device representing an I/O channel is returned. The opened device is then passed as a parameter when making I/O calls on the channel. See Figure IV-1-1.

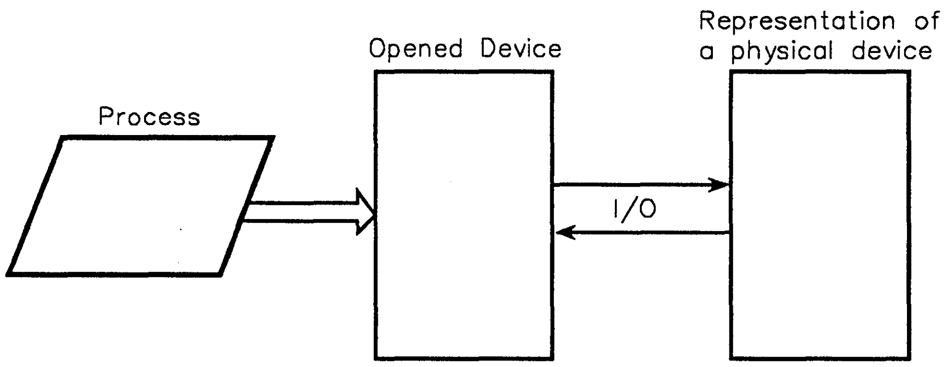


Figure IV-1-1. Opened Devices are I/O Channels to Devices

Zero or more opened devices can be active simultaneously for a given device. Depending on the device, subsequent I/O calls may either be restricted to the access method used to open the device, or allowed to be from any supported access method.

A `Close` call closes a single opened device. I/O processing with a device is active until all opened devices associated with the device are closed. When a job terminates, any opened devices which have not been closed by the application are closed. The access method `Is_open` calls can be used to determine whether a job's opened devices have been closed.

Multiple processes, in one or more jobs, can share a single opened device. An opened device exists until it is closed by all jobs that reference it.

IV-1.3 Concurrent Access to Opened Devices

Asynchronous I/O allows the execution of a process to proceed concurrently with the execution of I/O operations it has called. A process that opens a device is free to pass the opened device to other processes in the same or different jobs. These processes can then make concurrent I/O calls.

Multiple opened devices can be open concurrently on the same device. For example, multiple jobs can concurrently open and read a file. This increases efficiency and allows a process the option of waiting for the completion of I/O from multiple devices.

I/O operations are normally performed synchronously. For input calls, however, a process may choose not to block, and to wait for the arrival of input after having made an input call. Blocking can be avoided or postponed by the `Enable_input_notification` and `Disable_input_notification` calls common to all the access methods. These calls activate and deactivate asynchronous notification to processes of the arrival of input from an opened device.

A device manager acts as a traffic cop, supervising record locking, serialization of process I/O, etc. The application, however, can explicitly stipulate the degree of concurrency which can be exercised with an opened device. `Device_Defs.allow_mode`, a parameter in access method `Open` calls, defines what concurrent I/O is allowed on an opened device. These modes include:

`readers` - Allows only *readers* to share a device. A process is considered a reader when it opens a device for `input` or `input_partial`.

`anything` - Allows other readers or writers to share a device. Synchronization is the application's responsibility.

`nothing` - Allows no concurrent use of a device.

Figure IV-1-2 shows two processes communicating concurrently with the same opened device.

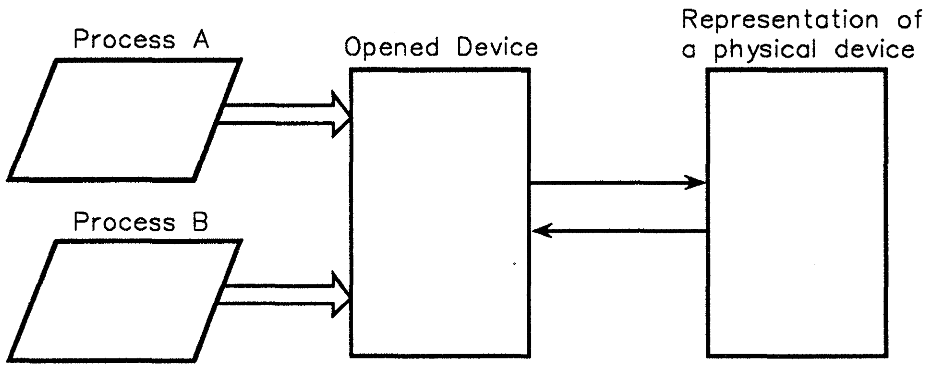


Figure IV-1-2. Concurrent I/O

IV-1.4 Device Independence

Each access method is supported by multiple devices. See Figure IV-1-3. This enables an application to be independent of any particular device. An access method is a set of operations that give a caller a pre-defined I/O channel to one or more devices.

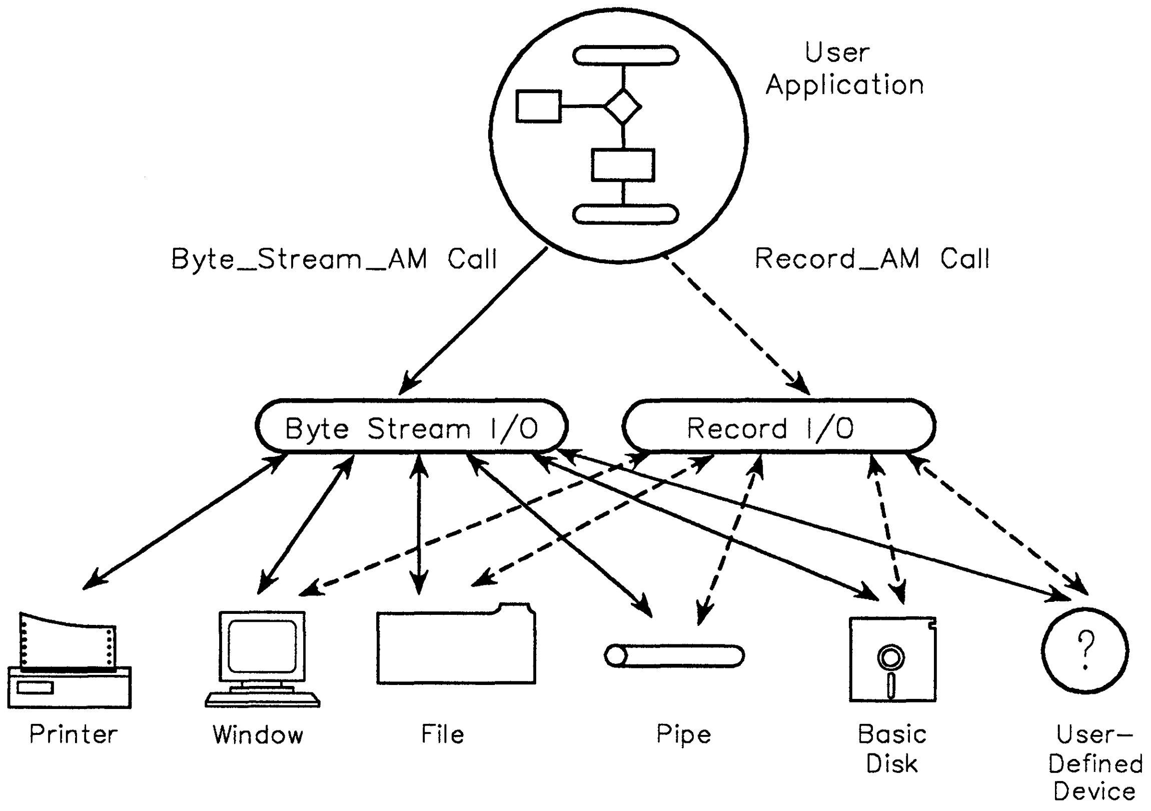


Figure IV-1-3. Access Methods are Supported by Multiple Devices

BiiN™ Operating System *attributes* provide a method for defining general purpose operations supported by multiple object types. This method enables type-specific implementations of general purpose operations, such as access method calls, to be bound to a specific object type, such as a device type. Devices with access method attributes can perform I/O via a standard access method interface. For example, an application may send records to a report device and to a file. Both devices contain the record I/O attribute and are able to interpret the data in their own way.

Devices may often be accessible by more than one access method. For example, an application calls the byte stream `Open` for a device which supports both byte stream and record I/O. The opened device, although opened with a byte stream I/O call, can be used for making record I/O calls.

Table IV-1-1. Devices and Supported Access Methods

	ACCESS METHOD		
Device	**Byte Stream**	**Record**	**Char Display**
basic disk	X	X	
basic streamer	X	X	
command input device	X	X	
directory	X	X	
file	X	X	

	ACCESS METHOD		
Device	Byte Stream	Record	Char Display
form		X	
name space	X	X	
pipe	X	X	
print device	X	X	X
report		X	
window	X	X	X

IV-1.5 How Access Method Implementations Can Vary

Devices may be opened in one of the following modes:

`input` - Input. Read rights required.

`input_partial` - Input. Read rights required. A file can be opened in this mode even if all volumes used by the file are not online.

`output` - Output. Write rights required.

`inout` - Input and output. Read and write rights required.

The modes in which a device can be opened depend on the type of device. For example, the device manager should not allow a printer to be opened in `input` mode.

A device manager must implement every operation of the access method being supported, although it need not support them. An instance of an access method operation may raise `System_Exceptions.operation_not_supported`. Some operations are common to all access methods. All others are access method-dependent.

IV-1.6 BiiN™ Operating System I/O Access Methods

The byte stream, record, and character display access methods have the following operations in common:

Open_by_name
Ops.Close
Ops.Disable_input_notification
Ops.Enable_input_notification
Ops.Flush
Ops.Get_device_info
Ops.Get_device_object
Ops.Is_open
Ops.Open

Table IV-1-2. Common I/O Operations

IV-1.6.1 Byte Stream I/O

An application should always be able to use byte stream I/O for all devices. (Forms and reports are exceptions since they are designed for record data.) Therefore, a device manager is expected to provide at least an implementation of the byte stream access method.

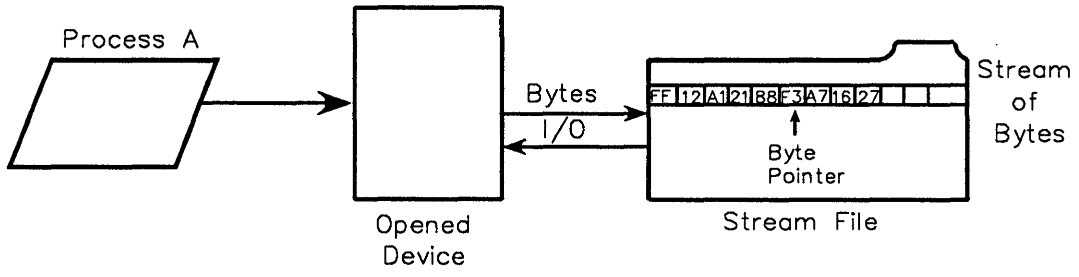


Figure IV-1-4. Byte Stream I/O

Table IV-1-3 lists selected byte stream access method calls.

Byte_Stream_AM.Ops.At_end_of_file
Byte_Stream_AM.Ops.Read
Byte_Stream_AM.Ops.Set_position
Byte_Stream_AM.Ops.Write

Table IV-1-3. Selected Byte Stream Access Method Calls

IV-1.7 Record I/O

Record I/O is used to read and write records. Forms and reports are specifically designed to be used with this access method.

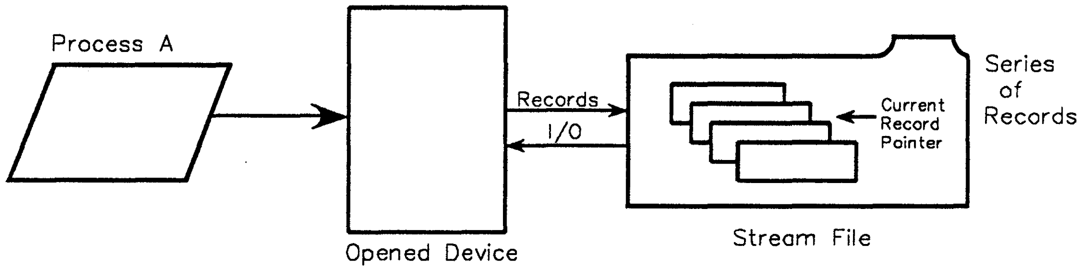


Figure IV-1-5. Record I/O

Records can be accessed in any of four ways:

Physical sequential
Physical random
Indexed sequential
Indexed random.

This access method provides access by index and key, and provides file and record locking.

Table IV-1-4 lists selected record access method calls.

Record_AM.Ops.Delete
Record_AM.Keyed_Ops.Delete_by_key
Record_AM.Ops.Get_DDef
Record_AM.Ops.Insert
Record_AM.Ops.Lock_all
Record_AM.Ops.Read
Record_AM.Keyed_Ops.Read_by_key
Record_AM.Keyed_Ops.Read_key_value
Record_AM.Keyed_Ops.Set_key_range
Record_AM.Ops.Set_open_mode
Record_AM.Ops.Set_position
Record_AM.Ops.Unlock_all
Record_AM.Ops.Update
Record_AM.Keyed_Ops.Update_by_key

Table IV-1-4. Selected Record Access Method Calls

IV-1.8 Character Display I/O

The character display access method is typically used to access devices with two-dimensional display surfaces.

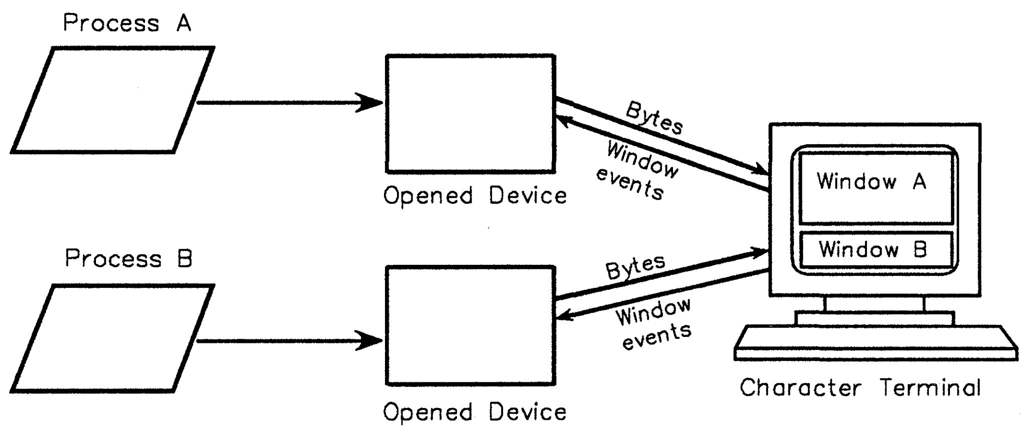


Figure IV-1-6. Character Display I/O

Character_Display_AM.Ops.Clear
Character_Display_AM.Ops.Clear_to_bottom
Character_Display_AM.Ops.Clear_to_end_of_line
Character_Display_AM.Ops.Delete_char
Character_Display_AM.Ops.Delete_line
Character_Display_AM.Ops.Get_cursor_position
Character_Display_AM.Ops.Insert_char
Character_Display_AM.Ops.Insert_line
Character_Display_AM.Ops.Move_cursor_absolute
Character_Display_AM.Ops.Move_cursor_relative
Character_Display_AM.Ops.Read
Character_Display_AM.Ops.Ring_bell
Character_Display_AM.Ops.Set_input_type_mask
Character_Display_AM.Ops.Write

Table IV-1-5. Selected Character Display Access Method Calls

IV-1.9 Standard I/O Connections

A process normally executes in an environment with four existing I/O assignments:

```
standard_input
standard_output
standard_message
user_dialog
```

These I/O assignments are declared in `Process_Mgt_Types.process_globals_entry` and are retrieved using `Process_Mgt`. (See the *BiiN™/OS Reference Manual.*) They are part of a caller's default environment and are contained in *process globals*, objects that hold user information about a specific process. `user_dialog` is normally a window which is guaranteed to be an interactive device for situations in which a timely response is needed.

The information in the process globals for a specific process is available to any user with an AD to the process. See Chapter VI-2 for more information on process globals.

IV-1.10 Summary

- An access method is a set of operations for accessing devices. Applications interact with devices through access methods.
- A *device* is an object that supports one or more I/O access methods, and which represents a hardware or software system device.
- Opened devices are I/O channels to devices.
- Multiple opened devices can be open concurrently on the same device.
- A process that opens a device is free to pass the opened device to other processes in the same or different jobs.
- BiiN™ Operating System *attributes* provide a method for defining general purpose operations supported by multiple object types.

- A process normally executes in an environment with four existing I/O assignments: `standard_input`, `standard_output`, `standard_message`, and `user_dialog`.

USING BASIC I/O 2

Contents

You use the *byte stream access method* or the *record access method* to perform basic I/O on an *opened device*. This chapter describes the techniques for using the `Byte_Stream_AM` and `Record_AM` packages to perform basic I/O.

Packages Used:

`Device_Defs`	Provides declarations common to different I/O access methods.
`Byte_Stream_AM`	Provides device-independent I/O using streams of bytes.
`Record_AM`	Provides device-independent I/O for one-record-at-a-time access.

The examples used in the following sections are parts of two procedures: `Output_bytes_ex` and `Output_records_ex`. See Appendix X-A for the complete listings.

IV-2.1 Opening and Closing an I/O Device

When opening a device with an open call from the `Byte_Stream_AM` package, the call returns an opened device. When you are through using an opened device, you remove access to it with a close call.

Calls Used:

`Byte_Stream_AM.Open_by_name`	Opens a device given its pathname, creating an opened device object.
`Byte_Stream_AM.Ops.Close`	Closes an opened device.

This example uses the `Open_by_name` call to open a device for reading:

```
      source_opened_device:  Device_Defs.opened_device;

    source_opened_device :=
        Byte_Stream_AM.Open_by_name(
            name         => name,
            input_output => Device_Defs.input,
            allow        => Device_Defs.readers);
```

In the example the open call returns an opened device to the variable `source_opened_device`. The `input_output` parameter specifies the type of I/O for the opened device; the type of I/O is set to `input` for reading. The `allow` parameter specifies how other callers can use the opened device, while you have it open; the allow mode is set to `readers` indicating that other callers can read from the device.

When you close an opened device, you remove the connection between your program and the device.

```
        Byte_Stream_AM.Ops.Close(
            source_opened_device);
```

IV-2.2 Reading and Writing Bytes

Once you create an opened device, you can read and write bytes using an I/O access method.

Calls Used:

`Byte_Stream_AM.Ops.Read`
Reads bytes from an opened device.

`Byte_Stream_AM.Ops.Write`
Writes bytes to an opened device.

A simple procedure might:

1. Declare a fixed-size buffer to hold the bytes read
2. Declare an opened input device and an opened output device
3. Open the devices.

You can declare a simple 4Kbyte fixed-size buffer as follows:

```
BUFSIZE:     constant System.ordinal := 4_096;
buffer:      array(1 .. BUFSIZE) of
             System.byte_ordinal;
bytes_read:  System.ordinal;
```

You can request the number of bytes to be read into the buffer using the opened input device (in this case, the opened device `source_opened_device`). A byte count is returned.

```
loop
  bytes_read := Byte_Stream_AM.Ops.Read(
      source_opened_device,
      buffer'address,
      BUFSIZE);
  Byte_Stream_AM.Ops.Write(
      dest_opened_device,
      buffer'address,
      bytes_read);
end loop;
```

The `Write` call writes the data from the buffer to the opened output device (in this case, the opened device `dest_opened_device`).

IV-2.3 Handling End-of-File

The `Devic_Defs.end_of_file` exception is raised when a read call attempts to read past the end of an input stream.

Calls Used:

`Byte_Stream_AM.Ops.Close`
Closes the caller's opened device.

`Byte_Stream_AM.Ops.At_end_of_file`
Checks whether an opened device is at EOF.

You can monitor the end-of-file condition using an exception handler. For example, after reading records inside a loop in the procedure `Output_bytes`, a `Close` call is made when end-of-file is detected:

```
exception
  when Device_Defs.end_of_file =>
    Byte_Stream_AM.Ops.Close(
        source_opened_device);
```

An alternative way to check for end-of-file is to use an `At_end_of_file` call. This call returns a boolean of `true`, if the opened device indicates an end-of-file condition. The next `Read` call on this opened device will raise `Device_Defs.end_of_file`.

IV-2.4 Using Default I/O Connections

Default I/O connections for standard input and standard output exist in a process's global variables, the *process global entries.*

Calls Used:

`Process_Mgt.Get_process_globals_entry`
Retrieves the process globals entry.

You can retrieve a process's standard output with the `Get_process_globals_entry` call. Here is a complete listing of the `Hello_OS_ex` example which uses this technique:

```
with Byte_Stream_AM,
     Device_Defs,
     Process_Mgt,
     Process_Mgt_Types,
     System;

procedure Hello_OS_ex is
  --
  -- Function:
  --   Write "Hello, world!" on a separate line to the
  --   standard output, using OS packages.

  hello:   constant string := "Hello, world!" & ASCII.LF;
  stdout:  constant Device_Defs.opened_device :=
      Process_Mgt.Get_process_globals_entry(
      Process_Mgt_Types.standard_output);
begin
  Byte_Stream_AM.Ops.Write(
      opened_dev => stdout,
      buffer_VA  => hello(1)'address,
      length     => System.ordinal(hello'length));
end Hello_OS_ex;
```

Normally, your standard output is directed to your terminal. You can redirect standard output to another opened device by setting the `Process_Mgt_Types.standard_output` process globals entry. See the `Process_Globals_Support_Ex.Set_standard_output` example call in Appendix X-A.

A simple utility that takes input from a device and directs it to standard output is shown below. It uses most of the programming techniques previously discussed.

Calls Used:

`Byte_Stream_AM.Open_by_name`
: Opens a device by creating an opened device for the caller.

`Byte_Stream_AM.Ops.Close`
: Closes an opened device for access.

```
  procedure Output_bytes_ex(
      name:  System_Defs.text)
        -- Input device to read.
    --
    -- Function:
    --   Opens the named input device and
    --   copies bytes from it to the caller's
    --   standard output, until end-of-file.
  is
    source_opened_device:  Device_Defs.opened_device;
    dest_opened_device:    Device_Defs.opened_device;
    function Opened_device_from_untyped is new
        Unchecked_conversion(
            source => System.untyped_word,
            target => Device_Defs.opened_device);
    BUFSIZE:     constant System.ordinal := 4_096;
    buffer:      array(1 .. BUFSIZE) of
                 System.byte_ordinal;
    bytes_read:  System.ordinal;
begin
  source_opened_device :=
      Byte_Stream_AM.Open_by_name(
          name         => name,
          input_output => Device_Defs.input,
          allow        => Device_Defs.readers);
  dest_opened_device := Opened_device_from_untyped(
      Process_Mgt.Get_process_globals_entry(
          Process_Mgt_Types.standard_output));

    loop
      bytes_read := Byte_Stream_AM.Ops.Read(
          source_opened_device,
          buffer'address,
          BUFSIZE);
      Byte_Stream_AM.Ops.Write(
          dest_opened_device,
          buffer'address,
          bytes_read);
    end loop;
  exception
    when Device_Defs.end_of_file =>
      Byte_Stream_AM.Ops.Close(
          source_opened_device);
end Output_bytes_ex;
```

IV-2.5 Positioning Within a Byte Stream

Prior to reading or writing bytes, the byte pointer for an open device can be set to any byte position in a byte stream.

Calls Used:

`Byte_Stream_AM.Ops.Set_position`
: Moves the byte pointer for an opened device to a specified byte offset in the byte stream.

To set the byte pointer, indicate the opened device and then specify a signed byte offset from the starting byte. Positioning can occur:

- From the beginning of the device
- From the current pointer position
- From the end of the file (last byte plus one).

If the byte pointer is set beyond the device's end-of-file, a subsequent write to the device causes the device to increase to the size indicated by the new byte pointer position.

IV-2.6 Reading and Inserting Records Sequentially

Reading and writing *records* sequentially is very similar to reading and writing *bytes*. Record I/O accesses a record as a unit, while byte stream I/O accesses a byte as a unit.

Calls Used:

`Record_AM.Open`
Opens a device given its pathname.

`Record_AM.Ops.Close`
Closes an opened device.

`Record_AM.Ops.Read`
Reads a record.

`Record_AM.Ops.Insert`
Inserts a record.

The following procedure, `Output_records_ex`, is a general-purpose utility that reads records sequentially from a file into a buffer, and then outputs them sequentially using standard output. It does the same thing with records that the `Output_bytes_ex` procedure does with bytes:

- Declares source and destination opened devices
- Accepts record input from the specified device
- Outputs the record using standard output.

The example `Output_record_ex` sets up a variable length record buffer to read in records of any length. The following declaration declares and allocates a variable length buffer:

```
buffer_size:  System.ordinal := 256;
buffer_AD:  System.untyped_word :=
    Object_Mgt.Allocate(buffer_size/4);
  -- 64 words (256 bytes) is the initial buffer
  -- size.  Buffer size is increased as needed.
  -- The buffer is in a separate object for easy
  -- resizing.
bytes_read:      System.ordinal := 0;
  -- If record requires multiple "Read" calls,
  -- then this variable tracks bytes read so far.
```

A `read_status_VA` variable is declared to contain status information about the read call. A `read_position` variable contains the record position for the next read.

```
read_status_VA:  Record_AM.operation_status_VA :=
    new Record_AM.operation_status_record;
read_position:   Record_AM.position_modifier :=
    Record_AM.next;
```

The `Read` call reads records from an input device into a variable length buffer. The buffer is resized when it is too small to handle a record.

```
loop
  begin
    bytes_read := bytes_read +
        Record_AM.Ops.Read(
            source_opened_device,
            read_position,
            System.address'(
                bytes_read,
                buffer_AD),
            buffer_size - bytes_read,
            status => read_status_VA);

    -- When control reaches this point, "Read"
    -- succeeded without a length error and
    -- this loop can be exited.
    EXIT;
```

If the buffer is too small, a recovery block resizes the buffer by handling the `Device_Defs.length_error` exception. The record buffer is resized by checking the length of the buffer for the `Read` call using `read_status_VA.rec_length`.

```
  exception
    when Device_Defs.length_error =>
      buffer_size := read_status_VA.rec_length;
      if buffer_size =
          Record_AM.unknown_length then
        buffer_size := 2 * 4 *
            Object_Mgt.Get_object_size(buffer_AD);
          -- Double the buffer size if an exact
          -- new size is not available.
      end if;
      Object_Mgt.Resize(
          buffer_AD,
          (buffer_size+3)/4);
        -- May make object even bigger than
        -- requested, but that's OK.
      read_position := Record_AM.rest_of_current;
  end;
```

The buffer is resized, the `read_position` is set to read the next record, and the `Read` succeeds without a length error. When the record is read, the record can be inserted to the standard output.

```
Record_AM.Ops.Insert(
    dest_opened_device,
    System.address'(0, buffer_AD),
    bytes_read);
```

See Chapter IV-9 for more information about record I/O operations.

MANAGING STREAM FILES 3

Contents

This chapter shows how to create, copy, and delete stream files. To read and write stream files, use the access method techniques that Chapter IV-2 describes.

Packages Used:

`Simple_File_Admin`	Manages stream files.
`Byte_Stream_AM`	Provides device-independent I/O using streams of bytes.
`Record_AM`	Provides device-independent I/O for one-record-at-a-time access. Contains the `Record_AM.Ops` and `Record_AM.Keyed_Ops` packages.

A *stream file* is a file consisting of a contiguous stream of bytes. Figure IV-3-1 shows a stream file being opened using byte stream I/O.

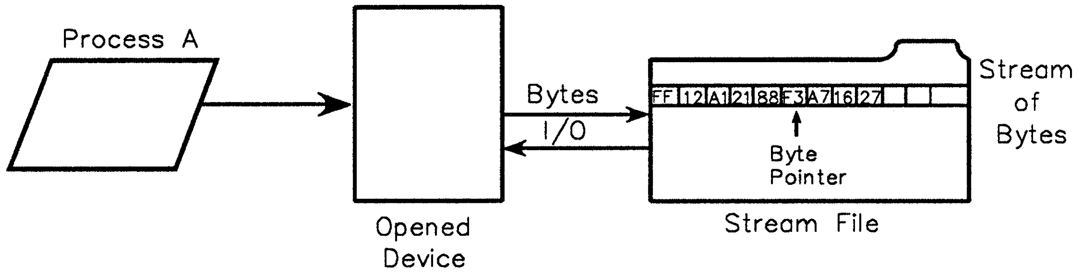


Figure IV-3-1. Stream File Being Opened for Access

IV-3.1 Concepts

For creating, copying, and destroying stream files, you use the `Simple_File_Admin` package. For basic I/O with stream files, you use the `Byte_Stream_AM` and `Record_AM` packages.

IV-3.1.1 What Is a Stream File?

There are two different classes of files: stream files and structured files. The main difference between a stream file and a structured file is that stream files cannot be indexed. But stream files can contain records, and you can use record I/O with stream files. For indexed files, you must use a structured file organization.

You create and destroy stream files using the calls in the `Simple_File_Admin` package. You can use these calls inside transactions, but the file is not created or destroyed until the calling transaction commits.

The system automatically handles disk space allocations for stream files (including file expansion and volume set selection). You can specify allocations for stream files with calls in the `Simple_File_Admin` package. This lets you control how much disk space to allocate for stream files. Stream files are allocated in this manner:

- Permanent stream files with pathnames are created on the same volume set as the directory or subdirectory indicated by the pathname.
- *Temporary files* without pathnames are created on the default system volume set, unless specified otherwise.

These type rights affect stream files:

read rights	Required to open a stream file for input.
write rights	Required to open a stream file for output.
control rights	Required to allocate disk space for a file and to otherwise manipulate a file.

IV-3.1.2 Using Access Methods with Stream Files

You can access stream files using either byte stream or record I/O. In general, you should use byte stream I/O to access a stream file, although you can use record I/O to access records in stream files. Mixing I/O methods when accessing stream files can cause problems. See the `Simple_File_Admin` package in the *BiiN™/OS Reference Manual* for guidelines.

Stream files are not transaction-oriented; transactions and transaction operations don't affect I/O operations or other operations on stream files.

IV-3.1.2.1 Byte Stream I/O

You perform byte stream I/O on stream files using the calls in the `Byte_Stream_AM` package. These calls let you open a stream file for input or output, read bytes from a stream file, or write bytes to a stream file. You can position a byte pointer to any byte in the file prior to reading or writing.

IV-3.1.2.2 Record I/O

You perform record I/O on stream files using the calls in the `Record_AM` package. These calls let you open a stream file for record input or output, read records from a stream file, or write records to a stream file.

During a `Record_AM` read operation, each record is read up to the next termination character. During a write operation, a termination character is appended to delimit the boundaries between records. The termination character (`term_char`) is set up when the stream file is created. It defaults to an ASCII line feed.

You can use stream files for pages of text. Each line of text is a record. A record containing only an ASCII form feed (that is, a page mark) is the end of a page of text. Page marks can be created using a `Record_AM.Ops.New_page` call.

Figure IV-3-2 shows possible line formats. The angle brackets <> indicate a required element. The braces { } indicate a repeating element.

empty-line := *<preceding-delimiter> <terminator>*

data-line := *<preceding-delimiter>* { *<data character>* } *<terminator or EOF>*

page-mark := *<preceding-delimiter> <form-feed> <terminator>*

Figure IV-3-2. Line Formats for Stream Files

preceding-delimiter	is either a terminator or the beginning of the file and is not part of the line that follows it.
data-character	is any character except a terminator.
terminator	is the file's specified termination character.

The definition of the page mark excludes its termination by the end-of-file marker. A form feed character written without the trailing terminator is treated by record I/O as a data character. To be compatible with stream files that use byte stream I/O, the data line is allowed to be terminated by the end-of-file marker.

Other record-oriented characteristics of stream files include:

- Each record in a stream file has a unique, system-maintained *record ID.*
- A termination character cannot be part of a record.
- Records cannot be indexed.
- Records cannot be deleted.
- Records can be read sequentially or randomly based on the physical ordering of the records in the file.
- Records can only be inserted at the end of the file.
- Records can be updated as long as the size of the record doesn't change.
- Stream files don't participate in transactions, don't support record locking, and don't support file logging.

IV-3.1.3 Temporary Files

Temporary files are stream files that are not named when created. They exist for the duration of the current job unless they are explicitly saved and named.

You use a special `Simple_File_Admin` call, `Create_unnamed_file`, to create a temporary file. If you intend to create a temporary file and then save it later, it's best to indicate its volume set when the file is first created. That way, you can make sure that the pathname you assign the file (when you save it) is on the correct volume set.

- You can explicitly destroy a temporary file by making a `Destroy_file` call.
- You can name a temporary file by making a `Save_unnamed_file` call.

- Once a file is named, it has a corresponding directory entry. By deleting this entry with a `Directory_Mgt.Delete` call, you get rid of the passive version of the file object. The file goes away when the last AD that references it goes away.

IV-3.2 Techniques

This section provides the techniques for creating, copying, and destroying stream files. After reading this section you will be able to:

- Create a stream file
- Copy a stream file
- Empty a stream file
- Delete a stream file
- Create a temporary stream file.

The examples used in the following sections are parts of one procedure: `Stream_file_ex`. See Appendix X-A for the complete listing.

IV-3.2.1 Creating a Stream File

When creating a stream file, you only need to specify the file's pathname to the call that creates your file. The filing system:

- Creates a stream file using the file's pathname
- Stores an entry in the new file's directory
- Returns an AD for the file with all type rights.

Calls Used:

`Simple_File_Admin.Create_file`
Creates a stream file.

You specify the file's pathname in the `name` parameter, which must be a `System_Defs.text` type. The directory where you want to create the file must exist and the filename must be unique, or you will get an exception. To create a file use:

```
Text_Mgt.Set(filename, "my_file_1");
file1 := Simple_File_Admin.Create_file(filename);
  -- Creates a stream file in the current
  -- directory.
```

The rest of the parameters are defaulted. You can specify the number of bytes to preallocate for disk storage with the `bytes_to_preallocate` parameter. The default is no byte preallocation. You can specify an alternate termination character with the `term_char` parameter. The default termination character is an ASCII line feed.

IV-3.2.2 Copying a Stream File

Once you create a file, you can copy its contents to another file.

Calls Used:

`Simple_File_Admin.Copy_file`
Copies the contents of one file to another.

You copy a file by specifying the `source_file` and `target_file` parameters. For example:

```
Text_Mgt.Set(filename, "my_file_2");
file2 := Simple_File_Admin.Create_file(filename);
Simple_File_Admin.Copy_file(source_file => file1,
                            target_file => file2);
  -- Creates a second file in the current directory,
  -- and then copies the contents of the first file
  -- to the second.
```

You have the option of deallocating unused pages in the second file by setting a `shrink` boolean to true.

IV-3.2.3 Emptying a Stream File

You can empty a stream file with one call.

Calls Used:

`Simple_File_Admin.Empty_file`
Empties a file of its contents without deallocating it.

For example, to empty the contents of a file use:

```
Simple_File_Admin.Empty_file(file1);
  -- Empties the first file.
```

If there is more than one caller using the file, the file is not emptied until all callers have closed the file. This default setting can be changed by setting a `notify_if_busy` boolean to true. If this parameter is true, `Device_Defs.device_in_use` is raised when a file is in use, and the file is not emptied.

`Empty_file` is slightly more complex when done inside a transaction. See the `Simple_File_Admin` package in the *BiiN™/OS Reference Manual.*

IV-3.2.4 Deleting a Stream File

You can delete a file by simply deleting its directory entry or by destroying the file.

Calls Used:

`Directory_Mgt.Delete`
: Deletes a directory entry. This may also destroy the referenced object's passive version.

`Simple_File_Admin.Destroy_file`
: Deallocates all parts of the file, but not its directory entry.

`Process_Mgt.Get_process_globals_entry`
: Gets the default directory for the stream file.

To delete a file's directory entry you call `Delete`, giving the file's pathname in the `name` parameter. For example, to delete a file in the current directory use:

```
Text_Mgt.Set(filename, "my_file_2");
Directory_Mgt.Delete(filename);
  -- The second file's pathname is deleted.  The
  -- second file is destroyed when the last
  -- reference to it goes away.
```

You can also delete a file with the `Destroy_file` call. You need to specify the file's AD with the `file` parameter. This example destroys a *temporary file*:

```
Simple_File_Admin.Destroy_file(file3);
-- Destroys the temporary file before its job
-- terminates.  If it is not destroyed or saved,
-- it goes away when the job terminates.
```

This call only destroys the file, not the file's directory entry.

If there is more than one caller using the open file, the file is not destroyed until all callers have closed the file. The `notify_if_busy` parameter can change this. If true, this parameter causes the `Device_Defs.device_in_use` exception to be raised when the file is in use; the file is not destroyed when this exception is raised.

Do not use a `Delete` call to delete a temporary stream file; use the `Destroy_file` call. If you use the `Destroy_file` call in a transaction, the file is not destroyed if the transaction aborts.

IV-3.2.5 Creating Temporary Files

You can create temporary stream files for intermediate data storage (for example, a work file). Temporary files exist for the duration of the current job, and are automatically destroyed when the job terminates.

Calls Used:

`Simple_File_Admin.Create_unnamed_file`
: Creates a temporary stream file.

`Passive_Store_Mgt.Home_volume_set`
: Returns an object's home volume set.

`Process_Mgt.Get_process_globals_entry`
: Gets a process globals entry.

To create a temporary file, you must specify its volume set in the `volume_set` parameter. For example:

```
file2 := Simple_file_Admin.Create_unnamed_file(
    Passive_Store_Mgt.Home_volume_set(
        Process_Mgt.Get_process_globals_entry(
            Process_Mgt_Types.current_dir)));
  -- Creates a temporary file in the current
  -- directory using the current directory's
  -- volume set.
```

This example retrieves the *home volume set* for the current directory and uses it as the volume set for the file. You can name a temporary file and save it as a permanent file using `Simple_File_Admin.Save_unnamed_file`.

IV-3.3 Summary

- Use stream files to read and write bytes.
- You cannot index stream files.
- Stream files don't participate in transactions.
- Temporary files exist only for the duration of the current job unless you explicitly name and save them.

USING WINDOWS 4

Contents

This chapter discusses the use of windows on character terminals, including terminal manager support for windows and I/O access methods.

Packages Used:

`Character_Terminal_Mgt`	Manages character terminals.
`Terminal_Defs`	Defines constants, types, and exceptions used by the terminal service packages.
`Window_Services`	Provides windows on character and graphics terminals, including pull-down menus.

A window is a portion of the terminal screen through which a user interacts with an application program. In general, the screen can contain multiple windows; the windows do not overlap. The user controls the size, location, and visibility of windows, but the application program controls the contents. The application uses one of three access methods to do I/O to a window. Coordinate systems describe the location of windows with respect to the screen and the location of specific points within a window. A user can move a window, resize it, pan it vertically or horizontally, and request that it be closed. Windows can have pop-up menus. Terminal managers provide support for windows and I/O access methods.

IV-4.1 Concepts

The following terms are used throughout this chapter:

physical terminal	The physical terminal is a video display device with a keyboard.
user	The human being interacting with the system through a terminal.
application	The application program being run by the user. Typical examples are text editors and spread-sheet programs. The term "application" also covers any additional software that lies between the application and the operating system.
virtual terminal	A device that, to an application, is indistinguishable from a physical terminal. A virtual terminal provides a screen-like drawing space for the output of characters or graphics, and a keyboard and mouse for input.
frame buffer	The drawing space of a virtual terminal. An application writes to the frame buffer associated with a virtual terminal. The frame buffer is visible to a user through a rectangular screen area; that is, through a *window*. The part of the frame buffer that is visible through a window is called the *view*.
window	The rectangular screen area where the view appears. Many windows may be visible on the screen at the same time.
view	The visible part of the frame buffer. The view may not be larger than the frame buffer.
input focus	There may be many windows on a terminal, but there is only one keyboard (and mouse). The input focus is that virtual terminal to which keyboard and mouse input are connected at a given time.

Figure IV-4-1 shows the relationships among views, windows, frame buffers, and a physical terminal.

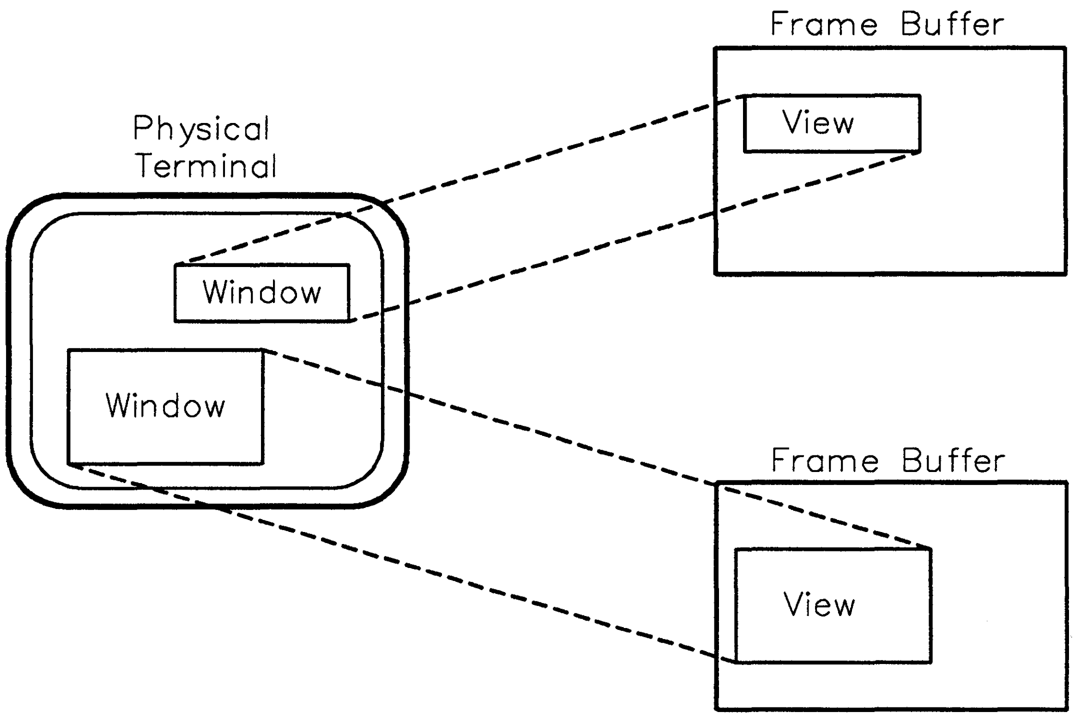


Figure IV-4-1. Windows Displayed on a Physical Terminal

IV-4.1.1 Terminals and Windows

The current system release supports character terminals. Graphics terminals will be supported in a future release.

Character terminals have some subset of the features specified in the ANSI X3.64 standard: character insertion and deletion, line insertion and deletion, cursor positioning, scrolling, and so forth. The DEC VT-100[1] is a typical character terminal.

The screen of a character terminal is divided into a Cartesian grid of fixed-width, fixed-height character cells each of which contains a displayable character. Each cell is identified by *(x,y)* coordinates, starting with (1,1) in the upper-left hand corner. The *x* coordinate increases to the right, and the *y* coordinate increases downwards. Graphics terminals have all the capabilities of character terminals and others as well; for example, line drawing, line and point attributes, and raster primitives. The graphics terminals to be supported are (TBD).

The screen of a graphics terminal is divided into a Cartesian grid of pixels, each of which is identified by *(x,y)* coordinates in the same fashion as character cells on a character terminal.

On a character terminal, a window consists of a rectangular grid of character cells; on a graphics terminal, it consists of a rectangular grid of pixels. The size of the window is initially suggested by the application, and may be modified by the user. Note the following:

[1]DEC VT-100 is a trademark of Digital Equipment Corp.

- The system constrains windows on character terminals to be as wide as the screen.
- Character terminal windows do not overlap. Graphics terminal windows can overlap.

IV-4.1.2 Accessing Windows

An application does I/O to a physical terminal through the frame buffer and keyboard associated with the application's virtual terminal, but we will talk about I/O to windows since they provide the visible evidence of I/O activity within a frame buffer.

An application creates and manipulates windows through the `Window_Services` package, which provides a procedural interface for creating and destroying windows, manipulating the characteristics of windows, and building and installing menus on windows.

Once a character terminal window has been set up, three *access methods* are available for performing I/O in the window:

- `Character_Display_AM`
- `Byte_Stream_AM`
- `Record_AM`.

`Character_Display_AM` provides operations for doing character I/O to windows, treating the window as a two-dimensional grid of character cells. Operations are provided for input, output, insertion and deletion of lines and characters, cursor movement, etc.

`Character_Display_AM` is the primary access method for character terminal windows, but an application can also use `Byte_Stream_AM` and `Record_AM`. Under these access methods, I/O to a window is treated as a stream of characters. Operations are provided for input, output, etc. Several of these operations have slightly different semantics when they are used to access windows.

An application *mixes access methods* when it calls the `Open` function of one access method and then uses the returned opened device to call operations provided by other access methods. An example would be calling `Byte_Stream.Ops.Open` and then using the returned opened device object to call `Character_Display_AM.Ops.Insert_line`. The extent to which access methods can be mixed is device-dependent.

IV-4.1.3 Window Coordinates

`Window_Services` treats terminals and virtual terminals as two-dimensional display surfaces, and provides operations that refer to positions on the surfaces. A position is specified by a pair of coordinate values within a *coordinate system.*

An application using `Window_Services` may need to access either character positions or pixel positions. Accordingly, `Window_Services` provides both a character-oriented coordinate system and a pixel-oriented coordinate system. Each operation that takes parameters to describe a position also includes a boolean parameter to select which coordinate system is used to interpret the values.

Figure IV-4-2 shows the position of a window relative to the screen as measured in the character-oriented and pixel-oriented coordinate systems. In each case, the horizontal component of the position is given first followed by the vertical component. The upper left corner of the window is in the 4th character column of the screen (counting from the left) and the 5th character row (counting from the top). Thus the character-oriented coordinate system position is (4,5). The same corner is in pixel column number 10 (1 is at the left), and pixel row number 40 (1 is at the top). Thus the pixel-oriented coordinate system position is (10,40).

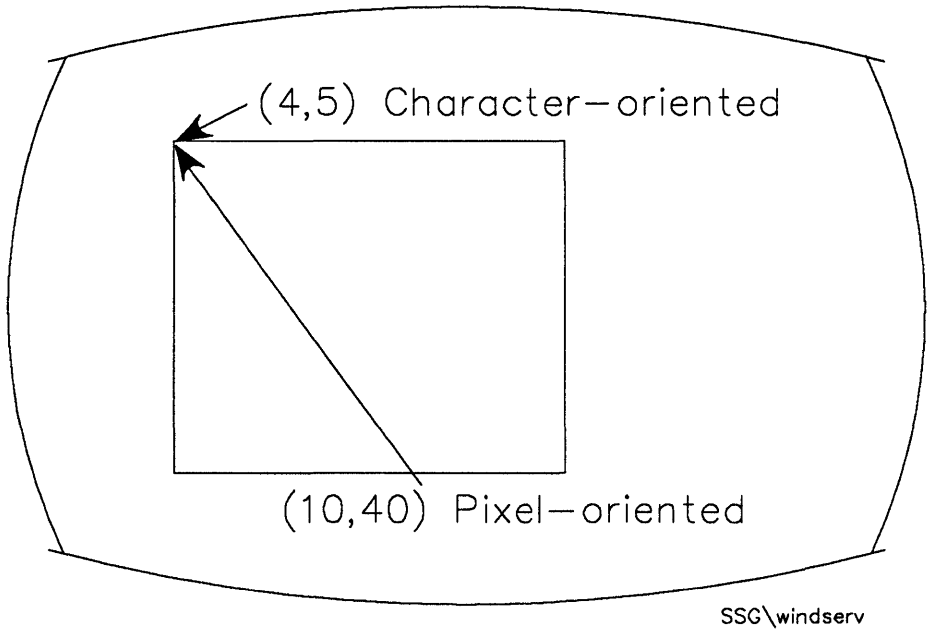


Figure IV-4-2. `Window_Services` **Coordinate Systems**

IV-4.1.4 Terminal Attributes

There are a number of attributes that affect the I/O done to a window. Terminal attributes are defined on a per-terminal basis and affect all the windows on the terminal. Table IV-4-1 lists these attributes. The table provides suggested defaults; the actual defaults are device dependent.

See the `Terminal_Defs` package for more detailed information.

Table IV-4-1. Terminal Attributes

Attribute	Default Setting
erase	BS (<Ctrl-H>)
cancel	NAK (<Ctrl-U>)
newline	LF (<Ctrl-J>)
EOL	NUL (<Ctrl-@>)
EOF	EOT (<Ctrl-D>)
literal_next	SYN (<Ctrl-V>)
redraw_input_line	DC2 (<Ctrl-R>)
suspend	SUB (<Ctrl-Z>)
interrupt	ETX (<Ctrl-C>)
termination	DEL (<Ctrl-?>)
debug	STX (<Ctrl-B>)

Table IV-4-1: Terminal Attributes (cont.)	
Attribute	**Default Setting**
stop	DC3 (<Ctrl-S>)
start	DC1 (<Ctrl-Q>)
do_flow_control_out	wait_for_any
do_flow_control_in	false
ignore_break	false
do_parity_checking	true
bad_char_handling	discard_bad_char
visual_bell	false
baud_rate	`b_9600`
physical_char_detail	`(none,one,eight)`

IV-4.1.5 The Input Model

An application running in a window obtains user input by calling the `Read` procedure in one of the four access methods mentioned earlier. The `Read` procedure returns one or more input events of a certain type. This section characterizes the various types of input and presents the general input model.

If a keyboard were the only input device of a terminal, and a terminal did not provide multiple windows, the input model would be simple. To obtain *keyboard input*, an application would call the appropriate `Read` procedure, which would (optionally) block until the user typed something. If the user typed while no application was reading, the data would be stored in an input event queue until the application requested it.

This simple input model is inadequate because there are other types of input:

Mouse input
: A terminal may have a mouse in addition to a keyboard. The mouse introduces a new type of *input event*. (Input events should not be confused with the events defined by `Event_Mgt`. As used here, the term event describes an action performed by the user.) A mouse input event contains information about the position of the mouse in addition to the state of its buttons. Furthermore, an application that wishes to read input from both the mouse and keyboard must be able to distinguish between keyboard input events and mouse input events.

Window-related input
: Imagine a *clock* application that does nothing but draw a clock face in the frame buffer, updating it each second. According to the material presented so far, the user sees a window that may be much smaller than the frame buffer itself, in which case only a small part of the clock face is visible, and the application is not particularly useful. The application could improve its user interface if it could draw a smaller clock face that would fit within the view of the frame buffer. To do this, the application would need a way to find out about changes in the origin and size of the view.

: `Window_Services` makes that possible by defining a number of window-related input events that can inform the application about events that occur to the window; the application can then take whatever action is appropriate.

User-defined input
: The user may define input events.

The system provides an enumeration type, `Terminal_Defs.input_enum`, that denotes all four types of input events (the literals from `menu_item_picked` through `scroll_requested` denote window-related input):

`keyboard`	The user typed something on the keyboard.
`mouse`	The user moved the mouse, or its buttons, or both.
`menu_item_picked`	The user picked an item from a menu.
`focus_changed`	The user changed the input focus.
`overlap_changed`	The user changed the window's overlap state.
`size_changed`	The user changed the window's size.
`view_changed`	The user panned the view to a new position on the frame buffer.
`position_changed`	The user moved the window to a new location on the screen.
`close_requested`	The user requested that the window be closed.
`scroll_requested`	The user requested that the window be scrolled.
`user_defined`	An application initiated a user-defined input event.

The system provides another type, `Terminal_Defs.input_type_mask`, that denotes a set of these input types. There is always an input type mask in effect for a given virtual terminal. Newly arriving input events are accepted if their type matches one of those in the input type mask for that window; otherwise they are discarded. When a new input event is accepted, it is stored in an input event queue or passed immediately to an application blocked on a read call.

IV-4.1.6 The Output Model

Output to a window is accomplished through output to the window's frame buffer using one of the three access methods. See the *BiiN™/OS Reference Manual.*

IV-4.1.7 Overlapped Windows

As mentioned earlier, the system supports multiple windows on a single physical terminal. The rest of this chapter describes `Window_Services`, the interface for creating, destroying, and controlling windows. But before describing this interface, a more detailed discussion of windows is required.

Recall that an application draws figures or characters on a drawing space called a frame buffer. A rectangular portion of the frame buffer, the view, is mapped to a rectangular area on the screen. This rectangular area is called the window and is the same size as the view. The *origin* of the view is measured relative to the frame buffer, and the *position* of the window is measured relative to the screen.

Figure IV-4-3 shows these relationships.

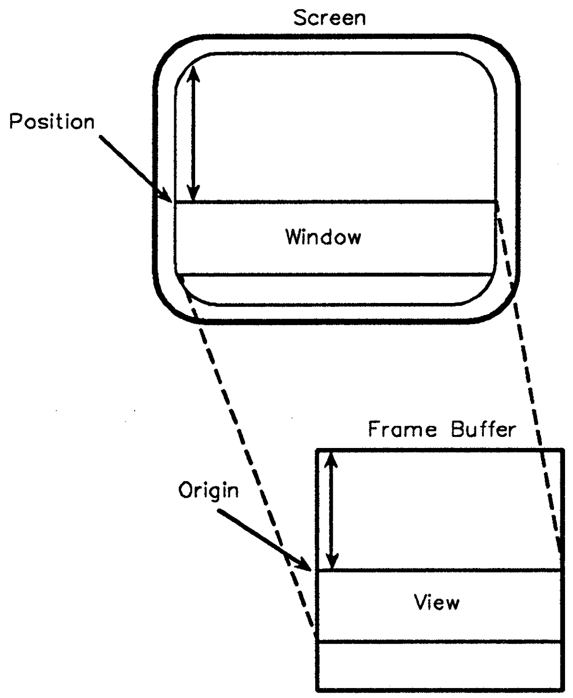


Figure IV-4-3. Relationship Between Window and View

Figure IV-4-3 shows a single window, but in general it is possible to display multiple windows on a single terminal screen and to have the windows overlap like sheets of paper on a desktop. (Support for overlap is device dependent, with each device manager defining the level of support it provides; in particular, overlap is not supported on character terminals.)

Overlapped windows are ordered in a *front-to-back* relationship that determines what is displayed in areas of the screen that are covered by more than one window. Some windows may be partially or wholly obscured by windows that are logically in front of them.

Figure IV-4-4 shows a possible layout for three windows whose front-to-back ordering is (W1, W2, W3). W2 is partially obscured by W1, but W3 is unobscured since it does not overlap with W1 or W2.

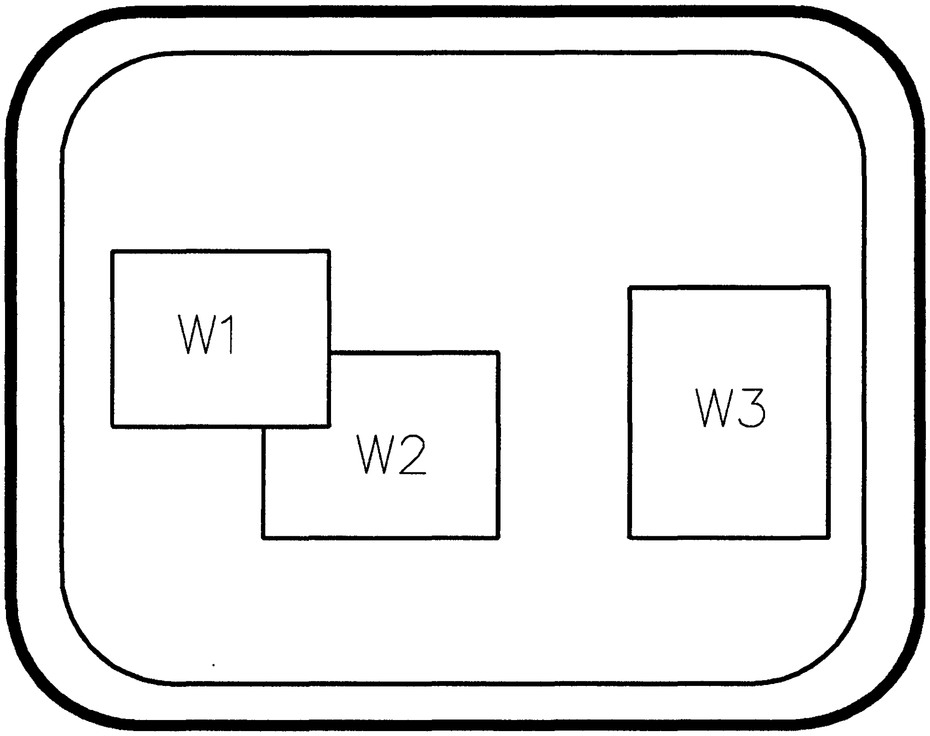


Figure IV-4-4. Example of Overlapped Windows

IV-4.1.8 Some Key Points

- Windows are independent of one another. The abstraction presented to an application is that it can draw images on a frame buffer that is completely independent from all other frame buffers. The application can treat the frame buffer as if it were an entire screen. Since frame buffers are independent, so are windows.
- An application cannot affect the screen outside its window. All drawing operations are *clipped* to the boundaries of the frame buffer. Only the portion of the frame buffer inside the view is visible in the window. Screen areas outside the window belong to other applications or are unused.
- A frame buffer's size is fixed when it is created, and never changes. The corresponding view cannot be enlarged beyond the edges of the frame buffer. Thus a window can never be larger than its frame buffer.
- The user controls the layout, size, and front-to-back ordering of windows, but an application has full control of a window's contents. This includes drawing on the frame buffer and panning the view over the frame buffer. Everything else is controlled by the system. This includes clipping, maintaining the coordinate system for the frame buffer, and changing the layout, size, and visibility of windows. The system software that interacts with the user to perform these tasks is called:
  - The *character terminal user agent* for character terminals.
  - The *graphics terminal user agent* for graphics terminals.

IV-4.1.9 Resizing a Window

If the user changes the size of the window, then the view's size must also change so that a different portion of the frame buffer becomes visible in the window. But what is the position of the new view in the frame buffer? As shown in Figure IV-4-5 there are several possibilities.

The figure shows a frame buffer and view before and after the view size is doubled. In (b), the left top corner of the new view is at the same position as the left top corner of the old view. In (c), the middle of the new view is at the same position as the middle of the old view. In either case, the application had no opportunity to control the position of the view and hence the window's contents, which means that the user's ability to control layout is in conflict with the application's responsibility to control contents.

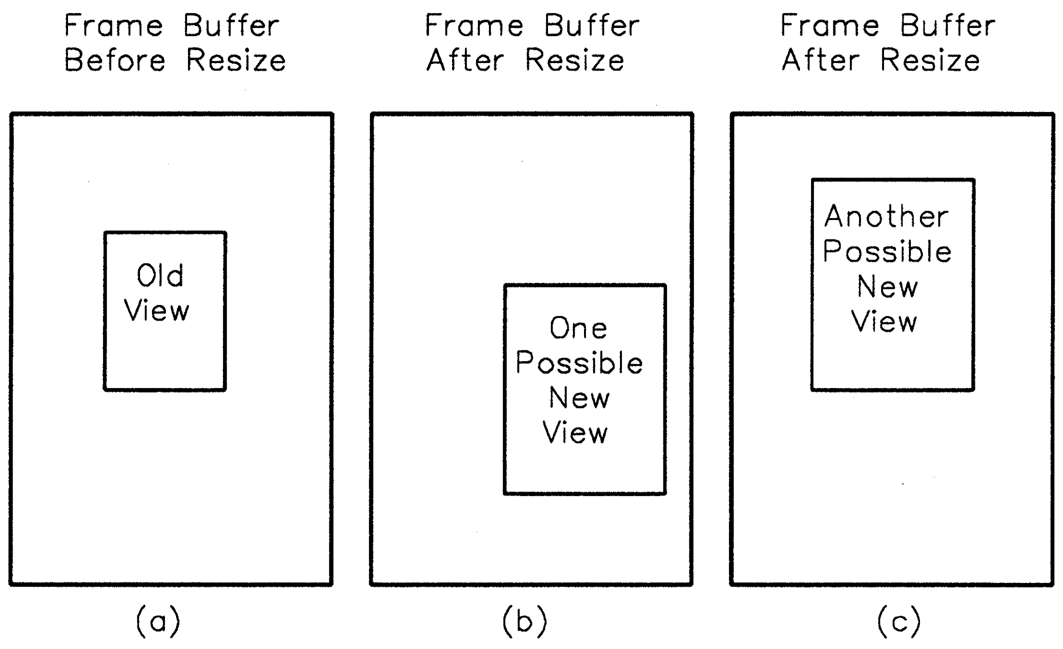


Figure IV-4-5. Example Showing Two Possible Resize Rules

To address this problem, `Window_Services` allows an application to specify a *resize rule* that describes the desired effect of a resize operation on the view origin. Whenever possible, this rule is used to resize a window. Thus the application indicates its wishes before the resize operation occurs.

A resize rule identifies a *stable point* within the frame buffer by describing its position relative to the view. The stable point occupies the same position relative to the view's borders after the resize as before the resize. For example, if the resize rule is *LT* the left top corner is the stable point. The frame buffer contents that appeared in the left top corner of the window before the resize still appear in the left top corner after the resize. Figure IV-4-6 illustrates this by showing a frame buffer and view before and after a resize that enlarges the view. The resize rule is *LT*. After the resize operation, the same contents still appear in the left top corner of the window.

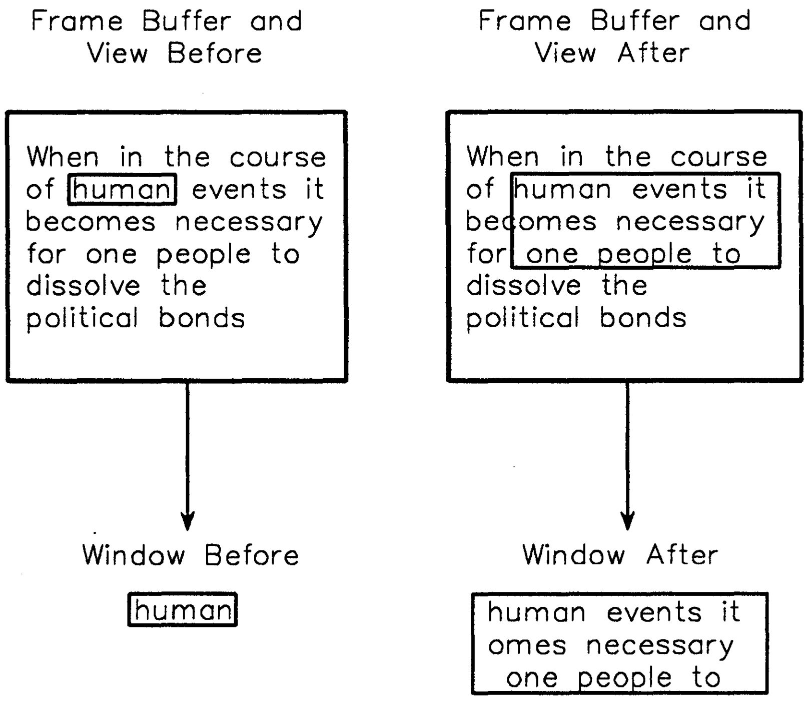

Frame Buffer and View Before
When in the course of human events it becomes necessary for one people to dissolve the political bonds
Window Before
human
Frame Buffer and View After
When in the course of human events it becomes necessary for one people to dissolve the political bonds
Window After
human events it
omes necessary
one people to

Figure IV-4-6. Left Top Resize Rule

Resize rules are two-character strings. The first character describes the stable point's horizontal position, and the second character describes its vertical position. Legal resize rules allow *L*, *M*, and *R* for the first character, indicating *left*, *middle*, and *right*, and *T*, *M*, and *B* for the second character, indicating *top*, *middle*, and *bottom*.

Two other characters are allowed in either position of a resize rule:

F	The horizontal or vertical dimension is *fixed*. Any point in the contents is stable relative to the left and right borders or top and bottom borders. The window cannot be resized in the corresponding dimension.
C	The position of the window's *contents* remains as specified, and its borders move as required to perform the resize.

An attempt to move the window partially off the screen moves the contents as required and adjusts borders as necessary. For example, Figure IV-4-7 shows a screen and a window with the *CC* resize rule. An attempt to move the window partially off the screen to the right would move the window's contents the requested distance and clip the window and view on the right to fit.

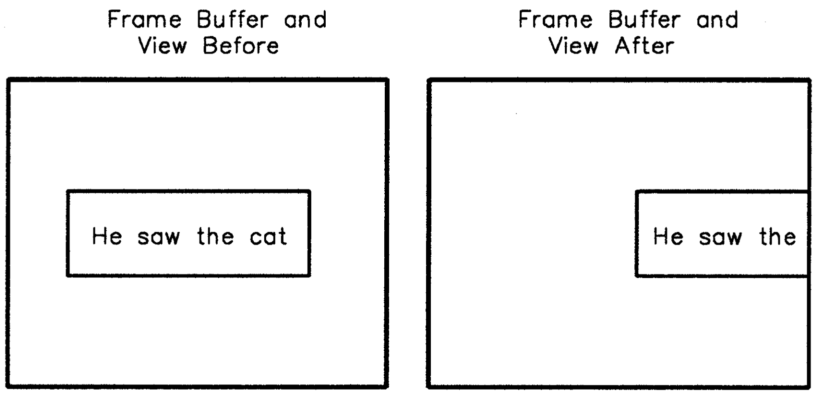


Figure IV-4-7. Contents Resize Rule Example

IV-4.1.10 Basic Window Operations

The above describes the basic functions of `Window_Services`; the following is a list of the corresponding calls:

`Get_Window_Services_Attr_ID`	Returns the windowing attribute ID.
`Create_window`	Creates a window on a terminal.
`Destroy_window`	Destroys a window on a terminal.
`Get_terminal`	Returns the terminal containing a window.
`Change_view`	Moves a window's view over its frame buffer to a (new) origin.
`Set_resize_rule`	Sets a window's resize rule.
`Get_resize_rule`	Returns a window's resize rule.
`Get_window_status`	Returns a window's status.
`Insert_input_event`	Inserts an input event into a window's input event queue.

See the `Window_Services` package for more information.

IV-4.1.11 Window Style

An application can control two aspects of a window's style:

Interactions	The application can choose what user-interactions are permitted for a window. These interactions include generating a *close* request, moving the window, re-sizing the window, and panning the window vertically or horizontally.
Appearance	The application can choose whether a window has title and information bars, and what the contents of the bars is. The application can also affect the size and position of sliders within scroll bars.

Note that the appearance of a window may be related to the set of permitted interactions. For example, graphics terminals may add scroll bars to a window and use the scroll bars to specify panning operations. The appearance and attributes of border areas like scroll bars are device specific.

`Window_Services` provides two calls for controlling the elements of a window's style:

- `Get_window_style`
- `Set_window_style`

These calls enable the application to control border areas and window interactions. The operations are fully described in `Window_Services`, but several points are worth emphasizing:

- Some combinations of border areas are not permitted. The exact combinations permitted are defined by each device manager. All devices allow a window with no border areas.
- Illegal combinations of border areas are silently transformed into legal combinations. If an application needs to know what border areas are actually present, it can call the `Get_window_style` operation.
- The contents of a border are maintained even if the border area is not present. For example, the application can supply text for the information line even if the line is not currently present. Then, when the information line is added, the text appears.

IV-4.1.12 Menus and Windows

Pull-down menus can be associated with windows. The number of menus per window and the menus' features are device dependent, but as a minimal level of support, all devices support a menu consisting of a single list of text items that can be selected from the keyboard. This section describes the full level of support.

IV-4.1.12.1 Menu Hierarchy

A pull-down menu is a three part hierarchy:

Menu group	A menu group is a collection of menus (see below). A window can have zero or more menu groups, up to a device-dependent limit.
Menu	A menu is a collection of menu items (see below). A menu group can have zero or more menus, up to a device dependent limit. Each menu has a title.
Menu item	A menu item is text or a bitmap (on devices that support bitmaps). A menu can have zero or more menu items, up to a device dependent limit.

Figure IV-4-8 shows the preferred layout for a typical menu group. The menus in the group appear side by side in a *Menu bar*, which displays the title of each menu. The character terminal user agent defines how menu items are selected.

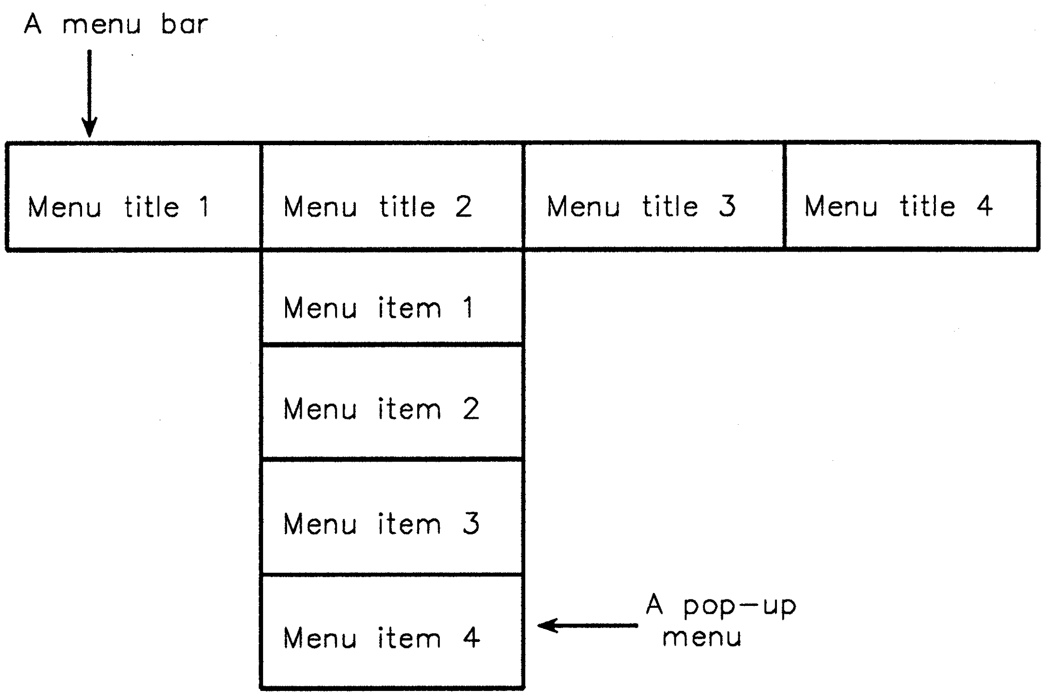


Figure IV-4-8. Menu Bar and Pull-down Menu

Full menu support includes the following additional features. (Some device managers do not support all these features.)

- A menu item can have a check mark in front of it. The meaning of a check mark is application dependent. A typical use is to indicate the most recent item selected from the menu.
- A menu item can have a character associated with it. The character is displayed along with the menu item. The character can be typed to select the item from the menu. Items that do not have a character can only be selected by using the mouse.
- A menu item can be disabled. A disabled menu item cannot be selected. Disabled menu items are displayed in gray.

IV-4.1.12.2 Building and Installing a Menu

An application builds a menu by building the menu's *DDef* description. This is done by using the `Data_Definition_Mgt` package, as desribed in the `Window_Services.Ops` package. The end result is that the application obtains an AD for the menu group.

The application then installs the menu group in a window, supplying the menu group AD and an ID for the menu group. This ID is analogous to the menu and menu item IDs. It is used in subsequent references to the menu group and is included in the input event passed to the application when a menu item is selected. After installing the menu group, the application no longer needs the AD and can discard it.

After installing the menu group, the application can enable and disable menu items, place and remove a check mark in front of items, and even replace the text of a text item.

To disable the user's ability to invoke menus in a menu group, the application removes the menu group from the window.

`Window_Services` provides the following operations for installing menus:

`Install_menu_group`
: Installs a menu group in a window and gives the group an ID.

`Remove_menu_group`
: Removes an installed menu group from a window.

`Menu_group_enable`
: Enables or disables a menu group.

`Menu_item_check`
: Adds or removes a menu item checkmark.

`Menu_item_enable`
: Enables or disables a menu item in an installed menu group.

`Replace_menu_item_text`
: Replaces the text of a menu item.

IV-4.1.13 User Agents

As noted earlier, the user controls the layout of windows; that is, the user controls the size, position, and overlap of windows, as well as which window is the input focus. The system provides this control through a user agent, which is software that interprets some user actions as requests to manipulate windows.

For each terminal in the system, there is a user agent that:

- Parses terminal input.
- Interprets special input sequences as user commands to manipulate the display.
- Calls the needed operations.

There is currently one user agent defined, for character terminals. See the "Character Terminal User Agent" appendix in the *BiiN™/OS Reference Manual.*

IV-4.1.14 Character Terminal Manager

See the `Character_Terminal_Mgt` package for more information about the character terminal manager.

IV-4.1.15 Character Terminal Manager Support for Input Operations

The general windows input model is described on page IV-4-6. The character terminal manager supports that model with the following clarifications.

The character terminal manager generates all defined input event types except `mouse`, `overlap_changed`, and `user_defined`.

An application uses `Character_Display_AM.Ops.Read` to read the input events of a window opened using `Character_Display_AM.Ops.Open`.

An application can use the `Set_input_type_mask` call of one of the access methods to specify the input events that are accepted, and later read, from a window.

The Read call of Record_AM and Byte_Stream_AM can be mixed with the Read call of Character_Display_AM as follows. The Read call of Record_AM and Byte_Stream_AM only returns one input event type, keyboard input. If an application calls Read in Byte_Stream_AM or Record_AM Read for a window whose current input mask does not include keyboard input, the keyboard input type is added to the current input mask. (The keyboard type is not subsequently removed from the input mask unless the application explicitly requests that action using Set_input_type_mask.) The Read call searches the input buffer of the virtual terminal for keyboard input. All input events with types other than keyboard are discarded until keyboard input is found. The keyboard input is then returned to the user. If no keyboard input is found and the block parameter is true, the application blocks until keyboard input arrives.

With respect to reading keyboard input, the Read functions of Character_Display_AM and Byte_Stream_AM have the same semantics. If the window is in line editing mode and keyboard input is currently at the front of the input queue, then characters are copied to the caller's buffer until the number of characters requested is reached, an end of line delimiter is encountered, or some other type of input event is encountered. If the window is not in line editing mode and keyboard input is currently at the front of the input queue, then characters are copied to the caller's buffer until the number of characters requested is reached, some other type of input event is encountered, or there is no more input currently available for the window.

The Read function of Record_AM is exactly the same with one exception. If the window is in line editing mode and the size of the line is greater than the length of the read request, then the number of bytes requested is copied to the caller's buffer and the caller receives a length_error exception. The remainder of the line can be read by calling Read with the rec_position parameter set to rest_of_current.

IV-4.1.16 Character Terminal Manager Support for Output Operations

The general windows output model is described on pages IV-4-7. The character terminal manager fully supports that model.

For more information about specific input and output operations, see the Byte_Stream_AM, Record_AM, and Character_Display_AM packages.

IV-4.1.17 Character Terminal Manager Support for Access Method Operations

The character terminal manager supports Byte_Stream_AM, Record_AM, and Character_Display_AM.

IV-4.1.17.1 Character Terminal Manager Support for Byte_Stream_AM

The character terminal manager supports all the operations of Byte_Stream_AM except for the following:

- Set_position
- Truncate.

Several Byte_Stream_AM operations have different semantics with windows:

- Open and Get_device_info take a window device as a parameter, instead of a terminal device.

- Get_device_object takes an opened device and returns the window device that was opened to get the opened device, rather than the terminal on which the window resides.
- Enable_input_notification will signal an action when any input is in the input queue, but the application may not receive any input (for example, because of the settings of the input mask or because of device limitations). If this happens and the application does not wish to block when it performs a subsequent Read operation, it should set the block parameter of Read to false and be prepared to receive no input.

IV-4.1.17.2 Character Terminal Manager Support for Record_AM

The character terminal manager supports all the operations of Record_AM except for the following:

- Delete
- Insert
- Insert_control_record
- Lock_all
- Set_position
- Truncate
- Unlock
- Unlock_all
- Update
- Write_control_record.

There are several restrictions in accessing windows by Record_AM. The only access mode supported is physical sequential. The sequential positions supported for Read are next and rest_of_current. The only write position supported is after_last. The operations not supported are Set_position, Write_control_record, Rewrite, Delete, Truncate, Lock_all, and Release.

Several Record_AM operations have different semantics with windows:

- Get_device_info and Get_device_object have the same semantics as their counterparts in Byte_Stream_AM.
- Open operates on a window device instead of a terminal, and the extend, single_rec_lock, read_ctl_rec, and load parameters are ignored.
- Write writes a line to the window. The line terminates at the end of the write and the cursor moves down one row or moves to the beginning of the next row depending on the current output attributes.
- New_line and New_page are interpreted appropriately for windows. New_line causes a line feed character to be written to the frame buffer. (Depending on the output attributes in effect for the window, other characters such as a carriage return may also be written to the frame buffer.) New_page clears the frame buffer to the background color and leaves the cursor at the first column of the first row.
- Get_status returns an AD to a record of information about the current status of a window. Most fields are set to constant values and are not meaningful for windows. The only meaningful fields are input_output, usage, operation_completed, last_call, rec_length, and at_EOF. If the length of the current record cannot be

determined (because the user has not typed a line terminator), `rec_length` is set to the `unknown_length` constant defined in `Record_AM`. The `at_EOF` field is set to true when an end-of-file character is encountered and all previous user input has been read. These fields are updated after each `Record_AM` operation.

- The above comments about `Enable_input_notification` and `Read` (see "Character Terminal Manager Support for `Byte_Stream_AM`") are also applicable here.

IV-4.1.17.3 Character Terminal Manager Support for `Character_Display_AM`

The character terminal manager implements all the operations of `Character_Display_AM` except `Font_list`.

Several `Character_Display_AM` operations have different semantics with character terminals.

- `Set_terminal_attr` and `Set_window_attr` have the normal semantics except that a few attributes are not relevant to character terminals and are ignored. Those attributes are `mouse_sampling`, `font_index`, `background_color`, and `text_color`.
- The above comments about `Enable_input_notification` and `Read` (see "Character Terminal Manager Support for `Byte_Stream_AM`") are also applicable here.

IV-4.2 Techniques

This section shows you some techniques for using `Window_Services`:

- Obtaining an AD for the underlying terminal
- Creating a window
- Setting a window's attributes
- Setting a window's style.

These techniques are shown through excerpts from the `Simple_Editor_ex` procedure. `Simple_Editor_ex` is a simplified screen editor allowing certain operations on an ASCII text file.

Appendix X-A contains the complete listing of `Simple_Editor`.

IV-4.2.1 Obtaining an AD for the Underlying Terminal

Calls Used:

`Window_Services.Ops.Get_terminal`
Returns the terminal containing a window.

One of the editor's first tasks is to create a new window from the old opened window. This requires the editor to obtain an AD for the underlying terminal, as in the following call to `Get_terminal`:

```
-- Create new window from old opened window.
old_window := Character_Display_AM.Ops.
    Get_device_object(Process_Mgt.Get_process_globals_entry(
        Process_Mgt_Types.standard_input));
underlying_terminal := Window_Services.Ops.
    Get_terminal(old_window);
```

IV-4.2.2 Creating a Window

Calls Used:

`Window_Services.Ops.Create_window`
Creates a window on a terminal.

After obtaining the underlying terminal, the editor needs to create a new window in which the editing operations can take place. To do this, the editor calls `Create_window`:

```
edit_window := Window_Services.Ops.Create_window(
    terminal            => underlying_terminal,
    pixel_units         => false,
    fb_size             => Terminal_Defs.point_info'(
        last_column, frame_rows),
    desired_window_size => Terminal_Defs.point_info'(
        last_column, preferred_window_rows),
    window_pos          => origin,
    view_pos            => origin);
```

IV-4.2.3 Setting a Window's Attributes

Calls Used:

`Window_Services.Ops.Set_window_attr`
Modifies a window's attributes.

Next, the editor is changes some of the window's default attributes by modifying selected fields of `window_attributes` and then calling `Set_window_attributes`.

```
window_attributes:  Terminal_Defs.window_attr :=
    Terminal_Defs.default_window_attr;
 . . .
-- Set window's input and output attributes
-- change from default:
window_attributes.enable_signal := false;  -- for ^C ^B
window_attributes.line_editing := false;   -- for ^H
window_attributes.echo := false;
-- NOTE: track_cursor NYI (use user agent to change view)
window_attributes.track_cursor := true;
Window_Services.Ops.Set_window_attr(
    window    => edit_window,
    attr      => window_attributes,
    attr_mask => (others => true));
```

IV-4.2.4 Setting a Window's Style

Calls Used:

`Window_Services.Ops.Set_window_style`
Modifies a window's style information.

The editor's final task prior to opening the window for editing is to define the window's *title bar* by calling `Set_window_style`:

```
-- Set Title and Info lines
Text_Mgt.Set(new_window_info.title, file_name);
Window_Services.Ops.Set_window_style(
    window     => edit_window,
    new_info   => new_window_info,
    style_list => (others => true));
```

IV-4.3 Summary

- A window is a portion of the terminal screen through which a user interacts with an application program. In general, the screen can contain multiple windows, and the windows can overlap. Both character and graphics terminals can have windows.
- An application creates and manipulates windows through the `Window_Services` package, which provides a procedural interface for creating and destroying windows, manipulating the characteristics of windows, and installing menus on windows.
- Once a character terminal window has been set up, four access methods are available for performing I/O in the window: `Character_Display_AM`, `Byte_Stream_AM`, and `Record_AM`.
- `Window_Services` offers both character-oriented and pixel-oriented coordinate systems.
- The user controls the layout, size, and visibility of windows, but an application has full control of a window's contents.
- Resize rules allow an application to specify a stable point in the view when a window is resized.
- An application can control two aspects of a window's style: the user interactions permitted for the window and the window's appearance. The window's appearance and the user's interactions may be related (for example, through border areas).
- Pull-down menus can be associated with windows. The number of menus per window and the menus' features are device dependent, but as a mimimal level of support, all devices support a menu consisting of a single list of text items that can be selected from the keyboard.
- Pull-down menus have a three-part hierarchy: menu group, menu, and menu item. There are two phases to menu manipulations: building the menu and using the menu.
- `Window_Services` provides a way for applications to find out about changing view and window information in a timely manner by defining a number of `Window_Services` input events that can be read by an application reading from a window. These input events inform the application about events that occur to the window. The application can then take whatever action is appropriate.

USING CHARACTER DISPLAY I/O 5

Contents

This chapter shows how to do I/O to a character display device.

Packages Used:

`Character_Display_AM`
: Provides device-independent I/O to character display devices such as printers, plotters, and windows on character and graphics terminals.

`Window_Services`
: Provides windows on character and graphics terminals, including pull-down menus.

`Terminal_Defs`
: Defines constants, types, and exceptions used by the terminal service packages.

IV-5.1 Concepts

This section presents an overview of character display devices and how an application accesses them using the `Character_Display_AM.Ops` package.

IV-5.1.1 Character Display Devices

A character display device displays and manipulates ASCII characters on a two-dimensional surface. The most common example of a character display device is a window on the screen of a character or graphics terminal. Windows are virtual devices through which I/O is done to a physical terminal. Another example of a character display device is a printer.

The operations for character display 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.

Mixing access methods means opening a device using the `Open` function of one access method and then using the resulting opened device object to call operations of another access method. The extent to which access methods can be mixed is device dependent. Character display I/O can generally be mixed with byte-stream I/O and record I/O.

IV-5.1.2 The Frame Buffer

To manipulate characters on the device's two-dimensional display surface, an application program performs operations on a conceptual entity called the *frame buffer*. The frame buffer is a two-dimensional grid of character cells whose contents are displayed on the surface of the device by a device-specific implementation of `Character_Display_AM.Ops`.

On character and graphics terminals, for example, part of the frame buffer is visible through a rectangular area of the terminal screen called a *window*. The visible part of the frame buffer is called the *view*. Figure IV-5-1 depicts these relationships.

For more information on windows, views, and frame buffers, see Chapter IV-4.

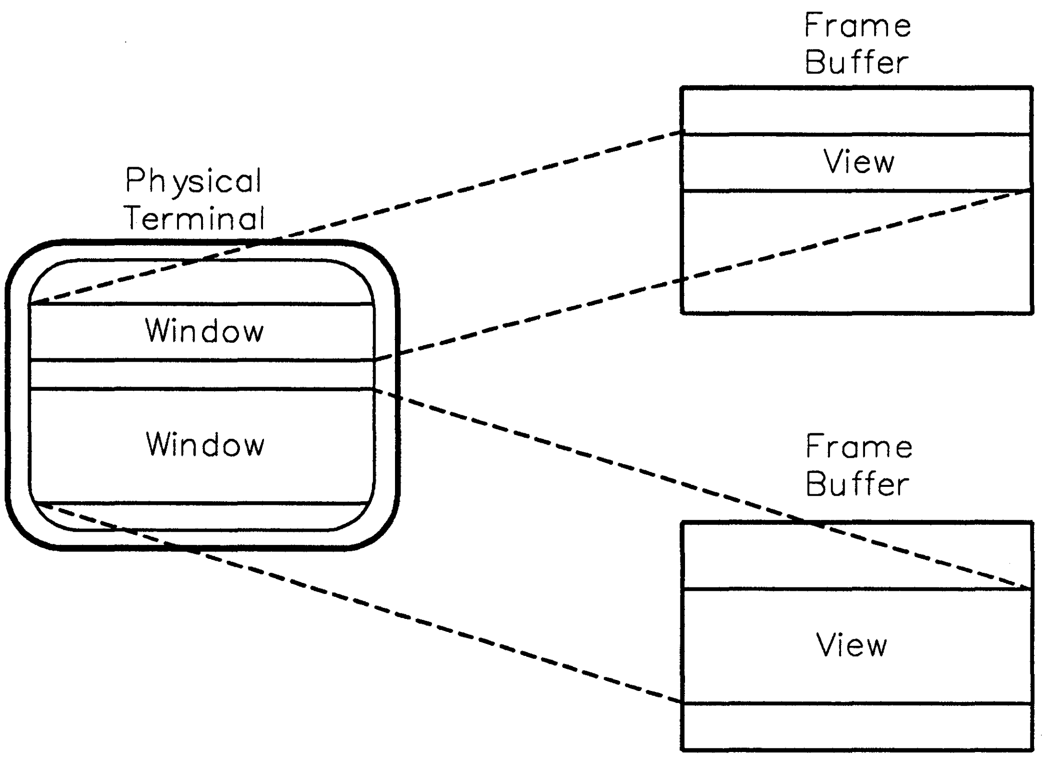


Figure IV-5-1. Views, Windows, and Frame Buffers

The mechanism for specifying the dimensions of a frame buffer is device-dependent. Calling `Window_Services.Ops.Create_window`, for example, sets the dimensions of the frame buffer as well as the desired size and position of the corresponding view and window.

On a character terminal, each cell in the frame buffer contains a displayable ASCII character; on a graphics terminal, each cell is measured in pixels and has a fixed width and height. In both cases, cells are identified by their (column,row) coordinates, starting with (1,1) in the upper left-hand corner of the frame buffer and increasing downwards and to the right. Note that the column coordinate comes first in the coordinate pair.

Figure IV-5-2 shows the frame buffer's coordinate system.

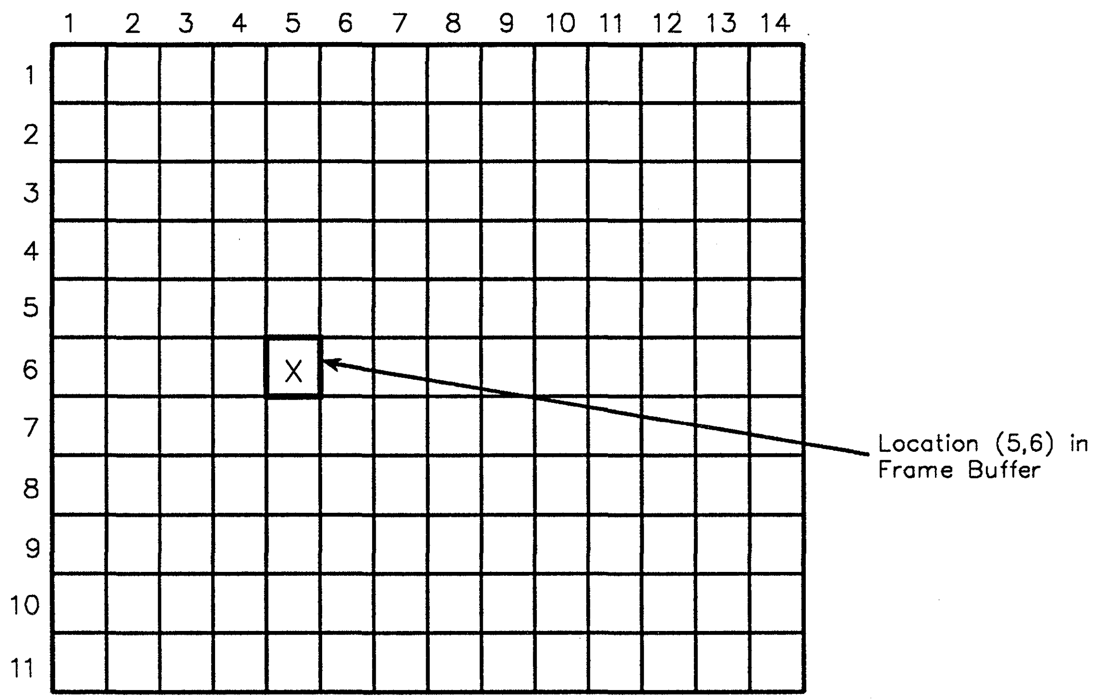


Figure IV-5-2. The Frame Buffer Coordinate System

A *cursor* is used to identify specific character cells within the frame buffer. For example, the `Write` procedure writes characters at the cursor's current location and moves the cursor to a new location. `Insert_char` inserts characters at the cursor's current location and moves trailing characters to the right. `Move_cursor_absolute` and `Move_cursor_relative` move the cursor without affecting the display.

IV-5.1.3 The Output Model

To write characters to the current cursor location in the frame buffer, an application uses the `Write` procedure. Printable characters are written and the cursor is moved *forward*, while control characters are either discarded, printed as **<Ctrl-X>**, or given special treatment. Note that writing to location *(x,y)* in the frame buffer overwrites the previous contents of that location.

To insert characters at the current cursor location in the frame buffer, an application uses the `Insert_char` procedure. Printable and control characters are treated as in `Write`. Trailing characters are moved to the right (and off the edge of the frame buffer if necessary).

IV-5.1.4 The Input Model

Input is obtained with the `Read` call. `Read` is not supported by output-only devices, such as print devices.

Input is classified according to the type of *input event* it represents. `Terminal_Defs.input_enum` enumerates the input event types, and `Terminal_Defs.input_type_mask` specifies which input event types are currently accepted for a device.

Note that input events are not related to the event types discussed in the `Event_Mgt` package.

The following is a list of the input event types:

- `keyboard`
- `mouse`
- `menu_item_picked`
- `focus_changed`
- `overlap_changed`
- `size_changed`
- `view_changed`
- `position_changed`
- `close_requested`
- `scroll_requested`
- `user_defined`

This input model has a number of important features:

- The `Read` procedure blocks until input arrives (if blocking is requested), and then returns input data.
- Input from all sources is merged into one stream. This preserves the time-ordering of input events, which is crucial to applications where keyboard input describes something to be drawn at the current mouse position.
- Although input from different sources is merged into one stream, contiguous sequences of input events from different sources are returned separately; that is, a call to `Read` returns only one type of input event. For example, if the input buffer for a device currently contains 7 keyboard characters, 2 mouse events, and 23 more keyboard characters, then three successive requests to read 80 bytes of input data will return, in succession, 7 characters, 2 mouse events, and 23 characters. Because different types of input events are not interleaved within a single read, the application does not have to parse the input stream to separate the input events.
- Because `Terminal_Defs.input_type_mask` determines which input event types to accept, unwanted input is discarded at the earliest possible time.
- The application can change the set of accepted input event types at any time by changing `Terminal_Defs.input_type_mask`, but caution is important when doing so. For example, suppose the current input mask accepts only keyboard input. Then suppose that the application decides to accept mouse input, notifies the user of this, and changes the input mask accordingly. If the user manipulates the mouse *before* the application changes the input mask, mouse events may be lost. The application can avoid this by initially setting the input mask to accept both keyboard and mouse events, and then discarding the mouse events when they are not meaningful.
- Traditional keyboard-only input looks like simple byte-stream input. By default, keyboard input is the only type of input accepted for a new device. An application can obtain keyboard input for a device by calling the `Read` procedure in `Byte_Stream_AM.Ops`, `Record_AM.Ops`, or `Character_Display_AM.Ops`.

IV-5.1.5 Window Attributes

A set of attributes is defined for each window on a terminal. These attributes affect the I/O operations performed in the window. Table IV-5-1 lists the attributes and their suggested defaults.

See `Terminal_Defs.window_attr` for detailed information about the affect of these attributes.

Window attributes can be queried and set using the following `Terminal_Admin.Ops` calls:

- `Get_Terminal_Attr`
- `Set_Terminal_Attr`

Table IV-5-1. Window Attributes

Attribute	Default Setting
font_index	1
background_color	dark (color intensities all 0)
text_color	white (color intensities all ordinal'last)
map_out_LF_to_CRLF	true
map_out_CR_to_LF	true
scroll	true
linewrap	true
map_control_chars	true
fill_is_DEL	false
track_cursor	false
optimize_output	true
enable_signal	true
line_editing	true
mouse_sampling	button_sampling
echo	true
echo_erase	true
echo_LF_cancel	true
no_flush	false
ignore_in_CR	false
map_in_CR_to_LF	true
map_in_LF_to_CR	false

IV-5.1.6 Operations

The calls in `Character_Display_AM.Ops` fall into six logical groups:

- Input
  - `Get_input_type_mask`
  - `Set_input_type_mask`
  - `Read`

- `Enable_input_notification`
- `Disable_input_notification`
- `Discard_input`

- Output
  - `Write`
  - `Insert_char`
  - `Flush`
  - `Discard_output`
  - `Ring_bell`
- Cursor Movement
  - `Get_cursor_position`
  - `Move_cursor_absolute`
  - `Move_cursor_relative`
- Display
  - `Clear`
  - `Clear_to_bottom`
  - `Clear_to_end_of_line`
  - `Delete_char`
  - `Delete_line`
  - `Insert_line`
- Control and Status
  - `Open`
  - `Close`
  - `Is_open`
  - `Get_device_object`
  - `Get_device_info`
  - `Begin_batch_changes`
  - `End_batch_changes`
- Attributes
  - `Get_enhancement`
  - `Set_enhancement`
  - `Set_region_enhancement`
  - `Font_list`

IV-5.2 Techniques

This section shows you some techniques for using `Character_Display_AM.Ops`:

- Opening a window
- Clearing the frame buffer
- Writing to the frame buffer
- Moving the cursor to an *absolute* position
- Moving the cursor *relative* to its current position
- Reading input events
- Inserting characters
- Deleting characters
- Identifying the underlying device.

These techniques are shown through excerpts from the `Simple_Editor_ex` procedure. `Simple_Editor_ex` is a simplified screen editor allowing the following operations on an ASCII text file:

- Moving the cursor forward one column
- Moving the cursor back one column
- Moving the cursor up one row
- Moving the cursor down one row
- Paging up by the size of the view
- Paging down by the size of the view
- Deleting a character (forward)
- Deleting a character (backward)
- Inserting text
- Saving the edited file
- Quitting the editor.

Appendix X-A contains a complete listing of `Simple_Editor_ex`.

Because of the nature of this example, these techniques are oriented toward windows and terminals.

IV-5.2.1 Opening a Window

After reading the file to be edited into a buffer and creating a window in which to do the editing, the editor needs to open the window:

```
-- Open the edit window
open_edit_window := Character_Display_AM.Ops.Open(
    device        => edit_window,
    input_output  => Device_Defs.inout,
    exclusive     => true);
```

edit_window is the device returned by a call to Window_Services.Ops.Create_window.

The other two parameters indicate that the window is to be opened for both input and output, with exclusive access.

IV-5.2.2 Clearing the Frame Buffer

The editor's next task is to clear the frame buffer and move the cursor to the (1,1) position:

```
-- Clear window on terminal screen.
Character_Display_AM.Ops.Clear(open_edit_window);
```

The opened_edit_window is the opened device returned by the earlier call to Open.

Related calls clear the frame buffer from the current cursor position to the end of the line (Clear_to_end_of_line) and from the current cursor position to the end of the window (Clear_to_bottom).

IV-5.2.3 Writing to the Frame Buffer

Next, the editor needs to display the file in the window. The Write operation writes characters to the frame buffer (overwriting its previous contents) and leaves the cursor after the written data:

```
-- Write from edit buffer to frame buffer.
-- NOTE: There cannot be more line_feeds in the length
-- of characters written than there are rows in
-- the frame buffer, otherwise some of the first
-- characters will be overwritten in the frame buffer
-- The last line is written up to the line feed to
-- avoid having a blank line at bottom of the window
Character_Display_AM.Ops.Write(
    opened_dev => open_edit_window,
    buffer_VA  => edit_buffer.lines'address,
    length     => System.ordinal((last_column * (frame_rows - 1))
        + (Last_char_in_row(frame_end) - 1)));

```

edit_buffer.lines'address is the address of the beginning of the buffer.

The formula for the length parameter writes up to, but not including, the linefeed in the last line that the frame buffer will write. If the last linefeed is written the first line of the frame buffer will be overwritten and the last line of the frame buffer will be blank.

IV-5.2.4 Moving the Cursor to an Absolute Position

Since the write operation leaves the cursor after the last character, the editor needs to move the cursor to the (1,1) position by calling Move_cursor_absolute:

```
-- Home the cursor (1,1 position).
Character_Display_AM.Ops.Move_cursor_absolute(
    opened_dev => open_edit_window,
    new_pos    => origin);
```

This procedure is called *absolute* because the new cursor coordinates specify an absolute location within the frame buffer. That is, calling the procedure with coordinates *(x,y)* moves the cursor to column *x* and row *y* of the frame buffer. (Column numbers increase to the right; row numbers increase downward.)

IV-5.2.5 Moving the Cursor Relative to its Current Position

Cursor movement may also be relative to the cursor's current location in the frame buffer. For example, the following call moves the cursor forward (to the right) one column by assigning the value 1 to the parameter `delta_col`:

```
Character_Display_AM.Ops.Move_cursor_relative(
    opened_dev => open_edit_window,
    delta_col  => 1,
    delta_row  => 0);
```

Assigning -1 to `delta_col` would move the cursor backward (to the left) one column, and assigning 1 (-1) to `delta_row` would move the cursor down (up) one row.

IV-5.2.6 Reading Input Events

The default input mask is for keyboard input so a call to `read` is all that needed for input.

```
procedure Handle_input

is

  event_num:       System.ordinal;
  event_type:      Terminal_Defs.input_enum;
  char_buffer_AD:  char_array_AD := new char_array'(others => ' ');

begin

  -- Enter the basic read and process loop
  loop
    -- Read the next input event
    -- default input mask is keyboard
    Character_Display_AM.Ops.Read(
        opened_dev => open_edit_window,
        buffer_VA  => char_buffer_AD.all'address,
        max_events => 1,
        max_bytes  => 0,
        block      => true,
        type_read  => event_type,
        num_read   => event_num);
      case event_type is
        when Terminal_Defs.keyboard =>
          -- ...
          key_input(char_buffer_AD(1));
        when Terminal_Defs.menu_item_picked =>
          -- ...
          key_input(char_buffer_AD(1));
        when others =>
          null;
      end case;
    end loop;
end Handle_input;
```

Setting the maximum number of events (`max_events`) to 1 and the maximum number of bytes (`max_bytes`) to 0 ensures that exactly one event will be read. The procedure blocks until input is available, and returns the number and type of the event in `event_type` and `event_num` respectively.

`char_buffer_AD.all'address` is the address of the buffer in which the input events will be placed.

IV-5.2.7 Inserting Characters

The following code inserts a character at the current cursor location in the frame buffer.

```
-- Insert the character in the frame buffer
-- (Frame buffer cursor is moved automatically)
Character_Display_AM.Ops.Insert_char(
    opened_dev => open_edit_window,
    buffer_VA  => insert_char'address,
    num_char   => 1);
```

Unlike `Write` which overwrites characters, `Insert_char` pushes trailing characters to the right and leaves the cursor at the end of the inserted string. Characters pushed off the edge of the frame buffer are lost.

IV-5.2.8 Deleting Characters

The following code deletes a character from the frame buffer.

```
-- Delete the character from the window.
Character_Display_AM.Ops.Delete_char(
    opened_dev => open_edit_window);
```

Note that the number of characters deleted can be specified (here it defaults to 1) and that deletion starts at the cursor's current location. Trailing characters on the line are moved to the left, and new space at the end of the line is cleared to the current background color.

IV-5.2.9 Identifying the Underlying Device

To create a new window from an old window, the editor obtains a reference to the device underlying the old opened window by calling `Get_device_object`:

```
-- Create new window from old opened window.
old_window := Character_Display_AM.Ops.
    Get_device_object(Process_Mgt.Get_process_globals_entry(
        Process_Mgt_Types.standard_input));
underlying_terminal := Window_Services.Ops.
    Get_terminal(old_window);
edit_window := Window_Services.Ops.Create_window(
    terminal            => underlying_terminal,
    pixel_units         => false,
    fb_size             => Terminal_Defs.point_info'(
        last_column, frame_rows),
    desired_window_size => Terminal_Defs.point_info'(
        last_column, preferred_window_rows),
    window_pos          => origin,
    view_pos            => origin);
```

IV-5.3 Summary

- The `Character_Display_AM` package is used for doing I/O to character display devices. Common examples of such devices are printers and windows on character and graphics terminals.
- To manipulate characters on the device's display surface, an application program performs operations on a frame buffer, a two-dimensional grid of character cells.
- The operations defined for character display devices fall into six groups: input, output, cursor movement, display, control and status, and attributes .
- Both input and output are defined for windows, but input is not defined for devices like printers. Input is implemented through a `Read` procedure that uses an input mask to

specify which types of input events are accepted. Output is implemented through the `Write` and `Insert_char` procedures.

- Cursor movement operations and output operations occur within the frame buffer. Each *(x,y)* location in the frame buffer specifies a character cell that contains a displayable ASCII character.
- Display operations alter the appearance of the display surface. There are several operations to clear the display.
- The attribute operations query and set the attributes of a device.
- The control and status operations perform basic activities like opening and closing devices.

PRINTING 6

Contents

IV-6.1 Concepts

This chapter describes spooled and direct printing.

Packages Used:

`Spool_Defs`	Declares types and constants used by spooling packages.
`Spool_Device_Mgt`	Manages spool devices.
`Spool_Queue_Admin`	Provides administrative calls for spool queues.
`Printer_Admin`	Provides administrative operations for printers.

A *spool queue*, which is required for all printing, is usually installed at system configuration. A *print device* is created by an application for printing. The print device is opened by an access method. If the print device is opened in spooled mode, the application's output is written to a spool file to await printing. Printing also requires an opened *printer*. See Figure IV-6-1.

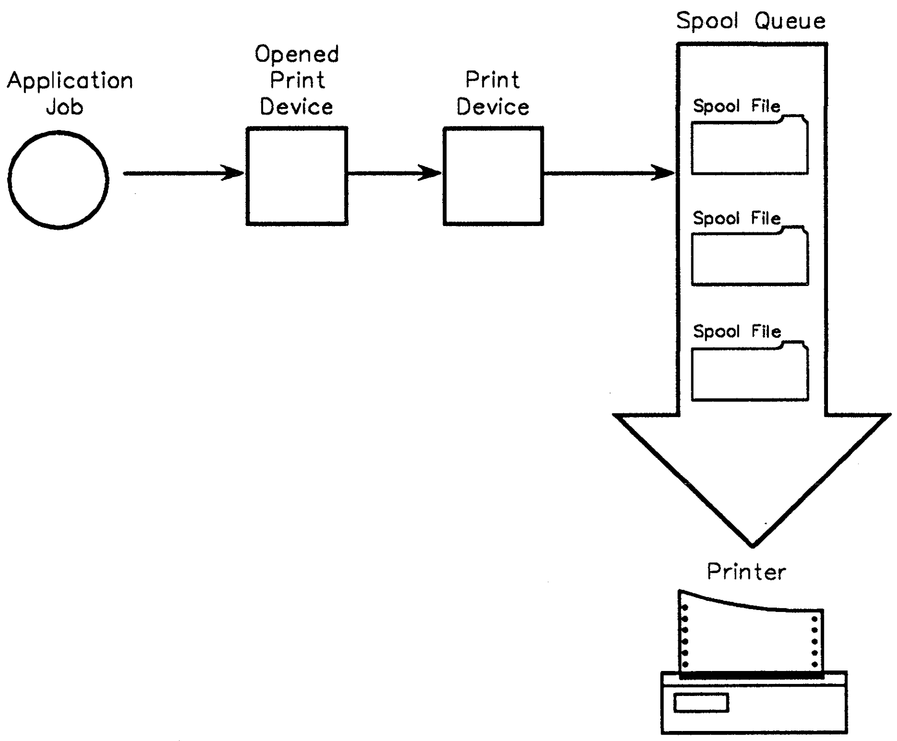


Figure IV-6-1. Spooled Printing

Output attributes, optional capabilities and basic printing and spooling functions can be controlled through the procedural interface.

IV-6.1.1 Spool Queue

A *spool queue* is required for all printing. A spool queue contains spooled files, printer properties and capabilities, spooling properties, and a list of connected printers. Printer devices are created in association with a specific spool queue.

IV-6.1.2 Print Device

A *print device* is a device created by an application through which data is spooled or printed. A print device must be created before writing a spool file or printing directly and is associated with a specific spool queue. It contains information specific to the print job including whether the print mode is *spooled* or *direct.*

The lifetime of a print device is limited by the application which created it. The application is responsible for explicitly deallocating the print device. The same instance of a print device can be used for several opens, writes and closes.

IV-6.1.3 Spooled Printing

A print device may be opened in *spooled* or *direct* mode. Spooled is by far the more commonly used mode. When the mode is spooled, output is written to a spool file which is attached to the spool queue. The point at which the spool file is printed depends on:

- Its position in the *rank* (relative position) of spool files for the spool queue,
- The priority of the spool queue compared to other spool queues, and
- Whether the spool file's rank is changed by moving it to another rank position, to another spool queue or by deleting it before it prints.

IV-6.1.4 Direct Printing

A print device may be opened in *direct* mode. Generally this mode is infrequently used, usually chosen when printed information is urgently required. As with spooled mode, direct mode is controlled by the spool queue. The difference is that printing is written directly to a printer rather than to a spool file.

IV-6.1.5 Spool File

When a print device is opened for spooled printing, an `Open` call creates a new spool file. `Write` calls send data to a spool file.

`Close` readies the spool file for printing and allocates it in first in, first out order on its spool queue. The spool file is then printed when it reaches the top of the spool file rank and one of the printers connected with the spool queue is ready.

IV-6.1.6 Printer Lists

Lists of printers available for spooled and direct printing are maintained in spool queues, print devices and spool files. A member of a *printer list* is free for printing if it is not already engaged in printing and has the required form type mounted.

When a print device is created for a spool queue, the print device inherits the complete printer list of the spool queue. This list may be modified but must be a full set or subset of the spool queue's list. The print device's printer list is inherited by its spool files.

A printer's connection with a spool queue is not exclusive. One spool queue may be connected with several printers, and one printer may be connected with several spool queues. In the latter case, the spool queue *priority* determines the order in which competing spool queues will gain use of a single printer to which they are connected. In direct printing, spool queue priority is not evaluated. The spool service simply uses the requested printer if it is available.

IV-6.1.7 Print Area and Print Position

The *print area* defines the area on a sheet or form where output may be printed. Characteristically, the print area will be less than the physical size of the page, being indented from the top and left side of the physical page. The *print position* is the location within the print area where printing will next commence. The initial print position for character output is top and left (column 1, line 1). See Figure IV-6-2.

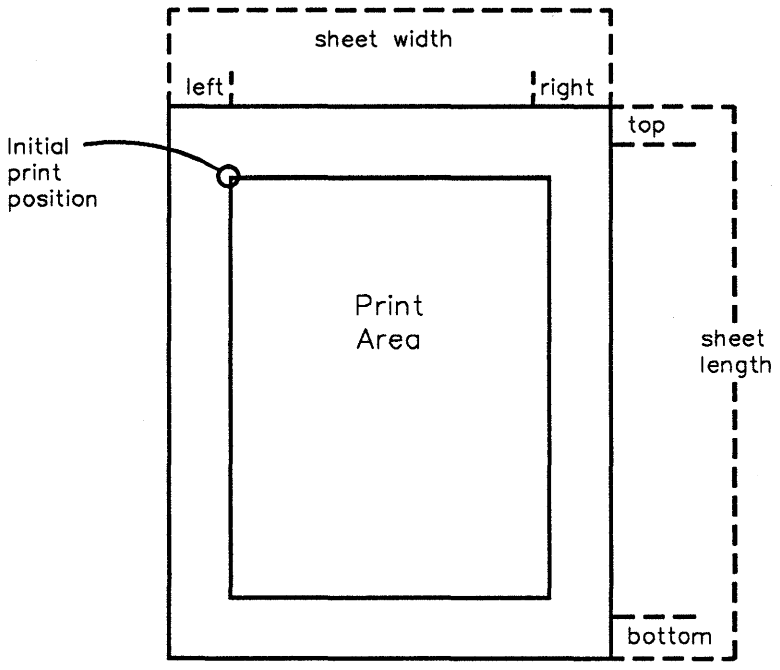


Figure IV-6-2. Print Area

IV-6.1.8 Requesting Form Type and Sheet Size

Applications whose output must print on a certain form or specific size of sheet may request that the needed form or sheet be mounted before printing. The request may be one of three kinds:

none - No form or sheet requirements

form - Name of a form

sheet - Size of (plain) sheet required.

If a form or sheet is specified, print or spool service will prompt for acknowledgement that the requested form or sheet size is mounted before printing.

IV-6.1.9 Printinfo

Printinfo entries contain printer capabilities corresponding to a *printer type* (printers with the same capabilities). The entries in the printinfo denote, for example, whether the printer supports colors, and what bytes trigger corresponding functions, such as the bytes that make the printer print in red. A spool queue contains a printinfo reference which is specified when the spool queue is installed.

When a printer is connected to a device, the printer's actual capabilities are checked against the capabilities in the printinfo entry associated with a spool queue. The capabilities associated with the printer must be equal to or be a superset of the capabilities referenced by the printinfo associated with the spool queue. If this requirement is met, spooled and direct printing and printer emulation can be supported on a specific printer.

IV-6.1.10 Print Properties

Print properties are maintained in spool queues, spool files, and print devices. These properties include:

- Device status
- Termination message description and enable/disable status
- Banner description and enable/disable status
- Spool file deletion enable/disable status
- Print copy count
- Printer list.

These properties can be set and queried. A `Close` call disconnects the spool file from the print device. Therefore, to modify or inquire about the properties of a spool file, the inquiry calls should be made to the spool file instead of the print device. Also, deleting a spool file does not affect an existing print device for which the spool file has been created, and vice versa.

IV-6.1.11 Implementation of Spool Device Attributes

Spool device is a collective term applied to the following objects: *spool queue*, *print device*, *spool file*, and *printer*. Table 1-1 defines the availability of spool device attribute implementations among the various spool devices. `"X"` indicates that the attribute(s) is available for the indicated spool device, `"XS"` indicates that the attribute is only available for a print device when it is opened in spooled mode, and `"-"` indicates that the attribute(s) is not available for the indicated spool device.

Table IV-6-1. Implementation of Spool Device Attributes

	SPOOL DEVICE			
Spool_Device_Mgt.Ops. calls	**print device**	**spool file**	**spool queue**	**printer**
Get_operation_support				
Get_access_method_support				
Get_device_status	X	X	X	X
Is_graphics_device				
Get_standard_character_size				
Get_parameter_text				
Set_parameter_text				
Get_print_area_position	X	X	-	-
Get_print_area_size				
Is_pixel_units				
Get_printer_list				
Connect_printer				
Disconnect_printer				
Get_termination_message				
Set_termination_message				
Is_termination_message_enabled				
Enable_termination_message	X	X	X	-
Disable_termination_message				
Get_banner_page_line				
Set_banner_page_line				
Is_banner_page_enabled				
Enable_banner_page				
Disable_banner_page				
Delete_device				
Get_request_class	X	X	-	X
Get_sheet_request				
Set_sheet_request	X	-	-	X
Is_printing_enabled				
Enable_printing	-	X	X	X
Disable_printing				
Abort_printing				
Get_copy_number				
Set_copy_number				
Is_deleting_enabled	XS	X	X	-
Enable_deleting				
Disable_deleting				
Set_test_print	XS	X	-	-
Get_print_count	-	X	-	-

IV-6.1.12 Delayed Printing

Printing can be explicitly delayed for a spool queue until a specified time or delayed until a specified time when the size of the print job exceeds a specified limit. Print delays are classified as `none` for no delay, `time` for a timed delay, or `size` for a delay dependent upon the size of the spool file and the number of outstanding print requests.

IV-6.1.13 Banner Page and Print Termination Message

A *banner page* is an optional page which may be printed to identify a print job. A banner page consists of up to five lines. A *print termination message* is an optional message which may be sent to the user who created the print device to signal completion of a print job.

The lines of banner pages and termination messages are defined by `Incident_Defs.incident_code`. Formal parameters `$p1` through `$p5` may be used within message texts used to defined banner page lines and termination messages. Formal parameters are defined in the *BiiN™ Command and Message Guide.* Each time a print device is created, these formal parameters are assigned the following defaults:

`$p1` - The *basename* of the spool queue where the print device has been created.

`$p2` - The *node* on which the print device has been created.

`$p3` - The *job* within which the print device has been created.

`$p4` - The *time* at which the spool file was created or the print device was opened for direct printing.

`$p5` - The *user* who created the print device.

The formal parameters are updated when a spool file is created (spooled mode) or a print device is opened (direct mode).

IV-6.1.14 Default Properties

A new spool queue receives the following default properties:

- Spool queue properties:

  Spooling is enabled

  The lowest spool queue priority is assigned

  No spool queue print delay is set.

- Print properties:

  No printers are connected

  Printing is enabled

  No multiple copies are requested

  Automatic deleting of printed spool files is enabled.

- Other properties:

  No banner page will be printed

  No printer termination message will be issued.

A new spool queue receives the following default properties:

Print mode is `spooled`

Print position is (1,1).

When a print device is created, it inherits the print properties of its associated spool queue. When a print device created in spooled mode is opened, the resultant spool file inherits the print device's print properties. The application may modify these print properties at any time between the creation of the print device and the time that the spool file begins to print. Print properties of a print device created for direct printing may be modified after the print device is created and before printing commences. Attempts to modify print properties after printing has commenced will fail.

IV-6.2 Techniques

After reading this section, you will be able to:

- Print to a spool file
- Print directly to a printer
- Control print properties
- Administer spool queues and printers
- Add a new type of printer.

The sample code segments are excerpted from the `print_cmd_ex` example package. Appendix A contains a complete listing of this package.

IV-6.2.1 Printing to a Spool File

Calls Used:

`Spool_Device_Mgt.Create_print_device`
Creates a print device for a spool queue.

`Spool_Device_Mgt.Create_print_device_by_name`
Creates a print device for a spool queue.

`Spool_Device_Mgt.Ops.Set_copy_number`
Sets the number of copies to print.

The first step in sending output to a printer is to create a *print device*. Then the print device is opened and the output is written to a *spool file* for spooled mode or to na *printer* for direct mode (via calls of an access method supported by the print device). The actual time that a spool file is printed depends on queue priorities and delays and printer availability. Output written in direct mode is printed immediately if the requested printer is available, otherwise the write call must be reissued until the printer is available.

In the following example, `Spool_Device_Mgt.Create_print_device` is called to create a print device for spooled printing. `pixel_units` are set to `false` which causes the `print_area` to be interpreted as character cells. The `print_area` is set to 132 characters wide by 66 characters long. The `print_mode` defaults to `spooled`. The print device is then opened and written to with byte stream access method calls.

After the print device is closed, the data is written to a spool file buffer to await an available printer or sent directly to a specific printer, depending upon the mode in which the print device was created.

```
sheet_size:       constant Spool_Defs.size_t :=
    (132,66);
on_untyped := Directory_Mgt.Retrieve(on_device);
if Spool_Defs.Is_spool_queue(on_untyped) then
  print_device :=
      Spool_Device_Mgt.Create_print_device(
          spool_queue => spool_queue,
          pixel_units => false,
          print_area  => sheet_size);
open_print := Byte_stream_AM.Ops.Open(
                         print_device,
                         Device_Defs.output);

while not
    Byte_Stream_AM.Ops.At_end_of_file(open_source)
    loop
  bytes_read := Byte_Stream_AM.Ops.Read(
      opened_dev => open_source,
      buffer_VA  => buffer'address,
      length     => buffer_size);

  Byte_Stream_AM.Ops.Write(
      opened_dev => open_print,
      buffer_VA  => buffer'address,
      length     => bytes_read);
 end loop;

Byte_Stream_AM.Ops.Close(open_source);
Byte_Stream_AM.Ops.Close(open_print);
```

When the application issues a `Close` call, the printer is reallocated in the spool queue printer list and again becomes available for spooled or direct printing.

The number of copies of the spool file that will be printed defaults to 1. The number of copies may be changed by calling `Spool_Device_Mgt.Ops.Set_copy_number` with the desired number of copies.

IV-6.2.2 Printing Directly to a Printer

When a print device is created for direct printing (the `print_mode` parameter in `Spool_Device_Mgt.Create_print_device` is set to `page_wise` or `line_wise`), an `Open` call establishes a direct connection with a printer from the print device's printer list.

```
elsif Spool_Defs.Is_print_device(on_untyped) then
  print_device :=
      Spool_Device_Mgt.Create_print_device(
          spool_queue => spool_queue,
          pixel_units => false,
          print_area  => sheet_size,
          print_mode  => Spool_Defs.page_wise);
            -- direct printing
```

IV-6.2.3 Controlling Print Properties

Calls Used:

`Spool_Device_Mgt.Set_page_output_attributes`
Sets page output attributes.

`Spool_Device_Mgt.Get_page_output_attributes`
Returns the page output attributes associated with a print device.

These calls permit query and modification of page output parameters for a print device. These parameters include:

- Line feed and carriage return mapping
- Mapping of control characters
- Scrolling
- Linewrap
- Print color (if the printer is capable of color).

The first four parameters default to true and the print color defaults to `black` when a print device is created. The settings of these parameters can be queried and reset.

The spool service and print service conform to the standard printing model in which:

- Linewrap is set so that text is never lost.
- Line feeds and carriage returns are mapped to the line feed/carriage return combination.
- Control characters are printed as a two-character sequence (`^<character>`).
- A page advance is performed when the current print position is in the last column of the last row.
- Text color, if applicable, is set black.

Table IV-6-2 lists other calls that control properties with brief descriptions of their uses:

Table IV-6-2. Getting and Setting Print Properties

Property	How to Get and Set It
Get the current print position and print area size.	`Spool_Device_Mgt.Ops.Get_print_area_position` `Spool_Device_Mgt.Ops.Get_print_area_size`
Set and check the completed number of copies of a print job.	`Spool_Device_Mgt.Ops.Set_copy_number` `Spool_Device_Mgt.Ops.Get_print_count`
Get a list of printers available for a spool queue, spool file, or print device.	`Spool_Device_Mgt.Ops.Get_printer_list`
Get a device's sheet request.	`Spool_Device_Mgt.Ops.Get_sheet_request`
Get a device's character size.	`Spool_Device_Mgt.Ops.Get_standard_character_size`
Check whether sizes are measured in pixels or character units.	`Spool_Device_Mgt.Ops.Is_pixel_units`

Get and set a spool queue's print delay properties.	`Spool_Queue_Admin.Get_delay_class`
	`Spool_Queue_Admin.Get_print_delay`
	`Spool_Queue_Admin.Set_print_delay`

Property	How to Get and Set It
Get and set the priority assigned to a spool queue.	`Spool_Queue_Admin.Get_priority` `Spool_Queue_Admin.Set_priority`
Get the request class for a spool file, printer, or print device.	`Spool_Device_Mgt.Ops.Get_request_class`
Get the printinfo associated with a spool queue.	`Spool_Queue_Admin.Get_printinfo`
Get and set a printinfo.	`Printer_Admin.Get_printer_type` `Printer_Admin.Set_printer_type`

IV-6.2.4 Administering Spool Devices

Table IV-6-3 lists calls that inquire about and set the states of spool devices, enable and disable spool device capabilities, install a spool queue and move spool files.

Table IV-6-3. Executing Print and Spool Tasks

Task	How to do it

Check what spool device an AD references.	`Spool_Defs.Is_emulation`
	`Is_print_device`
	`Is_printer`
	`Is_printinfo`
	`Is_spool_file`
	`Is_spool_queue`

Get and set the emulation in effect for a print device.	`Spool_Device_Mgt.Get_emulation` `Set_emulation`
Get the output device for a print device (spool file for spooled printing or a printer for direct printing.)	`Spool_Device_Mgt.Get_output_device`
Get a print device's print mode.	`Spool_Device_Mgt.Get_print_mode`
Get a print device's associated spool queue.	`Spool_Device_Mgt.Get_spool_queue`
Stop and disable printing.	`Spool_Device_Mgt.Ops.Abort_printing`
Delete a spool queue, print device, or spool file.	`Spool_Device_Mgt.Ops.Delete_device`
Enable or disable automatic deletion of printed spool files.	`Spool_Device_Mgt.Ops.Enable_deleting` `Spool_Device_Mgt.Ops.Disable_deleting`
Enable or disable printing by a spool file, spool queue, or printer.	`Spool_Device_Mgt.Ops.Enable_printing` `Spool_Device_Mgt.Ops.Disable_printing`

Print or don't print a banner page, and set a banner page line.	`Spool_Device_Mgt.Ops.Enable_banner_page`
	`Spool_Device_Mgt.Ops.Disable_banner_page`
	`Spool_Device_Mgt.Ops.Set_banner_page_line`

Set, print or don't print a termination message.	`Spool_Device_Mgt.Ops.Enable_termination_message`
	`Spool_Device_Mgt.Ops.Disable_termination_message`
	`Spool_Device_Mgt.Ops.Set_termination_message`

Associate a formal parameter with a text string.	`Spool_Device_Mgt.Ops.Set_parameter_text`
Determine supported access methods.	`Spool_Device_Mgt.Ops.Get_access_method_support`
Determine a spool device's status.	`Spool_Device_Mgt.Ops.Get_device_status`
Get a list of the attribute calls supported by a device.	`Spool_Device_Mgt.Ops.Get_operation_support`

Check enable or disable status.	`Spool_Device_Mgt.Ops.Is_banner_page_enabled`
	`Is_deleting_enabled`
	`Is_printing_enabled`
	`Is_termination_message_enabled`

Check whether a device supports graphics.	`Spool_Device_Mgt.Ops.Is_graphic_device`

Task	How to do it
Enable or disable spooling.	`Spool_Queue_Admin.Enable_spooling` `Spool_Queue_Admin.Enable_spooling`
Get a list of spool files in print order.	`Spool_Queue_Admin.Get_rank_list`
Install a spool queue.	`Spool_Queue_Admin.Install`
Determine if a spool queue contains any spool files, and if spooling is enabled.	`Spool_Queue_Admin.Is_empty` `Spool_Queue_Admin.Is_spooling_enabled`
Change the order of printing for a spool file.	`Spool_Queue_Admin.Modify_rank`
Move a spool file to another spool queue.	`Spool_Queue_Admin.Move_spool_file`

IV-6.2.5 Adding a New Printer

Calls Used:

`Printer_Admin.Set_printer_type`
Associates a printinfo with a printer.

`Spool_Device_Mgt.Ops.Connect_printer`
Connects a printer to a spool queue, or selects a printer from the spool queue's printer list to be stored in the printer list of a print device or spool file.

The process of adding a new printer requires an existing printinfo for the type of printer to be added, and a printer object. Then, the printer can be associated with its printinfo with `Printer_Admin.Set_printer_type`, and the printer can be added to a printer list with `Spool_Device_Mgt.Ops.Connect_printer`.

IV-6.3 Summary

- A *spool queue* is an instance of a spool device class and is required for all printing.
- A print job may be spooled or sent directly to a printer.
- A *print device* must be created before writing a spool file or printing directly.
- A print device opened in `Spool_Defs.print_mode_t.spooled` mode will send the print output to a *spool file*.
- A print device opened in `Spool_Defs.print_mode_t.line_wise` or `Spool_Defs.print_mode_t.page_wise` mode will send the print output directly to a printer.
- *Printinfo* entries contain printer capabilities corresponding to printers with the same capabilities.
- Lists of printers available for spooled and direct printing are maintained in spool queues, print devices, and spool files.
- The *print area* defines the area on a sheet or form where output may be printed.
- The *print position* is the location within the print area where printing will next commence.

- A *banner page* is an optional page which may be printed to identify a print job.
- A *print termination message* is an optional message which may be sent to the user who created the print device to signal completion of a print job.
- Applications whose output must print on a certain form or specific size of sheet may request that the needed form or sheet be mounted before printing.
- Print properties are maintained in spool queues, spool files, and print devices.
- Printing can be explicitly delayed for a spool queue until a certain time, or until a certain time if the size of the print job exceeds a limit.

UNDERSTANDING STRUCTURED FILES 7

Contents

This chapter is an overview of the concepts and terminology for *structured files*. This chapter does not present any programming techniques.

Packages Used:

`Data_Definition_Mgt`	Manages field and record data definition (DDef) creation for structured files.
`Field_Access`	Provides buffer access to fields in records that reference DDefs.
`File_Admin`	Provides calls and declarations for managing structured files.
`File_Defs`	Provides declarations used for filing operations.
`Join_Interface`	Provides database support for joins of indexed files or record stream devices. This package is only available to trusted type managers.
`Record_AM`	Provides device-independent I/O for accessing records one record at a time. Contains the `Record_AM.Ops` and `Record_AM.Keyed_Ops` packages.
`Record_AM.Ops`	Provides a common interface for record I/O calls.
`Record_AM.Keyed_Ops`	Provides record I/O calls for indexed files.
`Record_Processing_Support`	Provides calls for processing collections of records.
`Sort_Merge_Interface`	Provides calls for sorting and merging records.
`Trusted_Record_Processing_Support`	Provides calls for processing collections of records using user-supplied routines. This package is only available to trusted type managers.

The following chapters explain more about using structured files:

- Chapter IV-8 explains how to use the filing and record I/O calls to build indexes in structured files.
- Chapter IV-9 explains how to use record I/O to access structured files with indexes.
- Chapter IV-10 explains how to use locking to control concurrent access to structured files.
- Chapter IV-11 explains how to use the record processing and database support packages to process collections of records.

These chapters further explain the concepts in this chapter, and present programming techniques for using them.

IV-7.1 Stream Files and Structured Files

The filing service provides two kinds of files: *stream files* and *structured files*. Stream files contain a stream of contiguous bytes and allow random byte access within file. You usually use byte stream I/O to read and write stream files. Stream files cannot have indexes.

Structured files contain a collection of records having a common structure. These files can have indexes and allow record positioning within the file. You usually use record I/O with structured files, for reading and writing records.

This chapter describes structured files. Chapter IV-3 describes stream files.

IV-7.1.1 Data Areas

Files are represented by a *file object* that can be accessed through a *file AD*. The file's contents reside in several data areas. A structured file has a *primary data area* and can have one or more *secondary data areas*. Figure IV-7-1 shows a file AD to a structured file object with primary and secondary data areas.

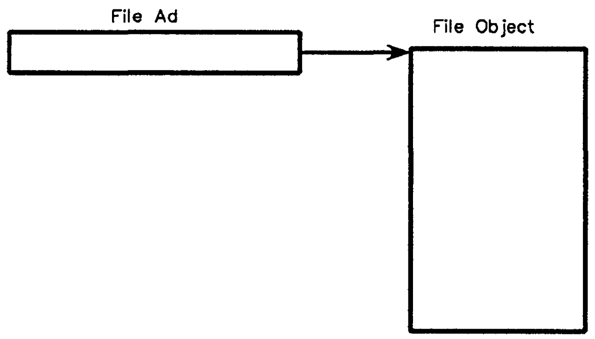


Figure IV-7-1. File Objects and Data Areas

A structured file has one primary data area, and it contains the file's data. A structured file can have up to 16 secondary data areas. Secondary data areas contain indexes.

IV-7.2 Records

Record are named collections of data having *fields* to hold the data. Fields can contain any type of data except ADs. Records can have two formats:

Fixed-length records
: Each field in the record has a fixed size.

Variable-length records
: One or more fields in the record vary in size.

A record has a size that you can specify when you create a structured file. The maximum size of a record can vary depending on your *file organization*. The maximum record size for records in sequential files is 16 megabytes. The record size is limited to the size of the bucket (minus the bucket overhead) for unordered, clustered, and hashed file organizations.

Each record in a file has a *record ID* that provides access to the record's physical location in a file. A record gets a unique ID when it is inserted in a file. Record IDs can only change when the key value the file's *organization index* changes. You can get a record ID when inserting or reading a record.

Records in relative files have a *record number* for quick record access. In a relative file, each record in the file has a fixed position with a corresponding unique number. When a record is inserted in the file, the record receives a unique record number. This record number does not change.

IV-7.3 Buckets

The filing service divides the primary and secondary data areas of structured files into a number of fixed-sized *buckets*. The filing service uses buckets to transfer records between active and passive store. A bucket can hold many records that can span across multiple buckets depending on the file organization. Some file organizations allow you to choose the bucket size.

IV-7.4 Indexes

Indexes provide fast access to structured files. An index can either be an *organization index* or an *alternate index*. An organization index defines the organization of the primary data for clustered and hashed files. When creating a file, you create an organization index which you cannot delete or deactivate while the file exists. An alternate index does not affect the organization of the primary data area. After creating a file, you can add alternate indexes which you can destroy, deactivate, or reorganize at any time. Up to fifteen alternate indexes can be added to a hashed or clustered file. Up to sixteen alternate indexes can be added to a sequential, relative, or unordered file.

An index can have one of these structures:

B-tree	Uses a b-tree data structure to organize an index's record key values.
Hashed	Uses a hashing function to index records.

B-tree indexes contain record key values in a b-tree. Searching for particular key values compares key values at each level of the tree until the correct key value is found. Key values map to record IDs that provide access to a specific record in the file's primary data area. This structure is particularly suited for indexed-sequential record access, for example, scanning all employee records from `G` through `P`.

A hashed index uses a hashing function to index records. The hashing function lets you access any record in the file quickly, for example, reading a specific employee record. An indexed-random read typically takes one disk access with hashed indexes. Indexed-sequential reads are not possible using this index.

Chapter IV-8 provides more information about using indexes with structured files.

IV-7.5 Structured File Organizations

Each structured file has a file organization that dictates how records are stored in the file. A structured file can have one of these organizations:

clustered	Records are organized in related groups (clusters) according to a clustering b-tree organization index.
hashed	Records are organized according to a hashed organization index.
relative	Records are organized in an array of fixed-size record slots.

sequential	Records are organized in the sequence in which they are inserted.
unordered	Records are organized according to available free space.

IV-7.5.1 Sequential Files

A sequential file is a stream of formatted records where the logical and physical order of the records are the same. Figure IV-7-2 shows a sequential file.

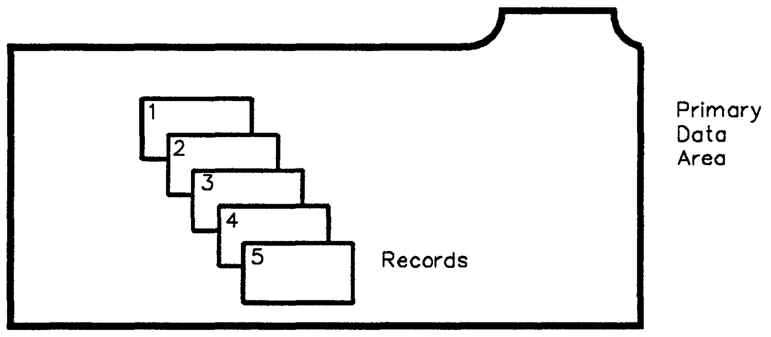


Figure IV-7-2. Sequential File

Sequential files do not depend on a termination character to read a record because the filing service already knows the structure of the record. Sequential files are efficient for serial or sequential processing of records, where you position reading and writing by retrieving the next record. Major characteristics of sequential files include:

- You cannot delete records in sequential file.
- You can update records as long as the updated record is the same size as the original record.
- The filing service automatically inserts records at the end of the file (useful for history or log files).
- Records can be indexed using up to 15 b-tree alternate indexes in separate secondary data areas.
- The maximum record size in primary data is 16 megabytes.
- Variable length records are not restricted to a bucket size.
- Indexes are allowed.
- There is no secondary data area unless the file has alternate indexes.
- The primary bucket size is always 4 kilobytes.

IV-7.5.2 Relative Files

A relative file consists of a sequence of fixed-size *record slots*; each slot can be empty or can contain one record. Figure IV-7-3 shows a relative file.

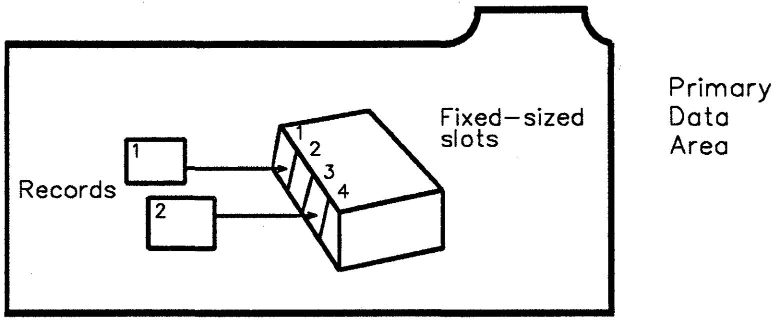


Figure IV-7-3. Relative File

Relative files let you randomly access a record by record number or record ID. This makes relative files more efficient for random access than sequential files. The characteristics of relative files include:

- You can insert and delete records.
- Records with varying lengths can change size only if the record does not exceed the size of the record slot.
- You can access records by record number.
- Indexes are allowed.
- The primary bucket size is always four kilobytes.
- You must explicitly handle free space for relative files.

IV-7.5.3 Unordered Files

An unordered file organizes records in the primary data area according to available free space. Figure IV-7-4 shows an unordered file.

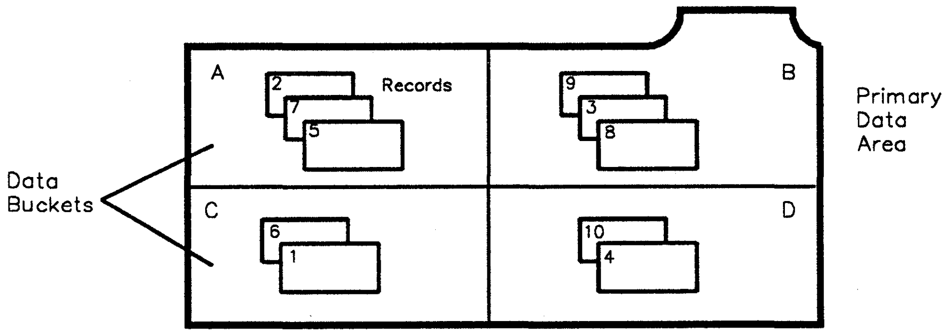


Figure IV-7-4. Unordered File

In unordered files there are no organization indexes; the system determines the record ordering. Unordered files are for applications that require concurrent record inserts.

The characteristics of unordered files include:

- Records can be deleted and updated.
- Unordered files works well for files with records of varying lengths.
- Favors concurrent record inserts.
- The maximum primary record size is limited to the size of the bucket minus bucket overhead.
- Unordered files pack variable-length records efficiently.

IV-7.5.4 Clustered Files

A clustered file organization provides fast indexed-sequential access to records. The organization index influences the placement of records in a clustered file. Figure IV-7-5 shows a clustered file.

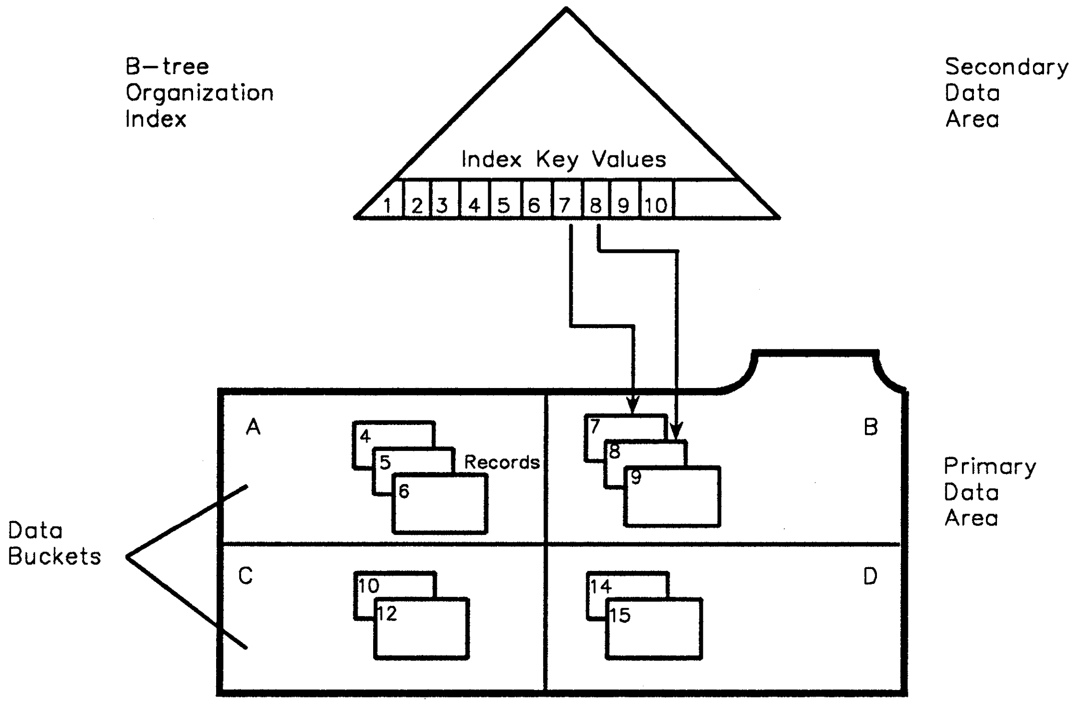


Figure IV-7-5. Clustered File

Records in the file's primary data area are *clustered*, meaning the filing service physically stores records with similar key values near each other. The characteristics of clustered files include:

- Records can be inserted, updated, and deleted.
- Updated records can change size.
- Random access is allowed.
- The organization index must be a b-tree index. (Note that the first index created on a file must be an organization index.)
- You can change key values in the organization index at any time.

- You can index records with up to fifteen alternate indexes. You define each alternate index in a separate secondary data area.
- The maximum primary record size is limited to the size of the bucket minus any bucket overhead.

IV-7.5.5 Hashed Files

Hashed files are designed for indexed-random access via an organization key. Like a clustered file, the organization index influences the placement of records in a hashed file. Figure IV-7-6 shows a hashed file.

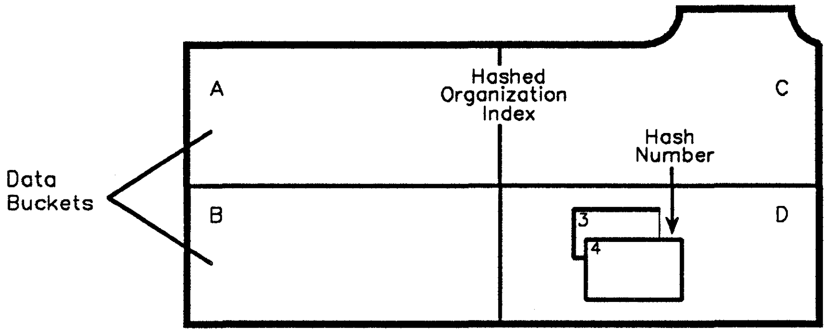


Figure IV-7-6. Hashed File

In a hashed file organization, the filing service stores records and keys in the file's primary data area; there is no secondary data area. The filing service organizes a hashed file using a hashing function that produces hashed values for record positioning in the primary data area. The characteristics of hashed files include:

- Records can be inserted, updated, and deleted.
- Updated records can change size.
- The organization index must be hashed.
- Random reads via the organization key usually require only a single disk access.
- Index-sequential access via the organization key is only possible for reading duplicate sequences.
- The maximum primary record size is limited to the size of the bucket minus bucket overhead.

IV-7.5.6 File Descriptors

File descriptors contain information about a file's logical and physical characteristics. There are three possible descriptors for a file:

Logical file descriptor
Contains options for selecting a structured file's record formats, record DDefs, transaction locking, and logging. See the `File_Admin` package for the logical file descriptor record.

Physical file descriptor
Contains information about a structured file's *volume layout.* See the `File_Admin` package for the physical file descriptor record.

Index descriptor	Contains information for an index in a structured file. A file can have several indexes and several index descriptors. See the `File_Defs` package for the specification of the index descriptor record.

You *must* use a logical file descriptor to create a file, but you only use a physical file descriptor to control the file's disk space and volume set allocation. You use index descriptors to create indexes in a file.

IV-7.6 Using Byte Stream and Record I/O with Files

You normally use byte stream I/O to access stream files and record I/O to access structured files. But structured files support byte stream and record I/O. Table IV-7-1 summarizes the differences between stream and structured file access.

Table IV-7-1. Accessing Stream and Structured Files

	`Byte_Stream_AM`	`Record_AM`
Stream	Read and write bytes	Termination character appended during insert. Read record up to next termination character.
Structured	Read bytes of a record. No writes allowed.	Read and insert records.

With stream files, you can read or write the entire contents of the file. Using byte stream I/O, you can read but not write structured files. This protects against inadvertent modification of the file's record structure. When reading a structured file using byte stream I/O, all internal control information and deleted records are filtered out.

You usually use record I/O to access a record-oriented device such as a file, pipe, or directory one record at a time. There are four *record access modes*:

physical-sequential	Sequential access according to the physical sequence of records in the file.
physical-random	Random access to records using physical positioning.
indexed-sequential	Sequential access to an indexed file in key sequence starting with any key value in an index.
indexed-random	Random access to records according to a record's key value.

You only use physical-sequential and physical-random modes to access structured files that do not have indexes. You can use any access mode to access structured files *with* indexes. Table IV-7-2 shows access modes for files.

Table IV-7-2. File Access Modes

	Stream	Seqential	Relative	Unordered	Clustered	Hashed
Physical-Sequential	x	x	x	x	x	x
Physical-Random	x	x	x	x	x	x
Index-Sequential (Organization Index)					x	x
Index-Sequential (Alternate Index)		x	x	x	x	x
Index-Random (Organization Index)					x	x
Index-Random (Alternate Index)		x	x	x	x	x

Chapter IV-9 discusses in more detail how to accessing files with record I/O.

IV-7.7 Structured Files and Transactions

Transactions can lock indexes, records, and structured files to synchronize concurrency during file access operations. Chapter IV-10 provides information about locking structured files.

IV-7.8 Summary

- Structured files are collections of records having a similar structure.
- You usually use record I/O to access structured files.
- The filing service always stores a file's data in a primary data area.
- Records are named collections of data having *fields* to hold the data.
- Indexes provide fast access to records in files.
- Organization indexes influence the placement of records in the primary data area.
- Alternate indexes do not influence the placement of records in the primary data area, and are *optional* for all structured file organizations.
- Structured files can be indexed with either b-tree or hashed indexes.
- The filing service provides five structured file organizations. Each structured file organization has characteristics that make them suitable for particular applications.
- There are three descriptors that can be defined for a file: *logical*, *physical*, and *index*.

MANAGING FILES AND INDEXES 8

Contents

This chapter provides the concepts and techniques for managing structured files and indexes.

Packages Used:

`File_Defs`	Provides declarations for filing and indexing.
`File_Admin`	Administers files.
`Data_Definition_Mgt`	Manages the data definitions (DDefs) for creating records, fields, and index keys.

IV-8.1 Concepts

Indexes provide fast access to records in structured files. To create an index, you define its structure and key definition. You use DDefs to define an index structure and the index key definition. Figure IV-8-1 shows an index for a file.

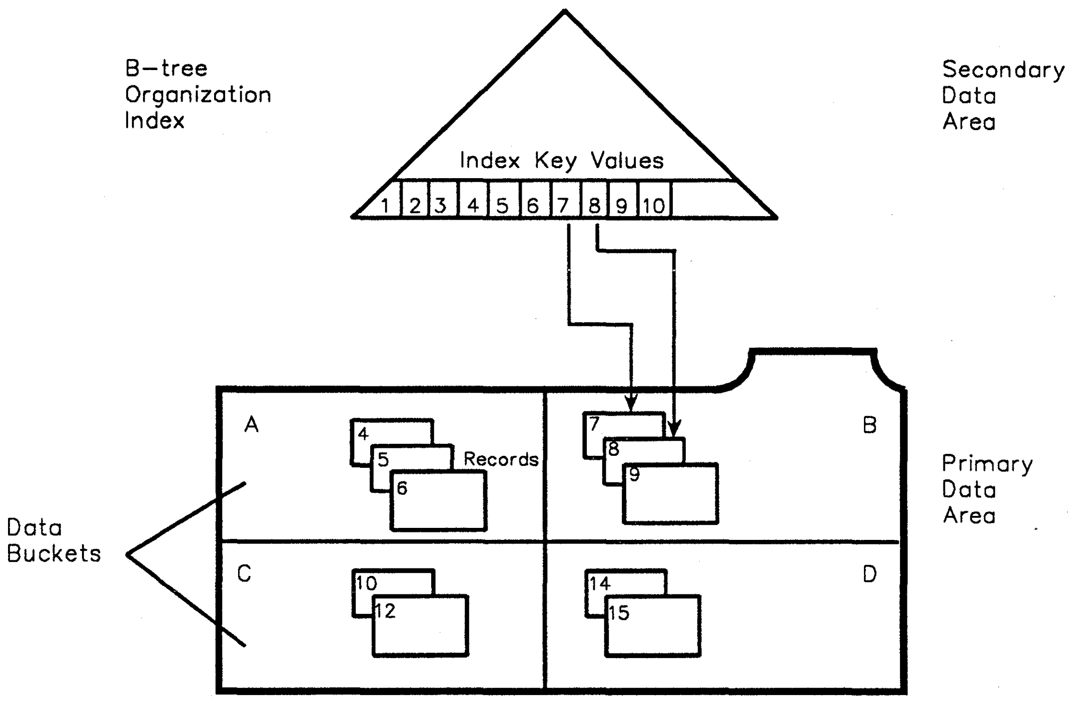


Figure IV-8-1. An Indexed File

The index has a *b-tree organization,* is located in the *secondary data area* of the file, and uses key values to reference records in the *primary data area.*

IV-8.1.1 Index Keys

Within an index, you can define particular record fields as *keys*. Keys let you identify particular records, order records in a file, or specify records you want to retrieve or update. You can order records in a key by ascending or descending order based on the *key value* in each record. Keys are efficient for reading and retrieving records in a file, but are less efficient for inserting updating, and deleting records.

If a key value uniquely identifies a record, the key is a *primary key*. For example, the `name` key for an employee record uniquely identifies an employee. Keys whose values do not uniquely identify a record are *secondary keys*. For example, the key `dept` in an employee record does not uniquely identify an employee record. Figure IV-8-2 shows a representation of an index with key values.

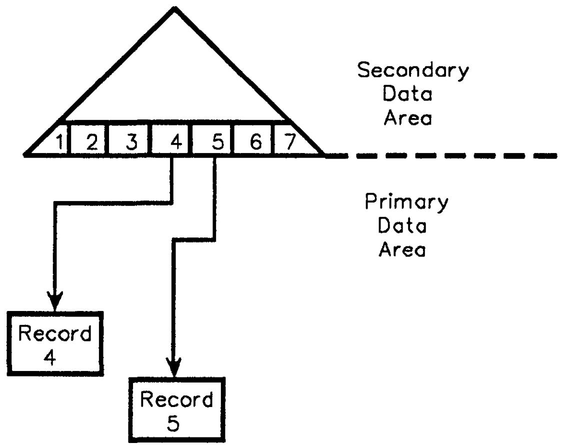


Figure IV-8-2. Index Key Values that Point to Records

The calls in the `Record_AM.Keyed_Ops` package provide specific *keyed operations* for indexed access to structured files.

IV-8.1.2 Index Structures

There are three index structures:

- *B-tree alternate indexes* are located in secondary data areas of structured files. These indexes do not affect the file organization.
- *B-tree organization indexes* organize records in the primary data area based on the sort sequence of an organization key.
- *Hashed organization indexes* organize records in the primary data based on a hash function that determines the record locations.

IV-8.1.2.1 B-Tree Alternate Index

You can build b-tree alternate indexes on any structured file B-tree alternate indexes do not reorganize a file, which means you can delete or deactivate them without destroying the file. These indexes support fast sequential access and moderately fast random access. Figure IV-8-3 shows a b-tree alternate index.

B-tree alternate indexes:

- Exist in the secondary data area
- Contain record key values in the b-tree structure.

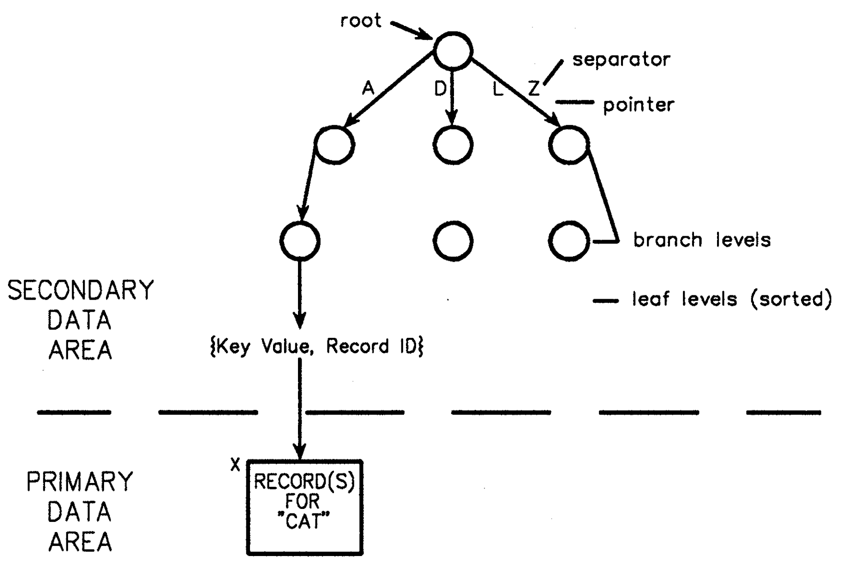


Figure IV-8-3. B-Tree Alternate Index

IV-8.1.2.2 B-Tree Organization Index

A b-tree organization index is used to organize clustered files. You can only delete or deactivate a b-tree organization index by destroying the file. Figure IV-8-4 shows a b-tree organization index for a clustered file.

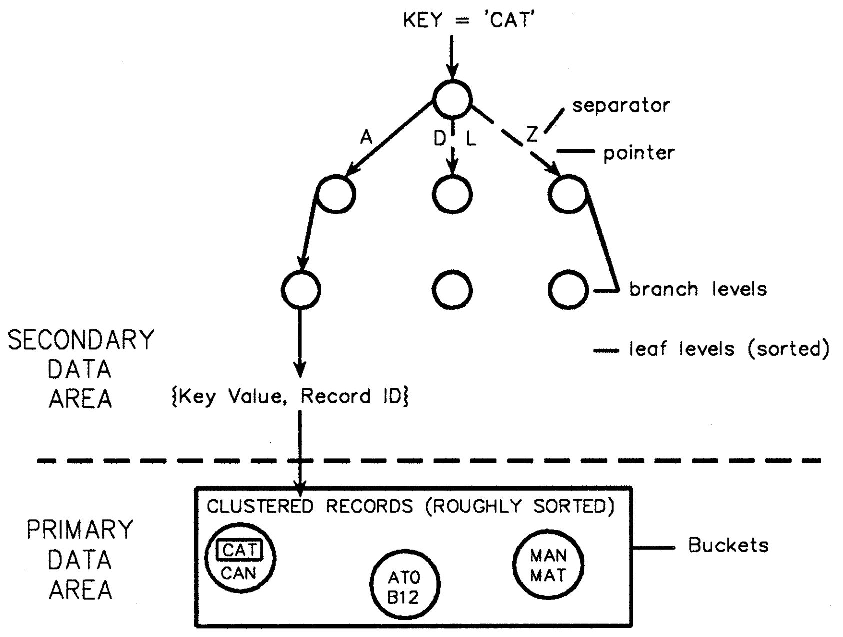


Figure IV-8-4. Clustering B-Tree Organization Index

B-tree organization indexes:

- Have buckets that contain the key portion of the record
- Are not strictly sorted
- Indexed-sequential access via the organization key is very fast for large scans.

IV-8.1.2.3 Hashed Organization Index

You use hashed organization indexes with hashed files for fast random access to key values. Figure IV-8-5 shows a hashed organization index for a hashed file.

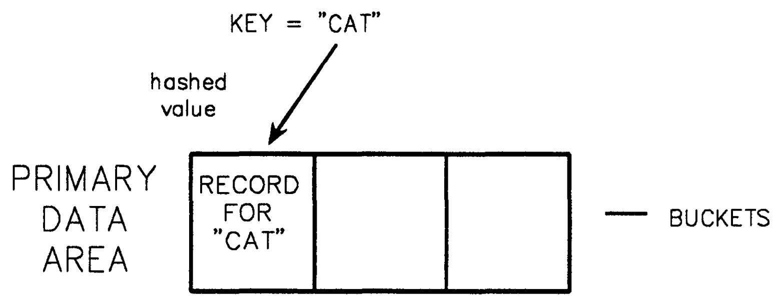


Figure IV-8-5. Hashed Organization Index

Hashed organization indexes:

- Only uses the primary data area.
- Uses a hashing function to provide quick access to any record in the file.
- Except for duplicate key values, are not accessible via indexed-sequential access.
- Do not reorganize the primary data area of hashed files.

IV-8.1.3 Choosing Indexes

Table IV-8-1 lists index performance considerations by application type:

Table IV-8-1. Index Performance Considerations

Application	B-Tree Alternate Index	Clustered B-tree Org Index	Hash Org Index
Full-ordered scan in sorted order -- employees G to T	Good	Excellent	Not available
Online random access -- employee Albert Einstein	Satisfactory	Satisfactory	Excellent
Update-intensive -- key value changes	Satisfactory	Poor	Satisfactory
Insert-intensive	Satisfactory	*	Satisfactory

* *In a b-tree index for a clustered file, insert-intensive operations cause the average access time to degrade after the initial reserved space is used up (if the proper fill-factor is selected), and after initial loading or reorganization.*

IV-8.1.4 Record DDefs

You use *data definitions* (DDefs) to define the data layouts for structured files. If you want to use indexes, you must use DDefs. *Record DDefs* specify a complete DDef for a record. You can use *field DDefs* to specify each record field so you can logically group the field DDefs to create record DDefs. Field DDefs have the advantage that you can reuse them to create other record DDefs in your application. You define record and field DDefs using the `Data_Definition_Mgt` package.

NOTE

If you do not use DDefs for record layouts, you have to maintain your own record layout. (Ada users use *record types*; C users use *structures*.) The filing service does not restrict using a single field for multiple keys, overlapping fields, or having the same record portion appear in different fields with different field types.

Before creating a structured file, you create a record DDef that defines the file's record structure. The record DDef includes the record's:

- Alignment
- Size
- Field data types.

Figure IV-8-6 shows a record DDef for an employee file.

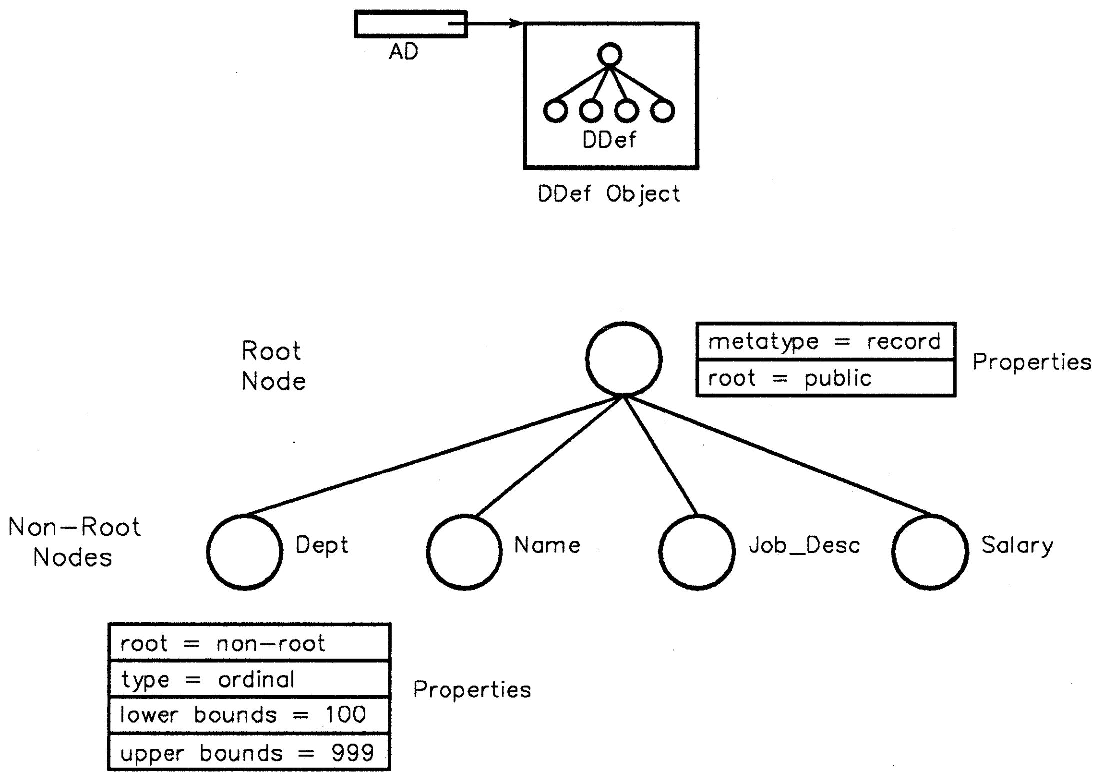


Figure IV-8-6. A Simple Record DDef

In this figure, the DDef consists of several nodes including:

- A *root* node that represents the DDef corresponding to a record with fields
- The *non-root* nodes that represent the record's fields.

To create a DDef node, you create the node and add property values to the node. A node's property values include the node's size, length, and data type.

To create a record DDef, you use the `Data_Definition_Mgt` calls in the following order:

1. Make a `Create_DDef` call to create a DDef object.
2. Make a `Create_node` call specifying `metatype = record` to create a DDef node inside the DDef object.
3. Define a DDef field.
   a. Make a `Create_simple_field` call to create the field.
   b. Make an `Add_property_value` call to define the property values for the field.

   Repeat this step until all the DDef fields are created.
4. Make a `Close` call to close and bind the DDef.

When you create a record DDef, you can derive the DDef from individual field DDefs. You can also define a record layout to ensure that the record DDef and the record layout correspond. You do this using an Ada representation clause. See section IV-8.2 for examples of setting up record DDefs.

You use `DDef_specified` field in a logical file descriptor to specify the record DDef that defines a structured file's record layout. See the `Data_Definiton_Mgt` package in the *BiiN™/OS Reference Manual* for more information about DDefs.

IV-8.1.5 Index Key DDefs

The *index key DDef* defines the record fields that make up a key value for a structured file. An index key DDef describes either:

- A *single* key built on a single field (for example, the `Dept` key has one field, department)
- A *composite* key built on two or more fields (for example, the `Dept_Salary` key has two fields, department and salary).

You can use index key DDefs to specify whether values for an index field are sorted in ascending or descending order. For example, the department field could be sorted in ascending order from department `1` to department `500`, and the salary field could be sorted in descending order from `10,000` to `1,000`. An index key DDef can take one of two forms:

Derived	The index key DDef is derived from a record DDef.
Non-derived	The index key DDef is created separately.

Figure IV-8-7 shows an index key DDef built on the department field in an employee file.

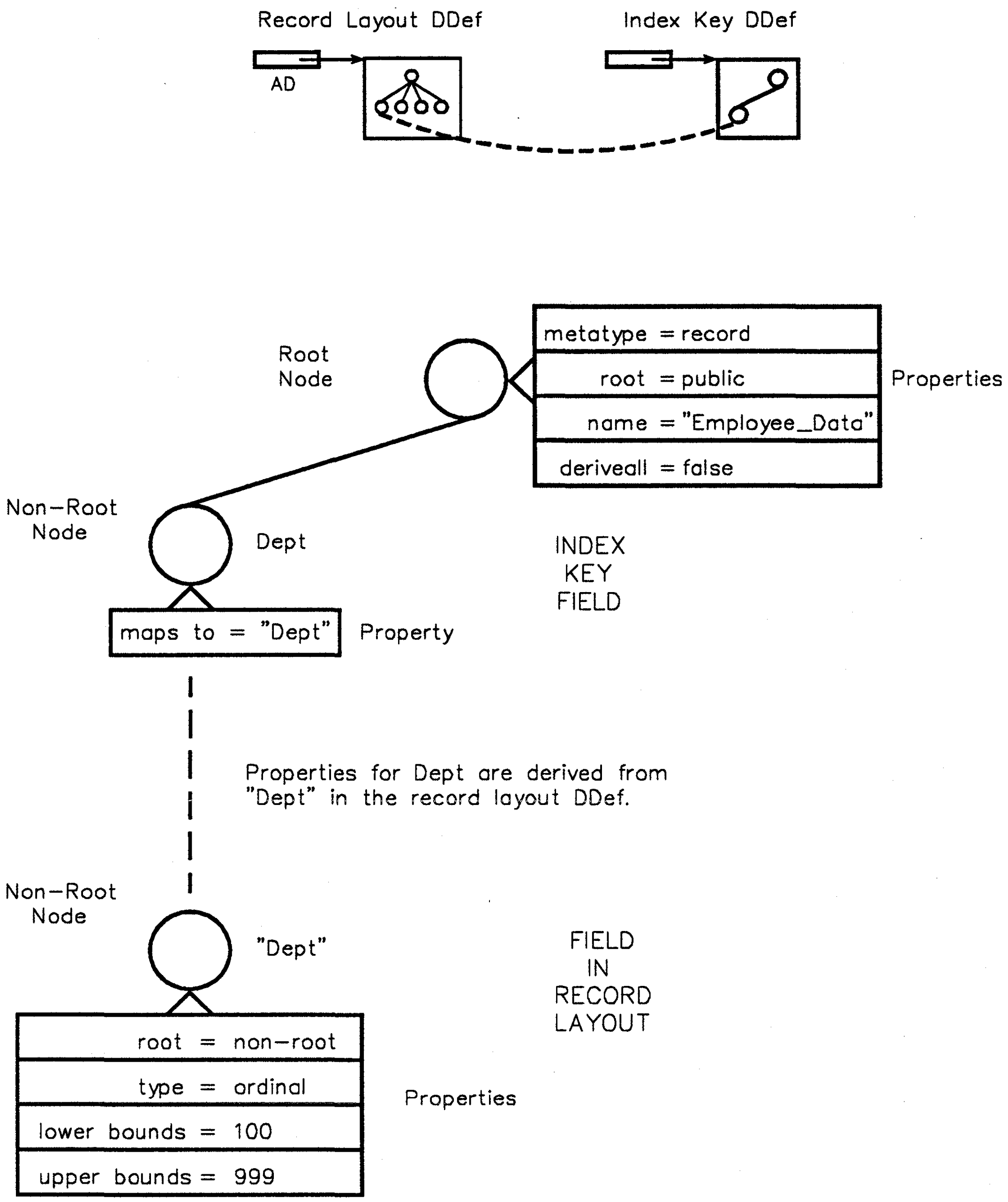


Figure IV-8-7. A Simple Derived Index Key DDef

You can derive an index key DDef from one or more of the record DDef's nodes, using the `Data_Definition_Mgt` calls as follows:

1. Make a `Create_DDef` call to create a new DDef object. This DDef is separate from the record DDef.
2. Make a `Create_node` call to create a record DDef node inside the DDef object.
3. Derive a DDef field from the record DDef.
   a. Make a `Create_field` call to create the field.

b. Make an `Add_property_value` call to define the property values for the field.

These calls create a field DDef node that references your existing field definitions (in the record DDef used for your file's record layout).

Repeat this step until all the DDef fields are created for your index key.

4. Make a `Close` call to close and bind the DDef. This computes field sizes and positions.

Figure IV-8-8 shows the layout of a record DDef and an index key DDef from the record DDef.

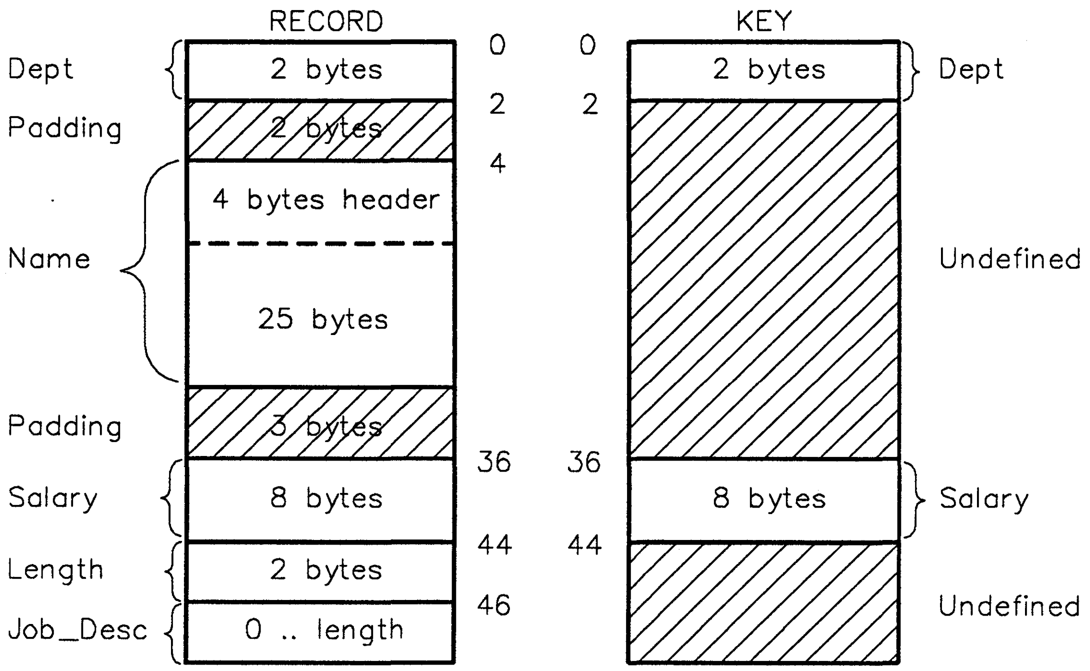


Figure IV-8-8. Layout of a Derived Index Key DDef

NOTE

The filing service adds the padding in the record layout for proper alignment of the record fields. The index key DDef uses the same image as the record DDef.

IV-8.1.6 Null Values

A *null values* is the highest possible value you can give a field. You can use null values to write efficient queries, because the filing service groups all keys with null values together. This provides a faster search algorithm for these fields. For example, a query for "all the employees in department 10 whose salary is null" can be very efficient for an index on department and salary.

The `null_attribute` in an index descriptor defines how an index handles null values. The `null_attribute` has three options:

`none`	Indicates an index does not allow null values.

`index_null`	Indicates an index allows null values, and the filing service enters them in the index structure.
`no_index_null`	Indicates the index allows null values, but the filing service does not enter them in the index structure.

Composite keys have the following rules concerning nulls:

1. Some fields can be null and others may have values. If one field is null, the entire key is *not* treated as null.
2. Multiple null values are allowed in a unique index.
3. If you use the `no_index_null` option in an index and any field in the key is null, the entry is not stored in the index.

NOTE

Organization indexes do not allow null values.

In your record layout, you need to account for any variable length fields or fields that accept null values because the format of a null field is different. If a null field is fixed, it is preceded by a one byte null indicator. If a null field is variable and it is null, the length is set to a special constant value.

IV-8.2 Techniques

This section presents the techniques for using structured files and indexes. After reading this section, you will be able to:

- Define a record DDef
- Define an index key DDef for a file
- Create a file
- Build an organization index
- Build an alternate index.

Chapter IV-3 describes the techniques for copying, emptying, and destroying files. The example package `Employee_Filing_Ex` in Appendix X-A complete code for the examples in this section.

IV-8.2.1 Defining Record DDefs

Before creating a structured file, you need to lay out the record structure for your file. The most convenient way to do this is to create a record DDef.

Calls Used:

`Data_Definition_Mgt.Create_DDef`
: Creates a DDef object to contain the record and fields layout for a record.

`Data_Definition_Mgt.Create_node`
: Creates a record DDef node inside the DDef object.

`Data_Definition_Mgt.Create_simple_field`
: Creates a field DDef node as a component of a record DDef node.

`Data_Definition_Mgt.Add_property_value`
: Adds a DDef property to a DDef node.

For example, to create a four-field layout, you must create five nodes and add the properties indicated below:

```
record DDef node:  pi_node_name is Employee_Data,
                   pi_meta_type is mt_record,
                   pi_root is public_root_node,
       ** Field nodes **

Dept:      pi_node_name is Dept,
           pi_root is non_root_node,
           pi_type is ord_2,
Name:      pi_node_name is Name,
           pi_root is non_root_node,
           pi_type is string,  (System_Defs.text)
           pi_header_for_max_length is true,
           pi_varying is true,
           pi_length is 25;
Job_Desc:  pi_node_name is Job_Descr,
           pi_root is non_root_node,
           pi_type is string,
           pi_length is 200;
Salary:    pi_node_name is Salary,
           pi_root is non_root_node,
           pi_type is real8,
           pi_default_value is 0;
```

The root DDef node is specified as equivalent to an Ada record structure (that is, it has a record *metatype*). It is declared as *public* so that nodes in other DDef objects can reference it. This is useful when deriving an index key DDef from this record DDef.

Each of the field DDef nodes is declared as a non-root node and has various properties such as data type, length, and upper/lower bounds added to it. For a complete list of these possible properties, including supported data types, see the `Data_Definition_Mgt` package.

The DDef structure shown above is equivalent to the following Ada record declaration:

```
subtype Job_Desc_length_t is
    integer range 0 .. 200;

type Employee_Data(
    Job_Desc_length: Job_Desc_length_t) is
  record
    Dept:       System.short_ordinal
                    range 100 .. 999;
    Name:       System_Defs.text(25);
    Job_Desc:   string(1 .. Job_Desc_length);
    Salary:     float;
  end record;
```

Be aware that the data definition service determines the alignment rules for each data type. Basically, these rules conform to the alignment rules for Ada with the exceptions noted in the `Data_Definition_Mgt` package. It's important that you understand these exceptions before you define your record layout.

IV-8.2.2 Defining Index Key DDefs

The easiest way to create an index key DDef is to derive it from the record DDef.

Calls Used:

`Data_Definition_Mgt.Create_DDef`
Creates a DDef object to contain the definition of an index key DDef.

`Data_Definition_Mgt.Create_node`
Creates an index key DDef node inside the containing index DDef object.

`Data_Definition_Mgt.Create_field`
Creates a field DDef node as a component of an index DDef node. This field becomes part of the key definition. The field DDef node can simply reference the definition of a field in an existing record DDef.

`Data_Definition_Mgt.Add_property_value`
Adds a DDef property (for example, a reference to a field definition in an existing record DDef) to an index key DDef node.

You derive the index key DDef from the file's existing record DDef. To derive a key using the record DDef `Employee_Data` described previously, you set up the following structure:

```
record DDef node:  pi_meta_type is mt_record,
                   pi_root is public_root_node,
                   pi_DDef_name is "Employee_DDef",
                      "Employee_Data",
                   pi_derive_all is false,
      ** Index key field nodes **
Dept:              pi_maps_to is "Dept",
Salary:            pi_maps_to is "Salary",
                   pi_descending is true,
```

This composite index key is set up by mapping DDef nodes from `Employee_Data` to a new record DDef consisting of these fields:

- `Dept` in ascending order (defaulted).
- `Salary` in descending order.

The root DDef node is specified as being derived from the structure for the "Employee_Data" record layout. It's declared as *public* so that nodes in other DDef objects can reference it. Each of the field DDef nodes is declared as referencing an existing field node. All properties associated with each field node in the file's record layout are *mapped to* the respective field node in the index key DDef.

IV-8.2.3 Creating Files

When creating a named file, you supply a pathname for the file and its logical file descriptor. The logical file descriptor contains options for selecting record formats, record DDefs, transaction locking, and logging.

Calls Used:

`File_Admin.Create_file`
Creates a permanent file.

`File_Admin.Create_unnamed_file`
Creates a temporary file that exists for the duration of the current job.

You specify a file using a logical file descriptor. The logical file descriptor contains options for selecting record formats, record DDefs, transaction locking, and logging. The `File_Admin.logical_file_descr_record` record represents a logical file descriptor. You must specify a logical file descriptor as parameter to the `File_Admin.Create_file` and `File_Admin.Create_unnamed_file` calls to create files. The following `Create_file` call creates an unordered file.

```
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));
```

To specify the file organization you use the `file_org` field in the logical file descriptor. Line 529 specifies the `file_org` parameter as `File_Defs.unordered`.

You specify a file's volume layout with a physical file descriptor. There is only one physical file descriptor for each data area of a file. The filing service provides a default physical file descriptor, so you only need to specify a physical file descriptor when you want to change the default (see the `File_Admin` package). You can specify a physical file descriptor as parameter to the `File_Admin.Create_file` and `File_Admin.Create_unnamed_file` calls when creating files. The `File_Admin.Get_physical_file_descr` call retrieves a physical file descriptor. The physical file descriptor fields you specify optionally include:

- A description of the number of bytes and volume locations of data areas
- Initial values for the expansion volume and expansion sizes of data areas.

IV-8.2.4 Building Organization Indexes

You can create indexes for any file organization. Before opening a file for indexed access, you must first create the file's organization index.

Calls Required:

`File_Admin.Build_index`
Builds an index.

You build an organization index by specifying the correct index descriptor to the `Build_index` call. You specify an index descriptor for each index a structured file has. An index descriptor specifies the index's name, the index's organization, and special options that indicate (among other options) whether an index can use duplicates, null values, and phantom protection. The data structure `File_Defs.index_descr_record` represents an index descriptor. You specify an index descriptor as parameter to the `File_Admin.Build_index` call to build an index. The `File_Admin.Get_logical_index_descr` call retrieves an index descriptor.

The following `Build_index` call builds an organization index for a unordered file.

```
534      File_Admin.Build_index(
535          file => new_file,
536          logical_index_descr => (
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));
```

Line 538 sets the index name to `dept_index_name`. Line 540 sets the index organization to `File_Defs.btree_index`. For a hashed file, you set this field to `File_Defs.hashed_index`. Line 546 sets the `DDef` field to `dept_index_DDef`, the index DDef created earlier. A file's organization index is *always* active. This assures that the organization index structure is always up-to-date so that it may be used to determine the placement of a newly inserted record.

The `File_Admin` package provides additional calls for getting information about index descriptors. These calls are:

`File_Admin.Get_index_names`
Gets the index names associated with the specified file.

`File_Admin.Get_index_status`
Gets dynamic information associated with the specified index.

IV-8.2.5 Building Alternate Indexes

You can add alternate indexes to any structured file organization. An alternate index (like an organization index) requires you to specify an index key DDef as part of the index descriptor parameter.

Calls Required:

`File_Admin.Build_index`
Builds an index.

You build an alternate index on a previously created file. You specify an index descriptor using `File_Defs.index_descr_record`. You provide the index descriptor as a parameter to `File_Admin.Build_index`. The following example defines a department-salary index that uses a composite key built on department and name fields.

```
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));
```

Line 556 sets the index name to `dept_salary_index_name`. Line 559 sets the index organization to `File_Defs.btree_index`, which is necessary for a unordered file. Line 568 sets the `DDef` field is set to `dept_salary_index_DDef`, the index DDef created earlier.

This index is a composite index. Composite keys are sorted field by field in prefix order. The greater-than comparison of descending fields is reversed so that their key entries are stored in reverse order in the index structure.

The `File_Admin` package provides additional calls for destroying, deactivating, and reorganizing alternate indexes. These calls are:

`File_Admin.Destroy_index`
Destroys an alternate index.

`File_Admin.Deactivate_index`
Deactivates an alternate index.

`File_Admin.Reorganize_index`
Reorganizes an alternate index.

IV-8.3 Summary

- Use the `File_Admin` package to manage *structured files*.
- DDefs define a file's record and index layout.
- Define a single or composite index key value using a DDef. You can derive the index key DDef from a file's record DDef.
- You build indexes using the index descriptor record in the `File_Defs` package.
- You can build alternate indexes for any structured file organization.
- Only clustered and hashed file organizations can have organization indexes.
- B-tree alternate indexes provide fast indexed-sequential and moderate indexed-random access.
- B-tree organization indexes used as organization indexes for clustered files are best for index sequential access.
- Hashed organization indexes used as organization indexes for hashed files provide very fast indexed-random access.

USING RECORD I/O WITH STRUCTURED FILES 9

Contents

This chapter presents the concepts and techniques for using record I/O with *structured files.*

Packages Used:

`Record_AM`	Provides device-independent record I/O.

Record I/O lets you access records one record at a time from any system device that supports record access. This chapter emphasizes using record I/O with structured files. You can use these same techniques to access records on other system devices.

IV-9.1 Concepts

This section discusses the concepts and terminology related to using record I/O with structured files.

IV-9.1.1 Current Record Pointer

The *current record pointer* (CRP) represents the current record location in a file. Figure IV-9-1 shows the current record pointer during a read of a sequential file.

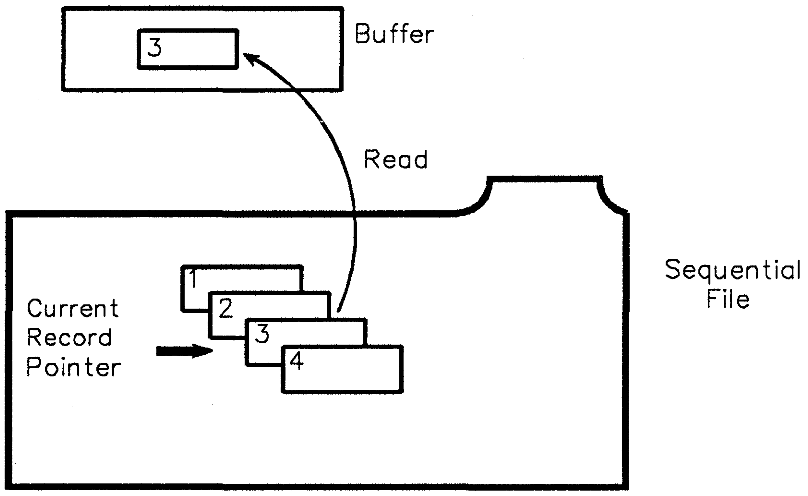


Figure IV-9-1. A Record I/O Read Operation

You use the CRP to access records by *physical* or *indexed* location. During *physical access*, the CRP points to the current record. You can use the CRP to step through a file's records in either forward or reverse sequence. You can set the position of the CRP to the first or last record in a file, a record ID, or to a record number. During *indexed access*, the CRP points to the current index value. You can set the CRP to the first, last, or any particular key value.

You can use the position modifier parameter `modifier` in record I/O calls to adjust the CRP. This parameter can be:

`current`	The current record.
`next`	The next record in the sequence of records.

`prior`	The prior record in the sequence of records.
`rest_of_current`	The rest of the current record. For reading the unread part of the current record when a `Device_Defs.length_error` is raised during a read call.

These calls modify the CRP:

```
Record_AM.Ops.Delete
Record_AM.Ops.Read
Record_AM.Ops.Set_position
Record_AM.Ops.Update.
Record_AM.Keyed_Ops.Read_key_value
```

These calls do not modify the CRP:

```
Record_AM.Keyed_Ops.Read_by_key
Record_AM.Keyed_Ops.Update_by_key
Record_AM.Keyed_Ops.Delete_by_key
```

Unsuccessful record I/O calls have no effect on the CRP.

IV-9.1.2 Access Modes

An *access mode* determines the order in which records in a file are accessed. You can access records either sequentially or randomly, and in either a physical or an indexed order. Depending on the operations you perform on a file, some access modes are more efficient. You can access structured files with one or more of these access modes:

physical-sequential	Access to a set of records by their physical order.
physical-random	Access to a single record by the record's physical address.
indexed-sequential	Access to a set of records using an index key value range.
indexed-random	Access to a single record using an index key value.

Sequential operations are always relative to the CRP. Random operations generally do not depend on the positioning of the CRP.

IV-9.1.2.1 Physical-Sequential Access

You can access any structured file using physical-sequential access. This mode is the fastest way to do full file scans. Figure IV-9-2 shows physical-sequential access.

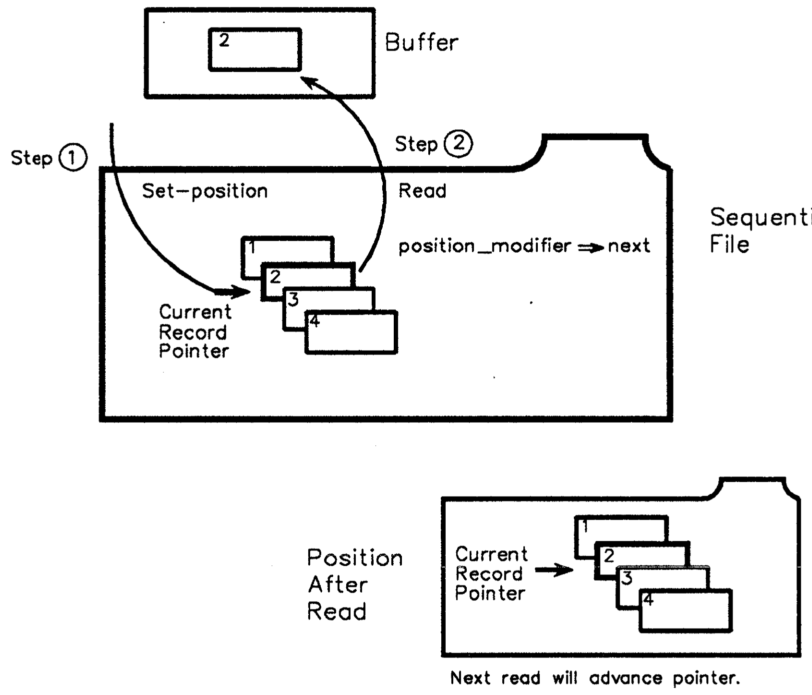


Figure IV-9-2. Physical-Sequential Access

For physical-sequential access mode, you can do the following:

- Read records in sequence and, based on the record's value, delete or update the current record.
- You read records according to the way the filing service places them physically in the file. This is the default mode.
- Set the CRP's initial position with one or more of these calls: `Record_AM.Open`, `Record_AM.Ops.Read`, or `Record_AM.Ops.Set_position`. After setting the initial position, you can read records in sequence using the position modifier.
- Do not use index ranges.

NOTE

Physical-sequential reads on unordered, clustered, and hashed files read records as they are physically located in the file.

After finishing a read operation, the CRP still points to the current record. The CRP moves just before the next read.

IV-9.1.2.2 Physical-Random Access

In physical-random access, you must precede each `Read`, `Update`, or `Delete` call with a `Set_position` call to set the position of the CRP. Figure IV-9-3 shows physical-random access.

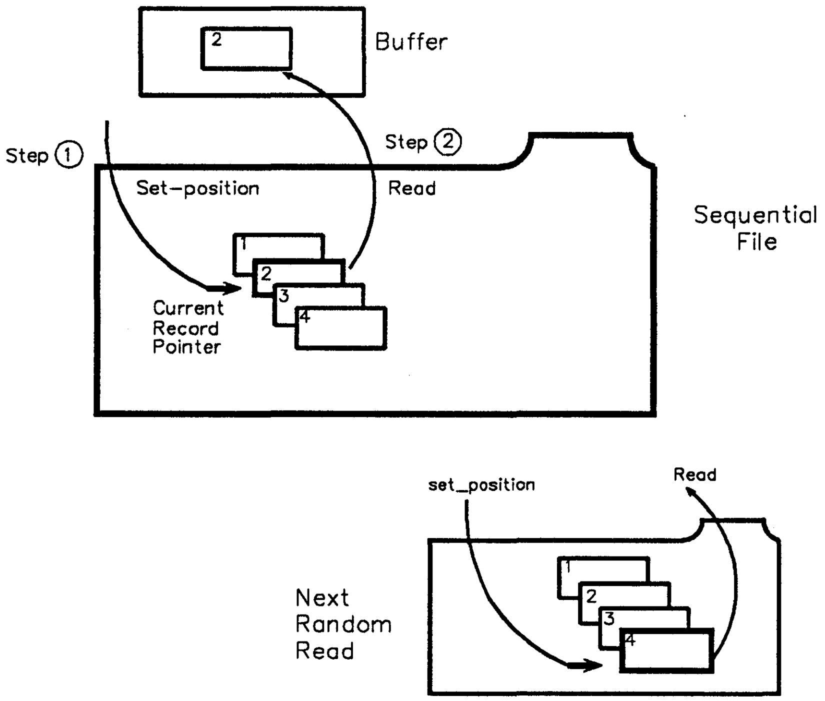


Figure IV-9-3. Physical-Random Access

For physical-random access mode, you can do the following:

- Read records in sequence and, based on the record's value, delete or update the current record.
- Use the `Set_position` call to move the CRP to the first position, last position, a record ID, or record number (if in a relative file) in a file.
- Read, update, or delete the current record using the `current` position modifier. You cannot do insertions by record ID or record number. The record ID and record number cannot change during the lifetime of a record unless the file is reorganized.

IV-9.1.2.3 Indexed-Sequential Access

In the indexed-sequential access mode, you select a range of key values to read. The key range determines the subset of records in a file and the logical ordering within the subset. Figure IV-9-4 shows indexed-sequential access.

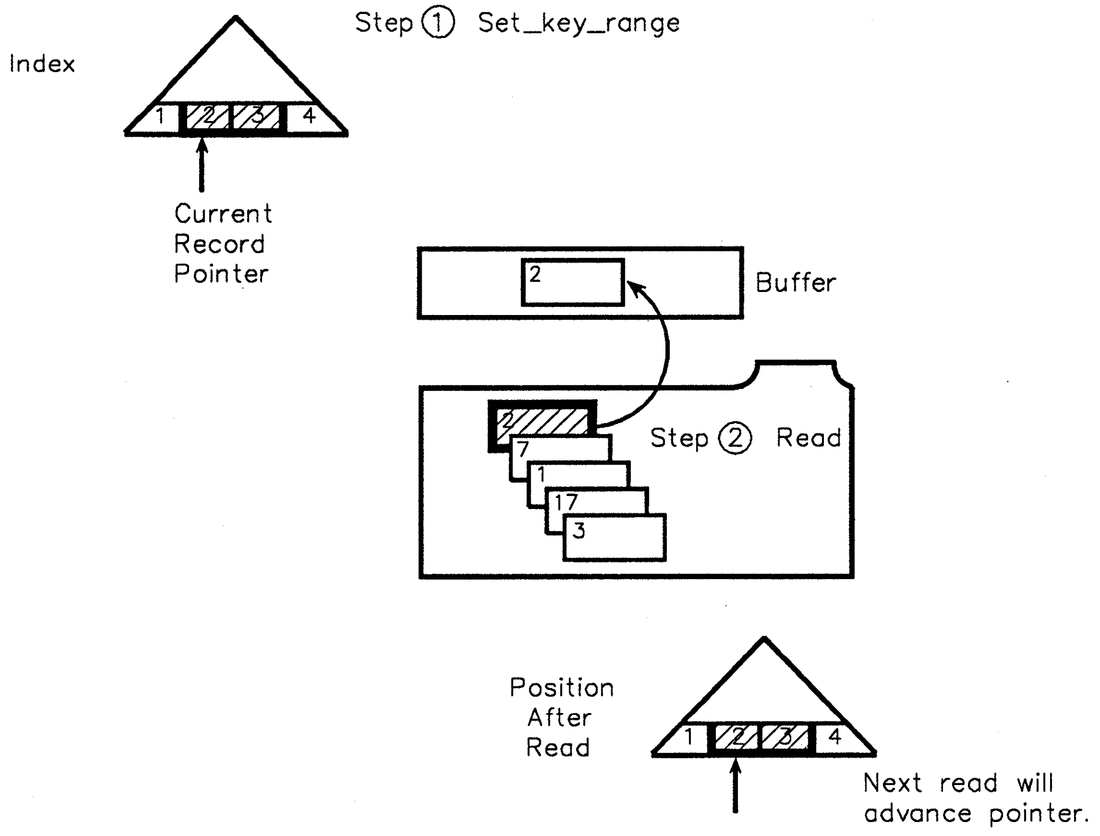


Figure IV-9-4. Indexed-Sequential Access

For indexed-sequential access, you can do the following:

- Select a range of key values to read.
- Read records in forward or reverse index order.
- Use this access mode for any file that has a clustered index or b-tree alternate index organization.

In keyed access, the CRP is actually an index key value. The CRP is initialized by a `Set_key_range` call. You specify an index name and a key value range. You can then read, update, or delete records within that range as follows:

1. You call `Set_key_range` to set starting and ending boundaries within the index. The values of the starting and ending boundaries depend on whether you are reading an ascending or descending index..
2. The modifiers `next` and `prior` are relative to the index structure, *not* the key values. For example:
   - Reading an index consisting of one descending field will return the next *lower* key value using `next` as a modifier.
   - Reading an index consisting of one ascending field will return the next *higher* key value using `next` as a modifier.

3. The first `Read` call after a `Set_key_range` call is relative to the starting boundary (which may or may not be inclusive).
   - If the starting boundary is designated as inclusive and the key value exists, the position modifier is ignored.
   - If the starting boundary is exclusive, or the key value specified does not exist, the position modifier is applied to locate a starting position.
   - The position modifier is always applied on subsequent reads. Other aspects of the position modifier are the same as those for physical-sequential access.
4. A `Read` call returns the actual record defined by the CRP and position modifier. For keyed access, it is possible to just read the key values and record IDs of the records. This is done by calling `Read_key_value`.
   - This permits a fast scan via an index without accessing the primary data area.
   - The returned record IDs can be used to read the actual records later, if desired.
5. Inserts do not affect the CRP (except for relative files where a `Set_position` can precede an `Insert` call). The filing service determines the locations of the insertions.

IV-9.1.2.4 Indexed-Random Access

In this access mode, records are read randomly by supplying a key value to a unique index. Figure IV-9-5 shows indexed-random access.

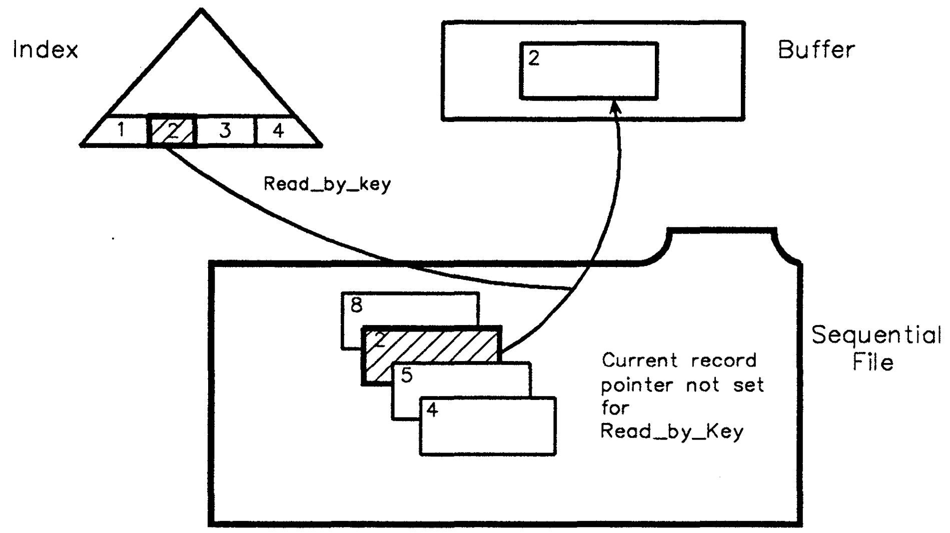


Figure IV-9-5. Indexed-Random Access

In the indexed-random access mode, you can do the following:

- Randomly access records without modifying the CRP with the `Read_by_key`, `Update_by_key`, and `Delete_by_key` calls.
- Specify a unique index name and an index key value for a particular record. The filing service determines the locations of the insertions.

IV-9.1.3 Record I/O and Structured Files

Some structured file organizations only allow you to use particular record I/O calls. These access characteristics apply to the following file organizations.

IV-9.1.3.1 Sequential Files

In this file organization, the physical sequence of the records corresponds to the order in which they were written. Sequential files have the following access considerations:

- You establish a current record position using `Record_AM.Ops.Set_position`.
- You can use the `next` or `prior` position modifiers to position the CRP. A `Record_AM.Ops.Read` call with the `modifier` parameter set to `next` returns records in the same order as the records are written. A `Record_AM.Ops.Read` call with `modifier` set to `prior` returns records in reverse order.
- A `Record_AM.Ops.Insert` call appends a new record at the end of the sequential file.
- When you read a record with the `Record_AM.Ops.Read` call or set the record position with the `Record_AM.Ops.Set_position` call, the `Record_AM.Ops.Update` call overwrites the record at which the CRP points.
- The length of variable-sized records cannot be changed.
- The `Record_AM.Ops.Delete` call is not allowed.
- The `Record_AM.Ops.New_lines` call inserts a control record with the specified number of empty lines in it.
- The `Record_AM.Ops.New_page` call inserts a control record with a page mark.

Sequential files with variable-sized records can store control data having no direct relationship to the contents of the records. For example, some control data in a file controls the format and page layout in text files. This additional information is stored in separate *control records* containing a number of empty lines or page mark characters. The `Record_AM.Ops.Insert_control_record` call inserts a record with control information. You must use the `Record_AM.Ops.Set_open_mode` call to set `Record_AM.read_ctl_rec` to true to get control records in read operations, otherwise control records are ignored. No index entries are added for control records.

IV-9.1.3.2 Relative Files

Relative files have the following access considerations for physical-sequential access:

- After using `Record_AM.Ops.Set_position` to initially set the CRP, you use the position modifiers `next` or `prior` for positioning the CRP.
- A sequential `Record_AM.Ops.Read` call returns records in ascending or descending order of the record numbers. Empty slots are skipped automatically.
- A `Record_AM.Ops.Update` call overwrites the record you just read. The length of variable-sized records may shrink or grow unless it exceeds the maximum record length given by the slot size.
- The `Record_AM.Ops.Delete` call deletes the record that was accessed by the immediately preceding `Record_AM.Ops.Read` call or the record that was defined by the preceding `Record_AM.Ops.Set_position` call.

Relative files have the following considerations for *physical-random* access:

- Access by record number, to the beginning of the file or the end of the file is done by calling `Record_AM.Ops.Set_position`. You do this prior to making `Record_AM.Ops.Update`, `Record_AM.Ops.Delete`, or `Record_AM.Ops.Insert` calls.
- For the `Record_AM.Ops.Insert` call, the filing service inserts the new record in the first empty slot if you specify `first` with `Record_AM.Ops.Set_position`. If you specify `last`, the filing service inserts the new record at the end of the file.
- The filing service raises the `invalid_record_address` exception when an addressed slot is not empty.
- You can insert records using any record number. For example, if the first record written to the file is record number 100, the slots with numbers 1 to 99 are marked empty.

IV-9.1.3.3 Hashed Files

Hashed files have the following access considerations:

- You can do indexed-sequential access only with hashed files having a hashed organization index supporting duplicates.
- Physical positioning is allowed.
- A `Device_Defs.end_of_file` exception is raised after all duplicates of the current key value are read.

IV-9.1.4 End of File

The EOF pointer points past the last record in the file to the end-of-file. When end-of-file is reached during a read, the `Device_Defs.end_of_file` exception is raised. The `at_EOF` boolean in the `operation_status_record` data structure also indicates the end-of-file.

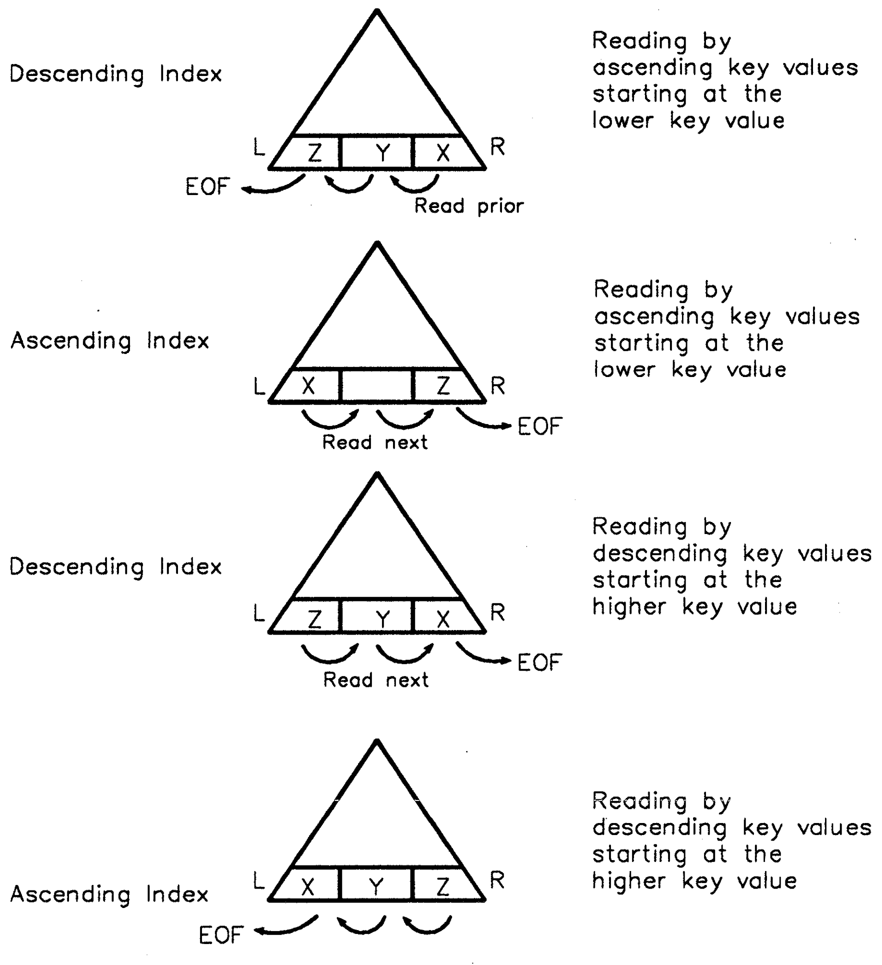
The EOF pointer for a storage file may change as records are inserted, updated, and deleted. The `at_EOF` boolean in the operation status record is set when the last record in the file is read. A `Device_Defs.end_of_file` exception is raised if a `Read` is attempted at or beyond the current end-of-file. If writers are active in the file, the end-of-file can change in the instant after a `Record_AM.Ops.Read` call.

IV-9.1.4.1 End of File for Indexed Access

During indexed-sequential access, the `Device_Defs.end_of_file` exception is raised if an attempt is made to read outside of certain boundaries. This attempt is detected in a manner consistent in all cases. The cases are:

- Reading by ascending key values in a descending index structure using `prior`, starting at the lower key value
- Reading by ascending key values in an ascending index structure using `next`, starting at the lower key value
- Reading by descending key values in a descending index structure using `next`, starting at the higher key value
- Reading by descending key values in an ascending index structure using `prior`, starting at the higher key value.

Figure IV-9-6 shows how EOF is detected for each of these cases.


Descending Index
L
Z
Y
X
R
EOF
Read prior
Reading by ascending key values starting at the lower key value
Ascending Index
L
X
Z
R
Read next
EOF
Reading by ascending key values starting at the lower key value
Descending Index
L
Z
Y
X
R
Read next
EOF
Reading by descending key values starting at the higher key value
Ascending Index
L
X
Y
Z
R
EOF
Reading by descending key values starting at the higher key value

L	=	Left of Index
R	=	Right of Index
EOF	=	End-of-file exception

Figure IV-9-6. EOF Detection During Indexed-Sequential Access

When reading sequentially via a nonunique index, the system returns the next record of the duplicate sequence and sets a `duplicate` boolean to true if records exist with the same index key value. See the `operation_status_record` in the `Record_AM` package.

Reading all duplicates for a single key value can be done for both b-tree and hashed indexes. (This is a *sequential* and not a random operation.) You must use `Set_key_range` to set the start and stop boundaries to the same value.

When rereading a record using a `current` position modifier, `key_value_changed` is raised if the key value that was used to locate the record changed as the result of an intervening `Update` or `Delete` call.

For a single opened file, you can alternate back and forth between physical-sequential and indexed-sequential access by varying the use of `Record_AM.Ops.Set_key_range` and `Record_AM.Ops.Set_position` calls.

IV-9.1.5 Record I/O and Transactions

Certain record I/O operations will *block* if a record or file is locked by another transaction. The duration of the blocking can be set using a `timeout` parameter. The default value for this parameter is `wait_forever`.

Once an opened file becomes part of a transaction (this is an implicit association), it may not be used by another transaction until the first transaction commits or aborts; otherwise `ODO_using_different_transaction` is raised. (An opened file associates with a transaction the first time it is used in a call that reads or modifies records.)

You can find information on transaction semantics and locking for record I/O calls on files in Chapter IV-10.

IV-9.1.6 Files and Disk Flushes

In order for a file to be consistent, all information changed by `Record_AM.Ops.Insert`, `Record_AM.Ops.Update`, or `Record_AM.Ops.Delete` calls must be physically written to the file. This information includes both your data and any affected system-maintained control information. In the case of files, both data buffers and control information are maintained in nonvolatile memory to protect against power failures. Because of this protection, information is not flushed from nonvolatile memory to a disk when a file is closed. However, a `Record_AM.Ops.Flush` call is available to write the modified contents of a file to disk at any time.

IV-9.1.7 Record I/O Operation Status

The status of a record I/O operation is returned in the `operation_status_record`. You can retrieve operation status after most `Record_AM` calls. Table IV-9-1 shows information available in the `operation_status_record`:

Table IV-9-1. Operation Status Record

Parameter	Description
`rec_length`	Actual record length in bytes if the last call was a "Read".
`rec_ID`	Record ID of the record that was processed by the last successful call.
`rec_num`	Number of the record that was processed by the last successful call. This is valid only for relative files.
`insert_dupl`	True if the record has a duplicate in at least one active index that allows duplicates, else false.
`at_EOF`	True if an end_of_file exception is raised, else false.
`at_EOP`	True if the CRP points to the end of a page in a stream file or in a sequential file, else false.
`ctl_record`	True if the last record read or positioned to was a control record, else false.
`duplicate`	True if additional records with the same key value have not yet been accessed by a Read or Read_key_value call, else false.

Query a file's open status with `Record_AM.Ops.Get_open_status`. See the `Record_AM.open_status_record` record in the `Record_AM` package.

IV-9.2 Techniques

After reading this section, you will be able to:

- Open and close a structured file
- Set a file's open mode
- Get record IDs and record numbers
- Insert a record
- Read and update a record
- Access fields in a record buffer
- Set a key range and read key values sequentially
- Delete indexed records sequentially
- Read records using physical-random, physical-sequential, indexed-random and indexed-sequential access.

This section assumes that you are familiar with the record buffering, record DDef, and index DDef techniques described in Chapter IV-2. The following techniques cover the common cases for using record I/O with different access modes. Appendix X-A provides the complete listing for these examples.

IV-9.2.1 Opening and Closing Structured Files

Opening a file makes an existing file available for processing.

Calls Used:

`Record_AM.Open_by_name`
Opens a file using the pathname of the file.

`Record_AM.Ops.Open`
Opens a file using an AD to the file object.

`Record_AM.Ops.Close`
Closes a file using an AD to the file object.

The `Record_AM` package has two open calls. `Record_AM.Open_by_name` lets you open a file by name; `Record_AM.Ops.Open` lets you open a file with a file AD. An open call returns an opened file that you can use in subsequent file processing operations. If multiple processes are allowed access to a file, the filing service maintains a file status for an opened file for each particular opening of the file. The open calls let you specify how you want to use the opened file and how you want to allow other callers to use the file. You can specify the parameters:

- `input_output` that specifies the type of I/O you want to do with the file
- `allow` that specifies the operations *other* users can do with the file
- `block` that specifies whether to wait for the file if it is in use.

When you open a file:

- You acquire a file-level lock (except when the `allow` parameter is set to `anything`).
- If the file is already opened or locked in a mode which is incompatible with your `allow` mode and you set the `block` parameter to false, the filing service raises the `Device_Defs.device_in_use` exception.
- If you set the `block` parameter to true, you wait until other users release their locks or until the system interrupts. This terminates file processing and closes the associated opened file.

A `Close` call terminates access to an opened file. If the opened file was passed to other jobs, each job must terminate or close the file before it is totally deallocated.

IV-9.2.2 Setting Open Mode

The `Set_open_mode` call sets particular options that modify the behavior of an opened file.

Calls Used:

`Record_AM.Ops.Set_open_mode`
Sets boolean values associated with an opened file.

Possible values are specified using the `open_mode_value` data structure as follows:

`read_ctl_rec`	If true you want to read data records and control records in a sequential or stream file; otherwise you want to retrieve only data records.
`load`	If true a clustered, hashed, or unordered file is opened for initial loading.
`auto_close`	If true the current transaction automatically closes the file when it commits or aborts; otherwise the file remains open until you close it or your job terminates.
`level_3`	If true you want level 3 consistency. Chapter IV-10 discusses level 3 consistency.

For example, the following procedure sets the open mode `level_3` boolean to true:

```
level_3_mode: Record_AM.open_mode_value(Record_AM.level_3) :=
    (mode_id => Record_AM.level_3,
     value   => true);
...
Record_AM.Ops.Set_open_mode(
    opened_dev => opened_file,
    mode_value => level_3_mode);
  -- Sets level 3 consistency.
```

IV-9.2.3 Inserting Records

When you insert a record in a file, the filing service determines where to insert the record in the file. Where the filing service inserts the record depends on the file organization. For example, the filing service appends a record to the end of a sequential file, but for clustered or hashed files it inserts the record based on the file's indexing algorithm. During the insert, the filing service automatically assigns each new record a record ID.

Calls Used:

`Record_AM.Ops.Insert`
: Inserts a record.

To insert a record into an opened file, you load the record into a record buffer and pass the buffer to the `Record_AM.Insert` call. To load the record into the buffer, you can use an Ada *address clause* or calls in the `Field_Access` package.

To use the Ada address clause to load a record buffer, you declare a record buffer and declare your record as a virtual address to the buffer. When you assign the record's data to your record (for example, an employee record), the record data is located at the address of the record buffer which in effect loads the buffer.

Remember, you need to understand the DDef alignment rules in the `Data_Definition_Mgt` package if your file references a record DDef.

IV-9.2.4 Accessing Fields in Record Buffers

If your file references a record DDef, you can also access a buffer using the `Field_Access` package. The `Field_Access` calls do field alignment and check for null values.

Calls Used:

`Field_Access.Initialize_record`
: Initializes and sizes a buffer for a record based on a DDef and, optionally, fills the record's buffer with default values.

`Field_Access.Get`
: Gets a value for a single field from a previously initialized record buffer.

`Field_Access.Put`
: Puts a value for a single field into a previously initialized record.

`Field_Access` aligns the fields according to the record layout definition contained in the DDef. It also checks if null values are allowed for a particular field.

The remainder of the examples in this chapter use an Ada record declaration, rather than the field access method. This is possible as long as the representation and alignment of the file's record DDef is the same as the Ada declaration (true in most cases, but watch out for varying strings since the data definition service lays out the fixed part of the record before the varying part).

IV-9.2.5 Deleting Records

When you delete a record and its key values, the record is no longer accessible.

Calls Used:

`Record_AM.Ops.Delete`
Deletes a record at the position of the CRP.

`Record_AM.Keyed_Ops.Set_key_range`
Sets two key value boundaries within an index.

To delete a set of records in a key range, you set the key range for the records to be deleted with this call:

```
Record_AM.Keyed_Ops.Set_key_range(
    opened_dev   => opened_file,
    index        => Employee_Filing_EX.
        dept_index_name,
    select_range => (
        start_comparison => Record_AM.inclusive,
        start_value      => start_key_descr,
        stop_comparison  => Record_AM.inclusive,
        stop_value       => stop_key_descr));
```

The following call reads and deletes the records in the key range:

```
loop
  -- CRP is updated after each delete
  -- (no read is necessary to preface
  -- the Delete).
  Record_AM.Ops.Delete(
      opened_dev => opened_file,
      modifier   => Record_AM.current,
        -- Normally defaulted.
      timeout    => Record_AM.wait_forever,
      status     => null);

end loop;
```

When the end of the key range is reached, the filing service raises the `Device_Defs.end_of_file` exception which exits the loop.

In addition to the `Record_AM.Ops.Delete` call, there is a `Record_AM.Ops.Truncate` call that deletes a specified record and all the records that follow it in physical sequence.

IV-9.2.6 Reading and Updating Records

You usually update a file by reading a record and doing the update.

Calls Used:

`Record_AM.Ops.Set_position`
Sets the CRP.

`Record_AM.Ops.Read`
Reads a record from an opened file.

`Record_AM.Ops.Update`
Updates a record.

The following example sets a position in the file to begin reading with the `Record_AM.Set_position` call, reads each record into a record buffer, changes the record in the buffer, and updates the record in the file.

```
Record_AM.Ops.Set_position(opened_file,
    where =>      Record_AM.record_specifier(
        type_of_specifier => Record_AM.first)'(
            type_of_specifier => Record_AM.first));
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => buffer'length);

  current_record_VA.salary :=
      pay_raise * current_record_VA.salary;

  Record_AM.Ops.Update(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => buffer'length);
end loop;

```

Each field in the record buffer (for example, a salary field) can be individually addressed by specifying the field's name (for example, `current_record_VA.salary`).

You can retrieve record IDs and record numbers from the operation status the read calls return. You declare a read status variable that contains information about the read, make a `Set_position` call to set the CRP, and read the record. You can then use the status variable to obtain the record ID:

```
read_status_VA:  Record_AM.operation_status_VA :=
    new Record_AM.operation_status_record;
  -- Virtual address of status record.

  Record_AM.Ops.Set_position(
      opened_dev => opened_file,
      where =>      Record_AM.record_specifier(
          type_of_specifier => Record_AM.first)'(
              type_of_specifier => Record_AM.first));
  loop
    bytes_read := Record_AM.Ops.Read(
        opened_dev => opened_file,
        buffer_VA  => buffer'address,
        length     => buffer'length,
        status     => read_status_VA);
    if current_record_VA.name = employee then
      RETURN read_status_VA.rec_ID;

    end if;
  end loop;
```

You can also use the status variable to obtain the record number:

```
Record_AM.Ops.Set_position(
    opened_dev => opened_file,
    where =>      Record_AM.record_specifier(
        type_of_specifier => Record_AM.first)'(
            type_of_specifier => Record_AM.first));
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => buffer'length,
      status     => read_status_VA);
  if current_record_VA.name = employee then
    RETURN read_status_VA.rec_num;

  end if;
end loop;
```

Note that only records in relative files have record numbers.

IV-9.2.7 Using Physical-Random Access

Two cases of physical-random access are discussed here:

- Reading Records Randomly Using Record IDs
- Reading Records Randomly Using Record Numbers.

Calls Used:

`Record_AM.Ops.Set_position`
Sets the CRP.

`Record_AM.Ops.Read`
Reads a record from an opened file.

Reading Records Randomly Using Record IDs. Random reads using record IDs are available to all file organizations. You can read records randomly by specifying a record ID:

```
Record_AM.Ops.Set_position(
    opened_file,
    where =>      Record_AM.record_specifier(
        type_of_specifier => Record_AM.id)'(
            type_of_specifier => Record_AM.id,
            rec_id            => rec_ID));

bytes_read := Record_AM.Ops.Read(
    opened_dev => opened_file,
    buffer_VA  => buffer'address,
    length     => buffer'length);
```

Reading Relative Files Randomly Using Record Numbers. Random reads (using record numbers) are available only for relative files.

```
begin
  Record_AM.Ops.Set_position(
      opened_file,
      where =>      Record_AM.record_specifier(
          type_of_specifier => Record_AM.number)'(
              type_of_specifier => Record_AM.number,
              rec_num           => rec_number));
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => buffer'length);
```

The records are addressed by a relative record number. The first record is assigned the number 1, the second the number 2, and so forth. Records can be written by specifying any record number (the number does not have to be in a numeric sequence).

IV-9.2.8 Using Physical-Sequential Access

There are four basic cases of physical-sequential access:

- Reading Records in a Forward Sequence Starting at the Beginning
- Reading Records in a Reverse Sequence Starting at the End
- Reading Records Sequentially Starting with a Record ID
- Reading Records Sequentially Starting with a Record Number.

Calls Used:

`Record_AM.Ops.Set_position`
Sets the CRP.

`Record_AM.Ops.Read`
Reads a record from an opened file.

You use the same calls and declarations in all these cases. You set up a record buffer and your record as shown on page IV-9-14. Remember that you use the Ada address clause to define your record. You can access each field in the record buffer as a virtual address. For example, you can specify the salary field with the buffer field name `current_record_VA.salary`.

Reading Records Sequentially in a Forward Sequence. This case is used to read all records in a sequence starting at the beginning of the file. The current read pointer is positioned to the *beginning* of the file.

```
Record_AM.Ops.Set_position(
    opened_dev => opened_file,
    where      => Record_AM.record_specifier(
        type_of_specifier => Record_AM.first)'(
            type_of_specifier => Record_AM.first));
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => buffer'length);

  -- DO ANY NEEDED PROCESSING HERE.

end loop;
```

Each successive record is read in a forward sequence until EOF is detected and `Device_Defs.end_of_file` is raised.

Reading Records Sequentially in a Reverse Sequence. This case is used to read all records in a sequence starting at the end of the file. The current read pointer is positioned to the *end* of the file.

```
Record_AM.Ops.Set_position(
    opened_dev => opened_file,
    where      => Record_AM.record_specifier(
        type_of_specifier => Record_AM.last)'(
            type_of_specifier => Record_AM.last));
      -- Positions current record pointer
      -- to last record in file.
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      modifier   => Record_AM.prior,
      buffer_VA  => buffer'address,
      length     => buffer'length);

  -- DO ANY NEEDED PROCESSING HERE.

end loop;

```

Notice that `record_spec` is initialized to the `last` record in the file. The `modifier` value is set to `prior`. Each successive record is read in reverse sequence until EOF is detected and `Device_Defs.end_of_file` is raised.

Reading Records Sequentially Using Record IDs. Sequential reads using record IDs are available to all file organizations. You use the record ID to establish a starting point for a sequence of `Read` calls. Your program must keep track of record IDs using the `operation_status_record` in order to specify a record ID from which to start reading. You initialize the read to a specific record ID using a `rec_ID` value from the status information. Use the same techniques as in the example on page IV-9-17.

Reading Relative Files Sequentially Using Record Numbers. Sequential reads using record numbers are available only for relative files. You use the record number to establish a starting point for a sequence of `Read` calls. Your program must keep track of record numbers using the `operation_status_record` in order to specify a record number from which to start reading. You initialize the read to a specific record number using a `number` value from the status information. Use the same techniques as in the example on page IV-9-17.

IV-9.2.9 Using Indexed-Random Access

The only way to manipulate a single specified record using indexed-random mode is to make an `Update_by_key` or `Delete_by_key` call. These calls do not affect the CRP.

Calls Used:

`Record_AM.Keyed_Ops.Update_by_key`
Updates a record by key value from an opened file.

An `Update_by_key` call requires you to specify `Record_AM.key_value_descr` record which supplies the key value for the update. Otherwise, the record at the current position of the CRP is updated. In the following example, the record at the current position is updated.

```
current_record_VA.salary :=
    pay_raise * current_record_VA.salary;

-- Default is the current record.
Record_AM.Keyed_Ops.Update_by_key(
     opened_dev => T2_opened_file,
     buffer_VA  => buffer'address,
     length     => buffer'length,
     index      => Employee_Filing_EX.
         dept_salary_index_name);
        -- Employee ID index.
```

In this example, an employee's salary is raised and the salary is updated.

IV-9.2.10 Using Indexed-Sequential Access

Indexed-sequential access lets you read in a forward or reverse sequence in a file using the index.

Calls Used:

`Record_AM.Keyed_Ops.Set_key_range`
Sets two key value boundaries within an index.

`Record_AM.Ops.Read`
Reads a record from an opened file.

The result of the `Read` is different depending on whether the index is ascending or descending, and whether the key range is being traversed in a forward or reverse sequence.

Reading in a forward sequence. To read forward in a simple index, use the `next` option as the `modifier` value to your `Read` call.

Follow these steps:

- Specify an index with an index DDef field that does not have the `pi_descending` property identifier (that is, the field is sorted in ascending order).
- Set the start value of the index key range to a minimum value at which you want to begin reading.

The read starts from the low key value and ascends to the high key value in the key range, going from *left to right* in the index structure until `Device_Defs.end_of_file` is raised.

This example uses a simple single-field index on department (ascending).

```
-- A simple index declaration.
dept_index_DDef:  Data_Definition_Mgt.
                       node_reference;

dept_index_name:  constant
   File_Defs.index_name :=
      (max_length => File_Defs.index_name_length,
       length => 14,
       value  => "Dept_Index_DDef                 ");

type dept_key_buffer is
  record
    dept:        department_number;
  end record;
```

KThis procedure positions to the beginning of the range, and reads successive records until the end of the range. You set the start value to the left of the index (the low end), and a stop value to the right of the index (the high end). You set the key range to start at the low end of the key range and stop at the high end.

```
Record_AM.Keyed_Ops.Set_key_range(
    opened_dev      => opened_file,
    index           =>
       Employee_Filing_EX.dept_index_name,
    select_range => (
        start_comparison => Record_AM.exclusive,
        start_value      => start_key_descr,
        stop_comparison  => Record_AM.inclusive,
        stop_value       => stop_key_descr));
```

The simple index is read by ascending key values starting at the low end of the key range.

```
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      modifier   => Record_AM.next,
        -- Next is normally defaulted.
      buffer_VA  => buffer'address,
      length     => buffer'length);

  -- DO ANY NEEDED PROCESSING HERE.

end loop;
```

Reading in a reverse sequence. To read in reverse in a simple index, use the `prior` option as the `modifier` value to your `Read` call.

Follow these steps:

- Specify an index with an index DDef field that does not have the `pi_descending` property identifier (that is, the field is sorted in ascending order).
- Set the start value of the index key range to a maximum value of the range at which you want to begin reading.

The read starts with the high key value of the key range and reads to the low key value of the range, going from *right to left* in the index structure until `Device_Defs.end_of_file` is raised.

This example uses a simple single-field index on department (descending).

```
-- A simple index declaration.
dept_index_DDef:  Data_Definition_Mgt.
                       node_reference;

dept_index_name:  constant
   File_Defs.index_name :=
       (max_length => File_Defs.index_name_length,
        length => 14,
        value  => "Dept_Index_DDef                 ");

type dept_key_buffer is
  record
    dept:        department_number;
  end record;
```

The start value is set to the right of the index (the high key value of the key range). The stop value is to the left of the index (the low key value of the key range). You set the key range with the start value (high end) and stop value (low end) values.

```
Record_AM.Keyed_Ops.Set_key_range(
    opened_dev => opened_file,
    index      =>
      Employee_Filing_EX.dept_index_name,
    select_range => (
        start_comparison => Record_AM.exclusive,
        start_value      => start_key_descr,
        stop_comparison  => Record_AM.inclusive,
        stop_value       => stop_key_descr));

```

The index is read by descending key values starting at the high end of the key range.

```
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      modifier   => Record_AM.prior,
        -- Sets read modifier to prior.
      buffer_VA  => buffer'address,
      length     => buffer'length);

  -- DO ANY NEEDED PROCESSING HERE.

end loop;
```

IV-9.2.11 Reading Key Values Sequentially

The `Record_AM.Keyed_Ops` package provides *keyed operations* that let you read just the key values of records.

Calls Used:

`Record_AM.Keyed_Ops.Set_key_range`
Sets two key value boundaries within an index.

`Record_AM.Keyed_Ops.Read_key_value`
Reads key values from an opened file.

Reading key values is no different than reading records. However, the primary data area of the file is never touched when key values are read. This allows a fast scan of the values in an index without the overhead of reading entire records.

To read duplicates of one key value, you must specify a nonunique index and set the start and stop values of the index to the same value.

The following declaration sets the start and stop values to the same value.

```
start_key_value:  constant Employee_Filing_EX.
    dept_key_buffer := (dept => 305);
       -- Start value for duplicate
       -- key field.


stop_key_value:   constant Employee_Filing_EX.
   dept_key_buffer := (dept => 305);
      -- Stop value for duplicate
      -- key field.

```

When reading sequentially via a nonunique index, the next key value is returned in a duplicate sequence. The `duplicate` field in the `operation_status_record` is set to true if additional records exist with the same key value.

IV-9.2.12 Reading and Updating Records by Key

Sequential reads by key can be combined with random updates.

Calls Used:

`Record_AM.Ops.Set_key_range`
Sets two key value boundaries within an index.

`Record_AM.Keyed_Ops.Delete_by_key`
Deletes a record designated by a unique index key value.

Essentially, you do a series of reads until you find a record you need to update:

```
Record_AM.Ops.Set_position(
    opened_dev => opened_file,
    where =>      Record_AM.record_specifier(
        type_of_specifier => Record_AM.first)'(
            type_of_specifier => Record_AM.first));
loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      buffer_VA  => buffer'address,
      length     => buffer'length);

  if current_record_VA.dept = 175 then
    Record_AM.Keyed_Ops.Delete_by_key(
       opened_dev => opened_file,
       index      => Employee_Filing_Ex.
           dept_index_name);

  end if;

end loop;
```

IV-9.3 Summary

- `Record_AM` can be used with a number of different devices including files, DDefs, pipes, and directories.

- Structured files are generally accessed using `Record_AM`. *Only* indexed structured files can use the `Keyed_Ops` nested package.
- The *CRP* points to the current record. Normally, this is the most recently read record or the current index and index key value. The CRP can be physically positioned to a particular record using `Set_position` or `Set_key_range` calls.
- Keyed access can also occur *without* setting the CRP using `Read_by_key`, `Update_by_key`, and `Delete_by_key` calls.
- Physical-sequential access can be done from the beginning of a file in a forward sequence, from the back of a file in a reverse sequence, or by record ID or record number (from any random starting point).
- The major difference between physical-sequential access and physical-random access is that each read is preceded with a `Set_position` call for physical-random access, while physical-sequential access uses the `next` position modifier.
- Indexed-sequential access depends on the properties of the DDefs used to define the indexes (that is, ascending or descending). Both ascending and descending indexes can be read in either forward or reverse order.

LOCKING FILES AND RECORDS 10

Contents

The filing service is built for high levels of concurrency and integrity. It uses sophisticated algorithms to support readers while writing, provides highly concurrent b-tree and hashed indexes and supports transaction based synchronization and recovery. For applications that want to eliminate the overhead of transactions, the filing service also supports nontransaction-oriented files.

Packages Required:

`Record_AM`	Provides device-independent I/O for record access. Contains the `Record_AM.Ops` and `Record_AM.Keyed_Ops` packages.
`Record_AM.Ops`	Common interface for record I/O calls.
`Record_AM.Keyed_Ops`	Special I/O calls for indexed access to files.
`File_Admin`	Basic declarations and calls for file administration.

In a multiuser system that does not support transaction locking, lost updates can occur. For example, when updating a record in a file, you read the record from passive store into a private copy, update your copy, and write the copy back to passive store. You can lose your update if another user reads the same record you are updating before your write completes, updates their copy, and writes their copy back to passive store after your write completes. Figure IV-10-1 shows how a lost update can happen.

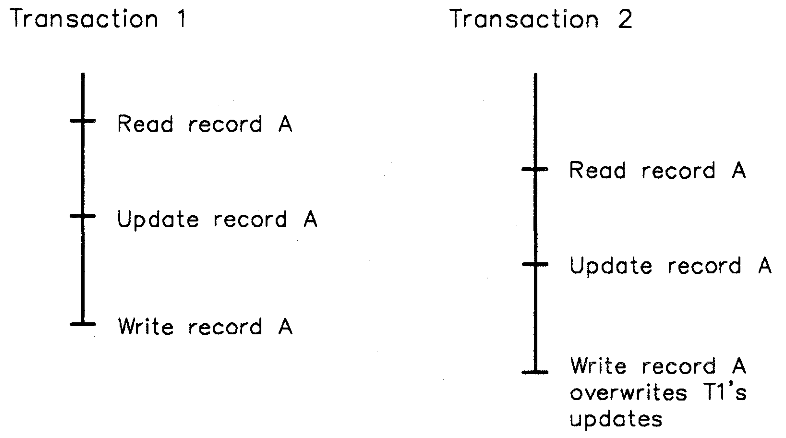


Figure IV-10-1. Lost Update Problem

Locking can prevent lost updates.

IV-10.1 Concepts

Some problems that can occur in multiuser systems that allow more than one user to access a file simultaneously are:

- Lost updates - when one user overwrites another user's changes.
- Reading uncommitted changes - when a user reads data modified by transactions that have not completed.

- Nonreproducible reads - when a user reads the same record twice in a transaction getting different results each time.

These problems can be avoided by using locking features the filing service provides.

IV-10.1.1 Concurrency Control and Recovery

An efficient online multiuser data processing system should:

- Assure that the data accessed by the users looks consistent from their viewpoint, especially when multiple users are trying to access the same pieces of data (*concurrency control*).
- Assure that any changes to the data are *undone* if a transaction has an unsuccessful termination (*rollback*).
- Assure that any changes to the system's data are *redone* if there is a media failure, system crash, or application error (*rollforward*).

The filing service supports concurrency control by providing transaction-based locking and recovery by using logging techniques. The filing service provides two different logging techniques:

- *Short-term logging* to support rollbacks of transactions
- *Long-term logging* to support rollforward operations.

The filing service lets you use locking, short-term logging, and/or long-term logging for any structured file, by appropriately setting the following fields in the *logical file descriptor* when creating the file:

`xm_locking`	Specifies that the file can be locked for transactions.
`short_term_logging`	Specifies short-term logging to support rollback for transaction aborts.
`long_term_logging`	Specifies long-term logging to support rollforward for recovery from media failure or user errors.

Any structured file you create with at least one of the locking or logging fields set to true is called a *transaction-oriented file*. Any file you create with all the locking and logging fields set to false is a *nontransaction-oriented* file. Nontransaction-oriented files do not support transaction locking, short-term logging, or long-term logging, and do not have the transaction overhead associated with transaction-oriented files. Nontransaction-oriented files are more efficient to use in applications where concurrency control and recovery are not important.

NOTE

If you do not have a transaction when accessing transaction-oriented files, the filing service will automatically start a transaction and commit it at the end of the operation.

All the transactions started internally by the filing service are low-overhead transactions. The only exceptions are the long-term logged files where the filing service starts real transactions.

IV-10.1.2 Transaction Locking

Most locking is done implicitly by the filing service when you access or modify the data contained in files. However, the filing service does provide the option to place certain locks explicitly, and to drop some locks.

The filing service uses *hierarchical locking* for concurrency control. The three levels of the hierarchy are:

- Files (highest level)
- Index key ranges
- Records (lowest level).

Figure IV-10-2 shows the locking hierarchy.

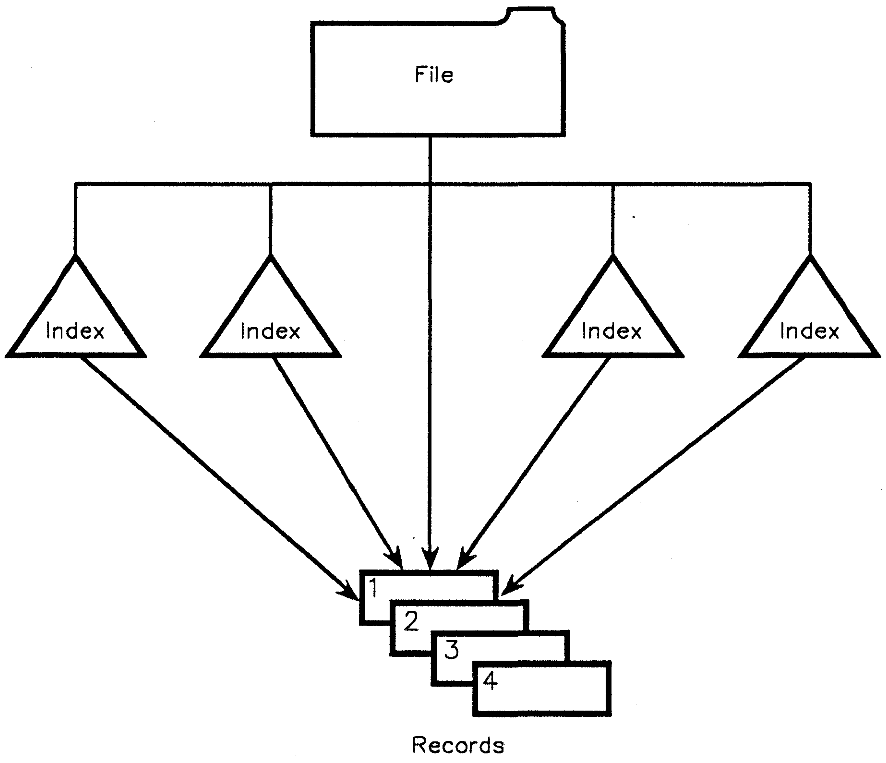


Figure IV-10-2. Locking Hierarchy

A record-level lock locks one individual record. An index-key-range-level lock locks all the records whose index key fields fall in the range that is being locked, which is usually a subset of the records contained in a file. A file-level lock locks all the records in the file. File-level locks are more efficient, because only one lock is required to lock all the records in a file. Record-level locks provide better concurrency, because each transaction locks only a subset of records in a file.

IV-10.1.2.1 Lock Modes

There are two basic lock modes supported by the filing service:

- read (r) lock (shared or *s-locks*).
- write (w) lock (exclusive or *x-locks*)

Read locks prevent other transactions from updating a record. This lock mode allows other transactions to read but not modify the record. Write locks prevent other transactions from reading or updating the record. Figure IV-10-3 shows how write locks prevent other users from accessing records you are using.

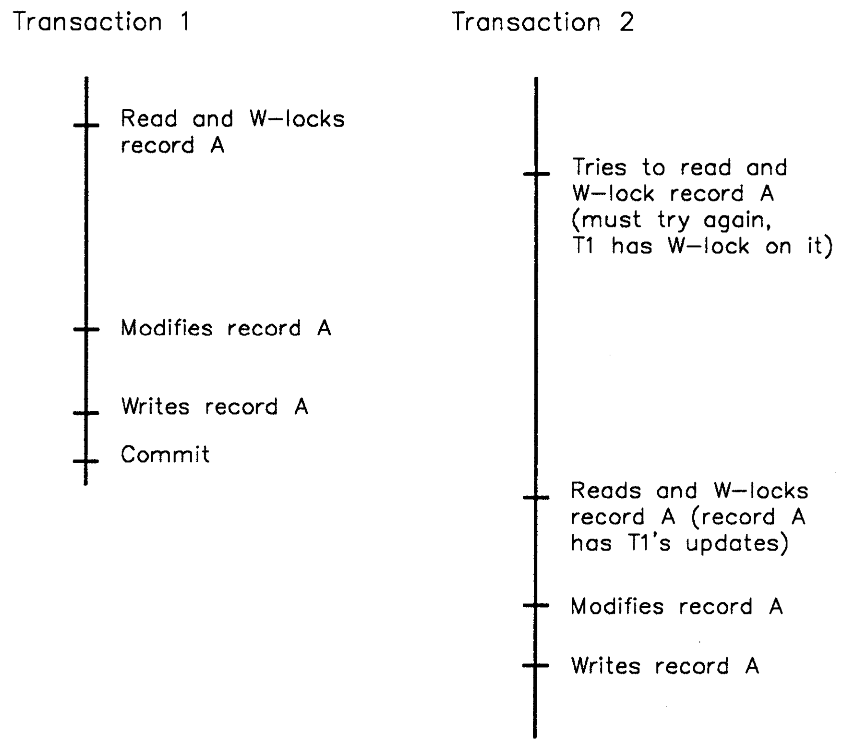

Transaction 1
Transaction 2
Read and W-locks record A
Tries to read and W-lock record A (must try again, T1 has W-lock on it)
Modifies record A
Writes record A
Commit
Reads and W-locks record A (record A has T1's updates)
Modifies record A
Writes record A

Figure IV-10-3. An Update with an X-lock

In the figure IV-10-3, transaction 2 cannot read record A until transaction 1 commits, because transaction 1 holds a write lock on on it.

To understand locking, it is necessary to explain several locks that are not visible to transactions. In addition to the read and write locks, the filing service uses *intention locks*. Intention locks tag an entity at a higher level and indicate that locking is being done at a finer level. (Only the read, write, and protect locks are visible to you. The intention-read and intention-write locks are used internally by the filing service.) For instance, an intention lock on a file indicates that locking is being done at a lower (key range or record) level. There are three intention locks:

- intention-read (ir) lock (also known as *intention-share or is-lock*)
- intention-write (iw) lock (also known as *intention-exclusive or ix-lock*)

- protect (r-iw) lock (also known as *shared intention-exclusive or six-lock*).

For instance, when there are record-level write locks in a file, the filing service puts an intention-write lock at the file-level. This prevents other transactions from placing a file-level write lock on the same file.

The *protect* (r-iw) lock is a combination of read (r) and intention-write (iw) locks. A protect lock is useful for transactions that may want the exclusive right to modify a file, but want to allow other transactions to read the file. The protect lock is also useful for transactions that may want to read many records but update only a few.

Another lock mode is the dirty read (null) lock. This lock mode does not provide any locking.

NOTE

If you lock one level of the locking hierarchy, the lower levels of the hierarchy are considered locked with the same lock mode. For example, when a file is locked with a write lock, all the records in that file are considered locked with a write lock.

IV-10.1.2.2 Lock Mode Compatibility

Table IV-10-1 shows the compatibility between lock modes.

Table IV-10-1. Compatibility of Locks

	null	ir	iw	r	r-iw	w
null	yes	yes	yes	yes	yes	yes
ir	yes	yes	yes	yes	yes	no
iw	yes	yes	yes	no	no	no
r	yes	yes	no	yes	no	no
r-iw	yes	yes	no	no	no	no
w	yes	no	no	no	no	no

Compatibility between lock modes is an issue only when different transactions are competing for the locks on the same record or file. All locks within the same transaction are always compatible.

IV-10.1.3 Acquiring Locks

Transactions implicitly acquire locks when they read or write records in a file. The filing service always places write locks on records that are written (inserted, updated, or deleted). For reads, the filing service allows you to choose the lock-mode for the record being read. The default lock for reading is the `read_lock`.

For applications that read records before updating them, you can place a write lock on the record when you read it. Therefore, the filing service does not have to upgrade the lock to a write lock during the record update. On the other hand, for applications where consistency is unimportant, you can do a *dirty read* using the `dirty_read` lock mode. With a dirty read, transactions do not acquire locks and other transactions can change the data you are reading.

Transactions can explicitly acquire file-level locks by calling `Record_AM.Ops.Lock_all` and setting the `by_transaction` parameter to true. With file-level locks, you do not need to have locks with the same mode at the key range or record level, avoiding the overhead of individual record locks. Transactions can also acquire file-level locks implicitly through *lock escalation* by the filing system. The appropriate file-level intention locks are always acquired whenever a lock is placed on a record or key range.

See page IV-10-8 for information about how transactions acquire locks from subtransactions.

IV-10.1.4 Lock Escalation

The filing service automatically attempts to upgrade the file-level lock, when a transaction reaches a threshold for the number of record locks a transaction can hold in the file. (The filing service tries to convert the transaction's record locks to a single file-level lock.) The filing service raises the `lock_escalation_failed` exception if it is unable to upgrade the lock.

When creating a file, you can specify the `lock_escalation_count` in the file's logical file descriptor to set the threshold value for the file's record locks. If you set this value to zero, the exception `Record_AM.too_many_locks` is raised when a transaction exceeds the default limit for the number of record locks per file per transaction.

IV-10.1.5 Releasing Locks

The filing service automatically releases all the locks acquired in the transaction when a transaction terminates. However, you can explicitly release some read and protect locks before a transaction terminates.

There are several ways to release read and protect locks. You can release file-level read locks and file-level protect locks using the `Record_AM.Ops.Unlock_all` call. (You need to set the `by_transaction` parameter to true.) You can release record-level read locks using the `Record_AM.Ops.Unlock` call. Only one instance of a lock is released unless the `unlock` parameter is set to `all_read_locks`. (If a lock has been acquired a number of times, each instance of the lock must be unlocked to completely unlock the file or record.) You can also release record-level read locks in the `Record_AM.Ops.Read` or `Record_AM.Keyed_Ops.Read_key_value` calls, by setting the `unlock` parameter to `free` or `all_read_locks`.

IV-10.1.6 Consistency Levels

A transaction can have different *levels of consistency*. In general, as the level of consistency increases, the amount of concurrency decreases. The filing service provides support for the following levels of consistency:

Level 3	Transactions with level 3 consistency are serializable, that is, their behavior is the same whether they are run concurrently or serially according to some serial order. Level 3 transactions guarantee that all modified data is kept locked until the end of the respective transactions, repeated reads always give the same results, and *phantoms* are prevented.
Level 2	Transactions with level 2 consistency guarantee that all modified data is kept locked until the end of the respective transactions, and that none of the data the transaction reads is dirty (changed but not committed by other transactions). No guarantees are made as to whether reads are repeatable and whether can phantoms occur.

Level 1	Transactions with level 1 consistency only guarantee that the data that is modified is kept locked until the end of the respective transactions (dirty reads). No other guarantees are made.

NOTE

A phantom is a nonexistent entity whose appearance after a transaction has started would change the results of the transaction.

Phantom protection may reduce concurrency and should be switched off unless necessary.

With level 3 consistency, the filing service automatically sets key range locks on record fields in an index when a transaction accesses a file through the index. When a transaction holds key range locks, no other transaction can write (insert, delete, or modify) records in the key range (phantom protection).

Your application has level 3 consistency when:

- Phantom protection is set. You do this by building indexes with the `phantom_protected` boolean set to true in the index descriptor.
- Key range locking for the opened device is turned on. You turn on key range locking with the `Record_AM.Ops.Set_open_mode` call setting the open mode value `Record_AM.level_3` to true.

Your application can have level 2 consistency when:

- Transactions release read locks before they complete.
- Key range locking for an opened device is turned off. You do this with the `Record_AM.Ops.Set_open_mode` call setting the open mode value `Record_AM.level_3` to false.

Your application can have level 1 consistency when:

- Transactions do reads without placing any locks (dirty read). A dirty read is set for a read call using a lock mode of `Record_AM.dirty_read`.

IV-10.1.7 Reading Key Range Values

When an application is interested only in reading key values in an index (by using the call `Record_AM.Keyed_Ops.Read_key_value`) but not the whole records, you can turn record locking off. This can be done by setting the `no_record_locking` parameter to true. You can do this only if the index is phantom protected and the opened device has level 3 consistency.

IV-10.1.8 Locking and Nested Subtransactions

The filing service supports logically nested subtransactions that are useful for providing synchronization in multithreaded applications and for constructing the building blocks for transaction-oriented applications. You can use subtransactions to build large applications by suitably nesting one building block within another. You can also use subtransactions to develop applications where the operations in a transaction need to be split into smaller atomic units. Splitting a transaction into smaller units provides a convenient way of containing errors, and provides a way of building more reliable software in a distributed environment.

Transactions and subtransactions can inherit locks from their child transactions. Inherited locks behave differently from the other locks described in this chapter.

When a child transaction commits, the parent subtransaction inherits the locks the child held. These inherited locks then become available to other descendants of the same parent. A descendant can acquire a lock in the base (noninherited) form as long as the lock being acquired does not conflict with the locks held by any other transaction. To transactions that are not descendants, the inherited locks look and behave similarly to the base locks.

If the child transaction aborts, the locks disappear and cannot be inherited.

The inherited locks are called the *held* locks. There is a held lock mode corresponding to each lock mode; for instance, when a write lock is inherited, the inherited version is called write-held lock.

Figure IV-10-3 shows how locks are inherited by subtransactions. In the figure, T2 cannot acquire a write lock on file F2 until T8 commits.

To release a read lock in a subtransaction before the subtransaction has terminated, you can specify these values for the `unlock` parameter in the read calls:

`unlock_to_held`	The subtransaction releases the (read) lock, and its parent inherits the lock.
`free`	The subtransaction releases the (read) lock, but the parent does not inherit it.
`all_read_locks_to_held`	All the read locks associated with a particular subtransaction are released. The parent inherits the locks.
`all_read_locks`	All the read locks associated with a particular subtransaction are released. The parent, however, does not inherit the locks.

If these parameters are specified for a root level transaction, the unlock mode `unlock_to_held` is equivalent to the `free` mode, and `all_read_locks_to_held` mode is equivalent to the `all_read_locks` mode.

You can also release file-level and record-level read locks by calling `Record_AM.Ops.Unlock_all` and `Record_AM.Ops.Unlock` respectively.

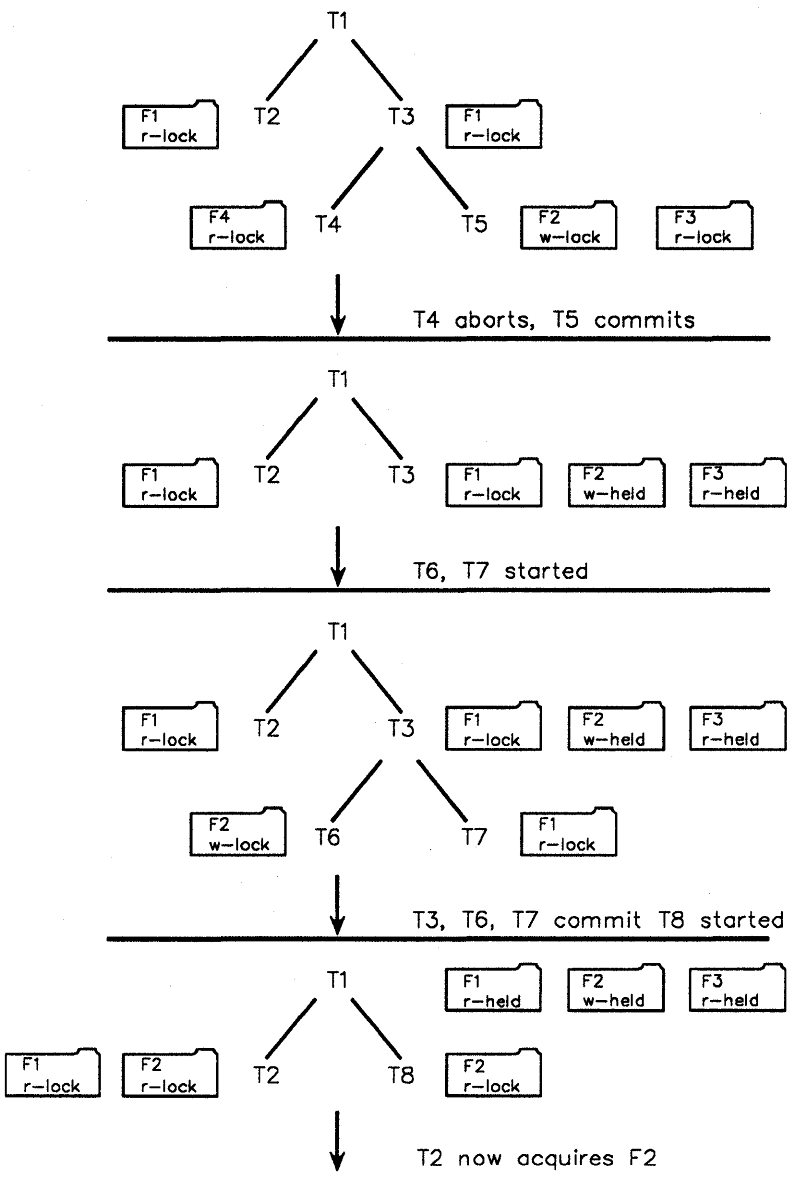


Figure IV-10-4. Locks Inherited by Subtransactions

IV-10.1.9 Lock Contention

Deadlock is a condition where two or more transactions are in a simultaneous wait state, each waiting for the other transactions to release a lock before it can proceed. The filing service uses two independent schemes for avoiding transaction deadlocks.

The default mechanism for avoiding deadlock is the *timestamp ordering* mechanism. If two transactions contend for the same lock, the filing service uses their timestamps to avoid deadlocks. When contending for a lock, a transaction can wait for another transaction that holds the lock when the other transaction has an earlier timestamp. If the transaction that wants the lock has an earlier timestamp, it is not allowed to wait. Subtransactions are exceptions to this rule: a parent transaction can wait for a lock held by a child transaction, even though the parent transaction has an earlier timestamp.

Instead of the timestamp ordering mechanism, you can use the *timeout mechanism* to break deadlocks. Most `Record_AM` calls have a `timeout` parameter that specifies the amount of time a transaction should wait for a lock. When you specify a timeout value in a call, the filing service makes sure that your transaction waits no longer than the duration of the timeout for any single lock. The `Record_AM.timeout` exception is raised at the end of the timeout period if the lock is unavailable.

If you give a timeout value of zero, the `Record_AM.timeout` exception is raised if a transaction cannot obtain a lock immediately. The timeout value `wait_forever` is the default value, and switches the transaction into the timestamp ordering mechanism.

The `Transaction_Mgt` package allows you to specify timeouts for transactions. A timed out transaction raises a `Transaction_Mgt.transaction_not_active` exception. You can also break deadlocks using calls in the `Event_Mgt` package to send signals (events) to a process that is blocked for a lock. Operations that are interrupted by a signal raise the exception `System_Defs.system_call_interrupted`.

See Chapter II-4 for more information about transactions.

IV-10.1.10 Logging

The filing service supports short-term and long-term logging of transactions. Short-term logging provides rollback recovery for transactions. You can request short-term logging when you create a file by setting `short_term_logging` to true. Short-term logging ensures that changes made in a file are rolled back if a transaction aborts or the system crashes.

When a subtransaction commits, the changes are passed on to the parent transaction. The changes are made permanent only when the root-level transaction commits. When a subtransaction aborts, the changes made in the subtransaction and the ones inherited from its children are rolled back.

Long-term logging provides rollforward recovery for transactions. You request long-term logging when creating a file by setting `long_term_logging` to true. Long term logging can be requested only for files that are transaction locking files and are short-term logged. The filing service uses the long-term log information to restore a file by rolling forward all the changes made to a file since a particular backup. Only the changes that a root-level transaction commits are rolled forward.

IV-10.1.11 Transactions and Opened Device Objects

Two or more transactions cannot use the same opened device simultaneously. The exception `Record_AM.odo_using_a_different_transaction` is raised if a transaction attempts to open a device that is already opened by another transaction.

NOTE

After a transaction releases the lock on the opened device, the value `current` is invalid for the current record pointer (CRP). The CRP has to be redefined in a new transaction. The values `next` and `prior` are valid.

IV-10.1.12 File-Level Locks Associated with Opened Devices

In addition to the file-level locks associated with transactions, the filing service associates file-level locks with opened devices. These locks apply to both files transaction-oriented and nontransaction-oriented, and are independent of transaction locks.

The filing service automatically associates file-level locks with opened devices when a file is opened. The lock mode is based on how you open the file using the `input_output` and `allow` parameters. Table IV-10-2 shows the lock modes.

Table IV-10-2. Lock Modes for Opened Device Locking

	Allow Mode During Open Call		
`input_output`	`anything`	`readers`	`nothing`
`input`	ir	r	w
`output`	iw	r-iw	w
`inout`	iw	r-iw	w

The kind of lock held by an opened device determines the operations allowed by other opened devices on a file. For instance, a write-lock disallows any other opens on a file, while a read-lock disallows all opens for writing to the file.

You can explicitly add and remove file-level locks associated with an opened device, using the `Record_AM.Ops.Lock_all` and `Record_AM.Ops.Unlock_all` calls. The `by_transaction` parameter must be set to false. You cannot remove the locks that the filing service automatically places at the time of doing the open.

All the locks associated with an opened device are released when the opened device is closed.

IV-10.1.13 File Administration Operations and Locking

The `File_Admin` package provides calls for managing files and indexes. Some of the file administration calls are transaction-oriented, and they interact with file processing on transaction-oriented files. File administration uses the same set of transaction locks to obtain isolation as those used by file processing.

File administration places write locks on files for `File_Admin.Create_file` and `File_Admin.Destroy_file` operations. If a transaction-oriented file is being read, the filing service will not allow a destroy file operation until the transaction that is reading the file has completed. File administration places intention-read locks on the file for the `Save_unnamed_file`, `Build_index`, and `Destroy_index` operations.

IV-10.2 Techniques

When you finish this section you should be able to write transaction-oriented applications that support different levels of consistency.

IV-10.2.1 Using Level 3 Consistency

Calls Required:

`Record_AM.Ops.Set_open_mode`
Sets the open mode parameter `level_3` to true.

`Record_AM.Keyed_Ops.Set_key_range`
Sets the index key range for the read.

`Record_AM.Ops.Read`
Reads a record from an opened device.

`Record_AM.Keyed_Ops.Update_by_key`
Updates the employee records by key values.

The following example starts a transaction that performs an indexed-sequential read and update of a clustered file using a unique b-tree alternate index. The transaction reads employee records using the `write_lock` lock mode, and updates the records with a new salary value. The transaction *does not* release the locks on its records as it reads them; it *holds* them until it resolves. Other transactions are not allowed to read or write the file until the transaction commits.

```
opened_file := Record_AM.Open_by_name(
    name         =>  file_name,
    input_output =>  Device_Defs.inout,
    allow        =>  Device_Defs.anything);

Record_AM.Ops.Set_open_mode(
    opened_dev => opened_file,
    mode_value => level_3_mode);
  -- Sets level 3 consistency.

...

loop
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      buffer_VA  => current_record_addr,
      length     => Employee_Filing_Ex.
          max_rec_size,
      lock       => Record_AM.write_lock,
      unlock     => Record_AM.no_unlock);
  -- Another caller cannot read or update
  -- the same record at any time.

  if current_record_VA.salary = 3_000.00 then
     current_record_VA.salary :=
         current_record_VA.salary + 300.00;

     Record_AM.Ops.Update(
         opened_dev => opened_file,
         modifier   => Record_AM.current,
         buffer_VA  => current_record_addr,
         length     => Employee_Filing_Ex.
             max_rec_size,
         timeout    => Record_AM.wait_forever,
         status     => null);
   end if;
end loop;
```

IV-10.3 Summary

- Locking is used to control concurrent access to the records, files, or key range values.
- There are transaction-oriented files and nontransaction-oriented files. Some files are more transaction-oriented than others.
- Transaction-oriented files can do transaction locking, short-term logging, and/or long-term logging.
- Transaction and opened devices can acquire locks. Transactions can lock files or records; opened devices can only lock files.
- Key range locking provides phantom protection.
- Locks held by child transactions can be passed to parents.
- Changes made in a file that uses short-term logging are rolled back if a transaction aborts.
- The long-term logging information is used to restore a file by rolling forward all the changes made to a file since a particular backup.
- A lock held by an opened device determines what kinds of opened devices and activities are allowed on a file.
- Nontransaction-oriented files do not support transaction locking, short-term logging, or long-term logging.

PROCESSING COLLECTIONS OF RECORDS 11

Contents

This chapter describes the filing service's support for processing collections of records.

Packages Used:

`Join_Interface`	Provides support for performing nested block joins of records from multiple opened devices. Only authorized users can use the calls in this package.
`Record_AM`	Provides device-independent I/O for records.
`Record_Processing_Support`	Provides support for processing large collections of records.
`Sort_Merge_Interface`	Provides calls for sorting and merging records from one or more input devices into a single ordered record stream.
`Trusted_Record_Processing_Support`	Provides additional support for processing collections of records. Only authorized users can use the calls in this package.

The record processing packages provide special calls for developing large record processing applications such as database filing systems. Figure IV-11-1 shows how you can customize a read call to do special processing of records in a file:

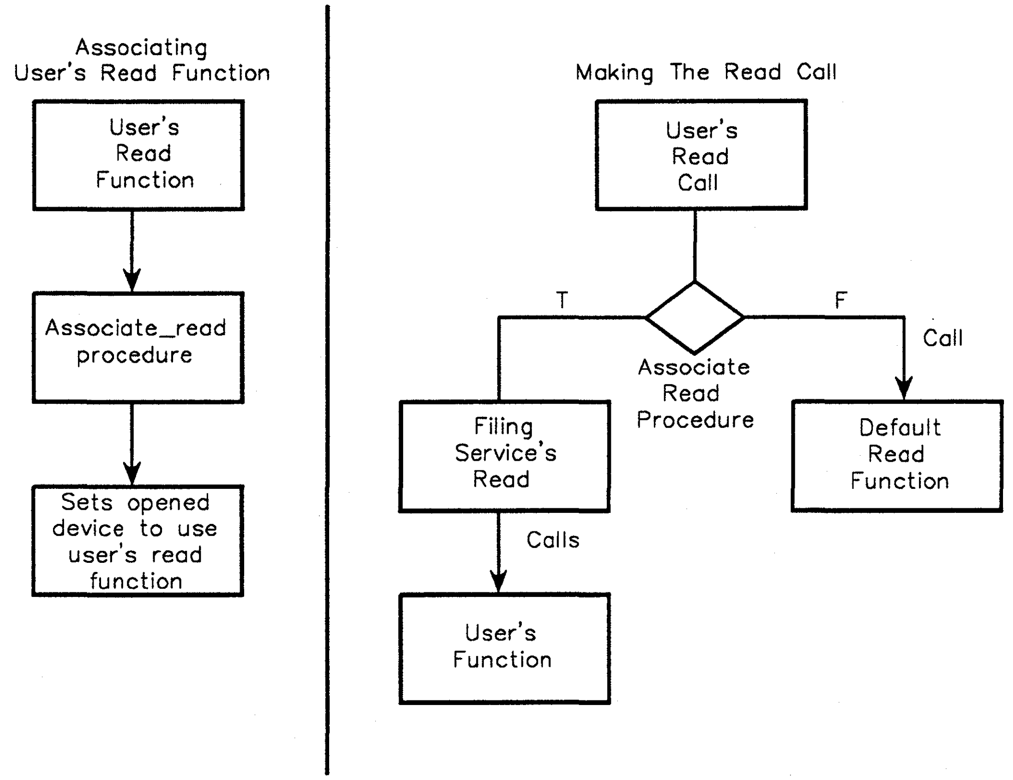


Figure IV-11-1. Customizing a Read Call

The `Trusted_Record_Processing_Support.Associate_read_procedure` associates an implementation for a read call with an opened device. When you read from the opened device, the filing service will use your implementation for the read call.

IV-11.1 Concepts

The operations for processing collections of records are more efficient than the record I/O techniques in Chapter IV-9 because:

- They minimize the processing time and the data transfer necessary between system file buffers and the user's address space.
- They can reduce the associated disk activity (seeks and latency).
- They minimize, in the distributed applications, the amount of communication necessary between nodes.
- They are efficient for processing collections of records.

IV-11.1.1 Reading Records

The calls in the record processing support packages let you read record fields and read data from a single index. You can use these calls together with the `Record_AM` read calls to create applications that efficiently read collections of records. You can use the record processing support calls together with these `Record_AM` read calls:

- `Record_AM.Ops.Read` - reads a record from an opened device
- `Record_AM.Keyed_Ops.Read_by_key` - reads a record from an opened device given a unique key value
- `Record_AM.Keyed_Ops.Read_key_value` - reads a key value form an opened device.

These record processing calls are for reading collections of records:

- `Record_Processing_Support.Set_oriented_read` - for reading all the records on a device
- `Trusted_Record_Processing_Support.Associate_read_procedure` - for customizing your read procedures.

The `Set_oriented_read` call lets you read a complete file with one call instead of having to write a call that does this. This is convenient when you want to read all the records on a device.

The `Associate_read_procedure` call lets you customize a read call with your own implementation of the call. To use the `Associate_read_procedure` call, you provide an implementation for the `Trusted_Record_Processing_Support.Process_record` (a subprogram template for a read procedure), and pass its subprogram type as the `read_function` parameter to `Associate_read_procedure`. Once you associate your read implementation with an opened device, the read call will use your implementation to read records from the opened device. Because this call is in the `Trusted_Record_Processing_Support` package, you must be an authorized user to use it.

NOTE

During a `Set_oriented_read` call, you can buffer the output records to one or two output streams. During the `Associate_read_procedure` call, returns a single buffer when the buffer is full and ready to be written.

IV-11.1.1.1 Record Streams

In record processing, you read collections of records called *record streams*. A *record stream device* is an opened device that is either:

- A sequential file that does not contain any indexes or use any transaction locking
- A pipe that is a stream of records.

You use record stream devices as temporary devices to hold the output from the record processing operations.

If you store *record IDs* in a *record stream device*, you create a *record ID stream*. A record ID stream is a record stream in which each record contains a record ID. Most of the record processing support calls use record ID streams to process records.

You can use the `Associate_record_ID_stream` call in the `Record_Processing_Support` package to associate a record ID stream with a file. Figure IV-11-2 shows an operation using record IDs and a file's primary data.

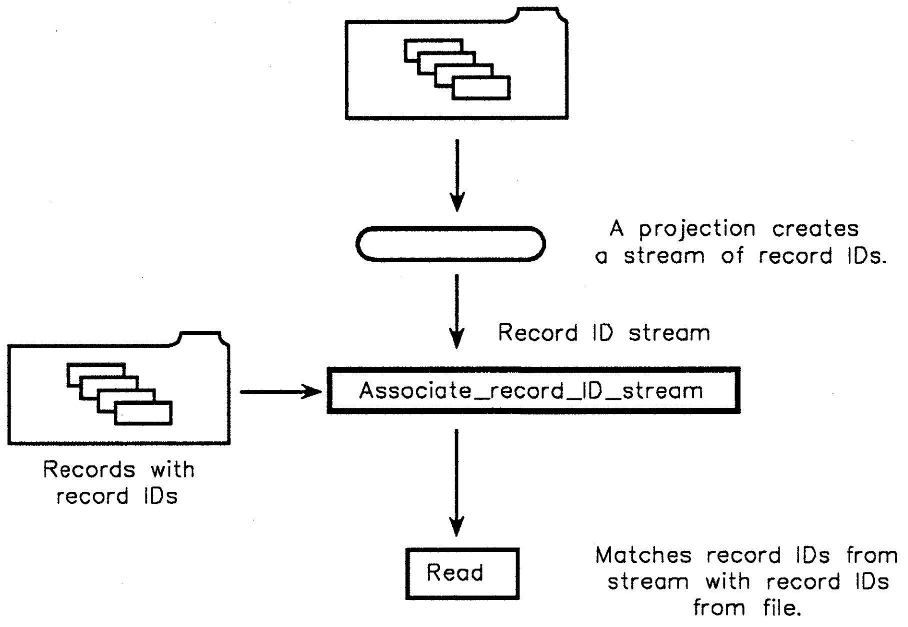

A projection creates a stream of record IDs.
Record ID stream
Associate_record_ID_stream
Records with record IDs
Read
Matches record IDs from stream with record IDs from file.

Figure IV-11-2. Associating a Record ID Stream with a File

You can direct a record stream or record ID stream to an output device using the `Record_AM` read calls or the `Set_oriented_read` call.

IV-11.1.1.2 DDefs and Record Processing Support

You can use *data definitions (DDefs)* to define the fields used for processing records. You can derive your DDef from the record DDefs of the file that is the specified input device for your application. See Appendix X-A for examples of derived DDefs.

IV-11.1.2 Updating Records

Using the basic record I/O calls, you update a record by reading and updating the entire record. But this is often inefficient when updating collections of records, because you usually only update a single field in each record rather than the entire record.

To efficiently update collections of records, you can use the `Record_AM` update calls together with the record processing support calls. You use these `Record_AM` calls for updates:

- `Record_AM.Ops.Update` - updates a record or record fields on an opened device
- `Record_AM.Keyed_Ops.Update_by_key` - given a unique key value, updates a record on an opened device.

Using the `Record_Processing_Support.Associate_update_fields` call, you can associate any number of fields to update with an opened device. Only the record fields you specify are updated. You can specify any number of fields to update with `fields_to_update` parameter. When you make an update call and specify a record, the filing service examines your buffer to find the new values for those fields. The filing service then logs the changes to the record and maintains any indexes that are affected by the update.

Variable-length fields may change in length when they are updated, and multiple value fields may have values added or deleted. The filing service automatically expands and contracts variable-length records to accommodate this.

IV-11.1.3 Database Operations

By combining the record processing calls and the `Record_AM` calls, you can implement many database and filing operations. The filing service provides special record processing support for these database operations:

- `selection` - extracts specific records (rows) from a given device.
- `projection` - extracts specific fields (columns) from a given device.
- `intersect` - builds a relation consisting of only the records appearing in two or more streams.
- `difference` - builds a relation consisting of all the records appearing in one stream, but not in any other given streams.
- `union` - builds a relation consisting of all records appearing in all of the given streams.
- `join` - combines the given streams by concatenating all of the records in the streams, creating a stream with all the combinations of records in the given streams.

Figure IV-11-3 shows the database management operations.

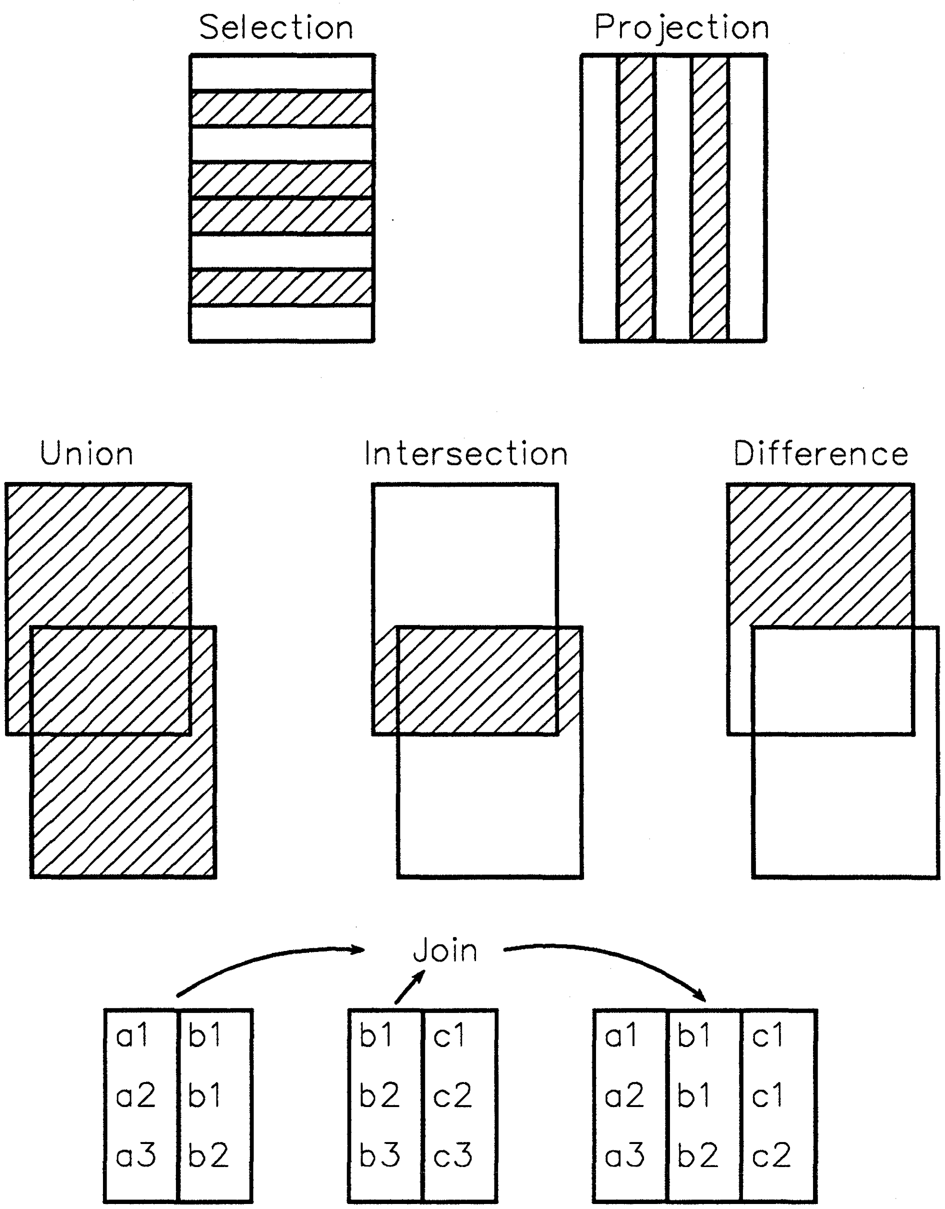


Figure IV-11-3. DBMS Operations

IV-11.1.4 Selection

Selection lets you select individual records or a collections of records from one or more opened devices. The read calls in the `Record_AM` package provide the basic calls needed to perform record selection. For example, you can use the `Record_AM.Keyed_Ops` call `Set_key_range` to set the range of key values to read in a file, and the `Record_AM.Ops` call `Read` to read all the employee records in a given department. You can do this without reading the records for the employees in any other departments. See Chapter IV-9 for examples of how to do this.

The Record_Processing_Support package provides this call to optimize record selection:

- Associate_record_ID_stream - for associating a record ID stream with an opened device with read rights.

The Associate_record_ID_stream call helps you select records by their record IDs. This call associates a record ID stream with the opened device you want to read. When you call Set_oriented_read, the record IDs of the records you are reading are automatically output to the associated record ID stream.

IV-11.1.4.1 Customizing a Selection

You can use the calls in the Trusted_Record_Processing_Support package to customize the implementation of your selection operation. This package provides two calls for optimizing record selection:

- Associate_index_selection_function - for customizing your own index selection function.
- Associate_read_procedure - for customizing your own read call for record selection.

These calls let you associate a selection implementation with an opened device. Whenever a read call is made to that opened device, the filing service uses the associated read implementation for the device to do the read. You can only use these calls if you are an authorized user.

IV-11.1.4.2 Using the Associate_index_selection_function Call

To use the Associate_index_selection_function call, you must provide the opened device the selection function is associated with and a *subprogram type* for the selection function. (See the System package for a description of the BiiN™ Ada System.subprogram_type type.) The selection function must be an implementation of the Trusted_Record_Processing_Support.Entry_qualifies call that does record selection with an index. You pass the subprogram type for your implementation of this call in the Associate_index_selection_function call's selection_function parameter. You can do this with the BiiN™ Ada subprogram_type attribute. When you execute a read on the associated device, the read will automatically use your Entry_qualifies implementation to perform the read.

IV-11.1.4.3 Using the Associate_read_procedure Call

To use the Associate_read_procedure call, you provide an implementation for the Trusted_Record_Processing_Support.Process_record that does record selection, and you associate your implementation with the opened device you are reading from. Your call returns the selected part of the record when records are read from the device.

You can also use Associate_read_procedure to do record projection or *both* record selection and projection in a single procedure.

IV-11.1.5 Projection

Projection is a database operation that extracts record fields (columns) from an opened device. This is most efficient for processing fields in records, rather than entire records. For example, the projection operation could get certain employee ID numbers in an employee database by accessing only the record fields with the ID numbers instead of the entire employee record.

You can set up projection operations for reading or updating fields in an opened device using either primary or index projections. You can use these `Record_Processing_Support` calls for projection operations:

- `Associate_primary_data_projection` - associates a projection of the primary fields with an opened device.
- `Associate_index_projection` - associates a projection of the index fields with an opened device.

To create a projection operation, you must first associate a projection with an opened device. Figure IV-11-4 shows a record projection operation that reads only the requested fields in an indexed file.

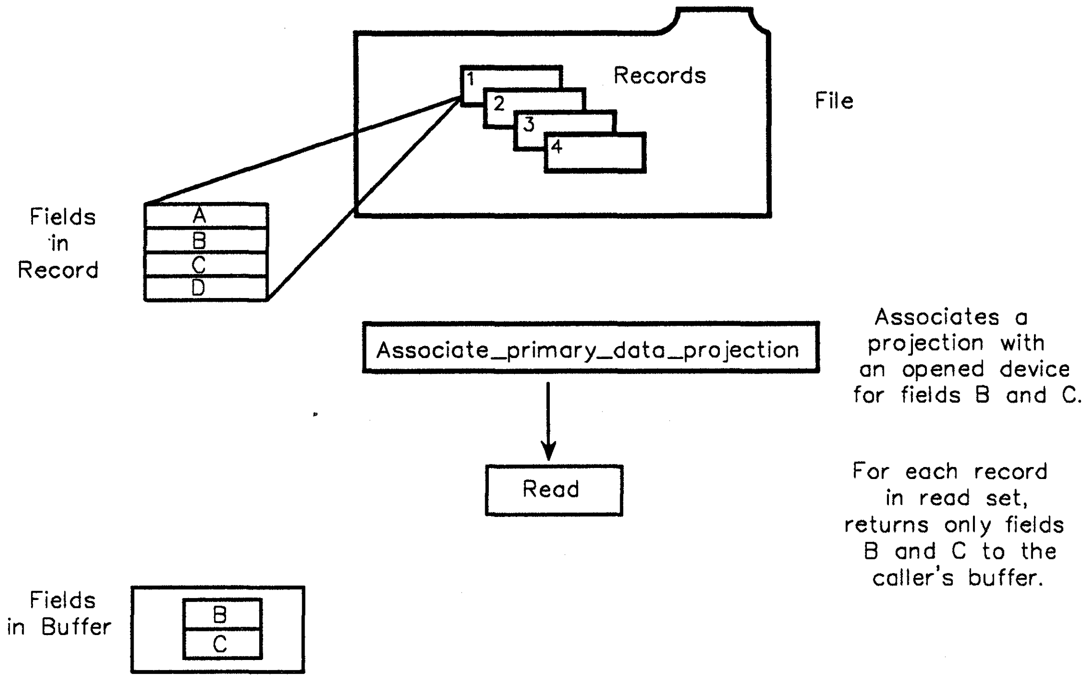

Records
File
1
2
3
4
Fields in Record
A
B
C
D
Associate_primary_data_projection
Associates a projection with an opened device for fields B and C.
Read
For each record in read set, returns only fields B and C to the caller's buffer.
Fields in Buffer
B
C

Figure IV-11-4. A Primary Data Projection

The projection operation is created using the `Record_Processing_Support.` `Associate_primary_data_projection` call to associate the projection operation with a read call for an opened device. After setting up a projection, any read call on the opened device automatically projects to the output stream the fields in each record you specify in your projection implementation.

IV-11.1.5.1 Using the `Associate_primary_data_projection` Call

The `Associate_primary_data_projection` lets you specify the primary data fields you want to project, and whether you want the output with record IDs. You specify your output with these parameters:

`primary_fields`	The fields to be extracted from the primary data of the record being read.
`record_ID_output`	The record ID of the record being read is included in the output.

If you specify both parameters, the output appears in this order:

[record ID, primary fields]

This call creates a projection of an opened device's primary data area. Each time a read call reads a record from the opened device, the primary fields are projected to the output device associated with the read call. The projection writes the primary fields in the order you specify, with fixed length fields first followed by the variable length fields.

IV-11.1.5.2 Using the `Associate_index_projection` Call

To project only key values, you can use the `Associate_index_projection` call. You specify your output with these parameters:

`key_value_output`	The specified key to be extracted from the index data of the file being read. If this key does not exist, the read produces erroneous results.
`record_ID_output`	The record ID of the record being read is included in the output.

When you specify both parameters, the output appears in this order:

[record ID, key value]

With this call, you can only access the secondary data area containing the index's key values during a read. The rest of the record is inaccessible. For each record, a record ID and/or the key value is returned to the output device. Prior to making this call, you must set the index key range with a call to `Record_AM.Keyed_Ops.Set_key_range`.

You can also use `Trusted_Record_Processing_Support` call `Associate_read_procedure` to create your own projection operation.

IV-11.1.6 Difference, Intersection, and Union

Three calls in the `Record_Processing_Support` package manipulate streams of record_IDs to provide low-level support for the database operations *intersection*, *union*, and *difference.*

- `Intersect_record_IDs` - forms the intersection of two record ID streams.
- `Union_record_IDs` - forms the union of two record ID streams.
- `Difference_record_IDs` - forms the difference of two record ID streams.

These calls let you perform intersection, union, or difference on the record ID streams created by a selection or projection operation.

IV-11.1.7 Interaction Between Record Processing Calls

Certain combinations of the record processing calls are incompatible when you use them on the same opened device. A record processing call disables an incompatible call that is active on the same device. Table IV-11-1 lists the record processing calls and the calls they are incompatible with.

Table IV-11-1. Interaction of DBMS Calls

Call	Disables these calls
Associate_read_procedure	Associate_primary_data_projection Associate_index_projection
Associate_primary_data_projection	Associate_read_procedure Associate_index_projection
Associate_index_projection	Associate_read_procedure Associate_primary_data_projection
Associate_index_selection_function	Associate_record_ID_stream
Associate_record_ID_stream	Record_AM.Set_key_range Associate_index_selection_function
Record_AM.Set_position	Record_AM.Set_key_range Associate_record_ID_stream Associate_index_selection_function
Record_AM.Set_key_range	Associate_record_ID_stream Associate_index_selection_function

IV-11.1.8 Joins

The join operation lets you join together records from multiple opened devices into one opened device based on the values of common fields. Figure IV-11-5 shows a join using two input files.

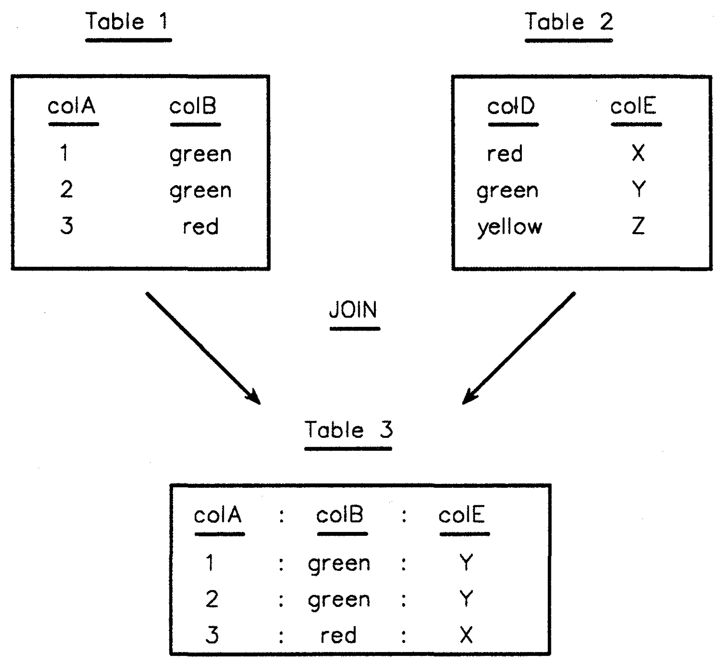


Figure IV-11-5. A Join Operation

The `Join_Interface` package provides two calls:

- `Join` - the call that joins the records from the input devices producing a single stream of joined records and an optional stream of alternate records.
- `Block_join` - the function template you provide for joining blocks of records.

The `Join` call performs the low-level block access to data in the input devices, and in the process calls your implementation of the `Block_Join` call that actually performs the join.

When you make a call to `Join_Interface.Join`, you specify a list of input devices in the `participating_devices` parameter. If you only specify one input device, you only access the primary data in the input device. You reserve the buffers for the join with the `buffers_to_reserve` parameter. You specify the record streams for the output in the `join_output` and `alternate_output` parameters. The `user_info` parameter returns the process-specific information for the join procedure.

The parameter `join_procedure` specifies an implementation of the `Block_join` function. This parameter is the subprogram type of your implementation of the join operation. You obtain the subprogram type value by applying the BiiN™ Ada `subprogram_type` attribute to the block join implementation you want to use.

`Join` calls your implementation of `Block_join`. Your implementation of this call:

- Joins the sets of records
- Fills the output buffers
- Requests the next set of input from the `Join` call.

The `Block_join` operation gets the set of records you want to join from the `records` parameter. This parameter is the list of record locations for each input device, and contains null ADs the first time `Block_join` is called. `Block_join` completes the join and fills the output buffers. `Block_join` then returns a `communication_block_VA` type that contains the next block list, the output buffers, and a request for the next set of records. You must instantiate the `communication_block_VA` data structure prior to the `Block_join` call. You can pass this data structure into `Block_join` as part of the `user_info` parameter. If there are more blocks to join, the process is repeated.

The join operation only provides data from files; any selections associated with files cannot be active during a join. To select records for a join, you must do a selection on the file to produce a record ID stream, and pass the record ID stream to the `Join` call.

IV-11.1.9 Sorting and Merging

You use sorting and merging to produce one ordered record stream from one or more record streams. Figure IV-11-6 shows how records can be sorted and merged.

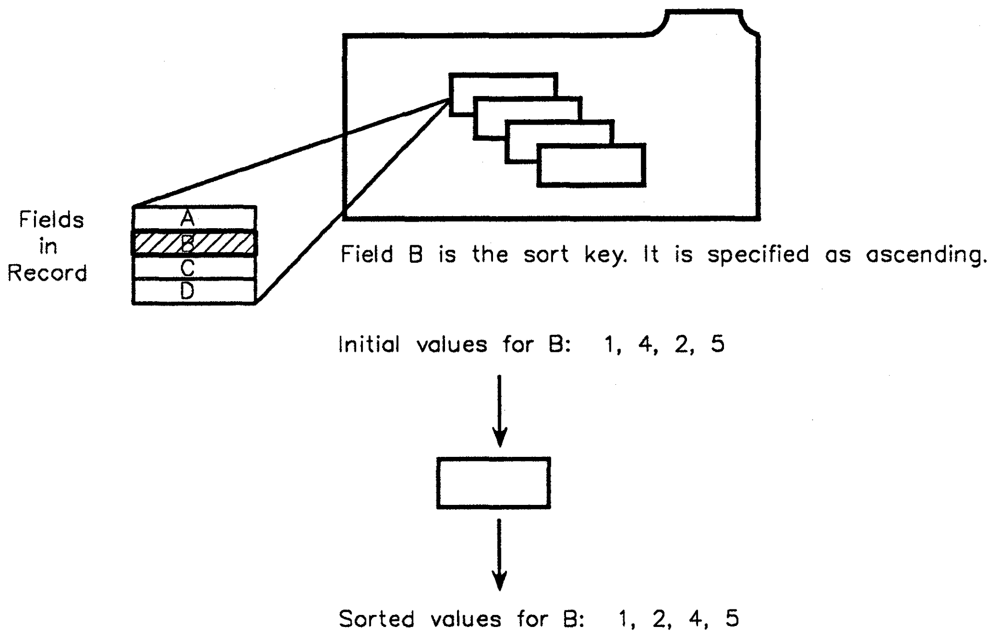

Fields
in
Record
A
B
C
D
Field B is the sort key. It is specified as ascending.
Initial values for B: 1, 4, 2, 5
Sorted values for B: 1, 2, 4, 5

Figure IV-11-6. Sorting and Merging Records

The Sort_Merge_Interface package provides these calls for sorting and merging record streams:

- Sort - sorts a single record stream and produces an ordered stream of records.
- Sort_merge - sorts and merges an arbitrary number of input record streams, and produces a single ordered stream of records.
- Special_collation_sort_merge - sorts and merges records with arbitrary string fields ordered by a collating sequence that you specify.

You can only use Sort_Merge_Interface calls to sort opened structured files or pipes. If you attempt to sort records using an unsupported device, the sort/merge service does not return exceptions. You cannot use Sort_Merge_Interface calls with transactions. If you try to use these calls with transactions, the operations will not be atomic and exceptions will not be raised.

IV-11.1.9.1 Sorting Records

To sort records, you specify the input device (where you are reading the records from) and the output device (where you want to write the records to). Because the sort calls use the Record_AM.Ops calls Read and Close to access input devices, you must assure that the input device is opened to allow physical-sequential or indexed-sequential reading of records. The reads begin at the current position of the input device and continue until Device_Defs.end_of_file is raised. After the sort, the input and output devices remain open.

The Sort call sorts records from a single opened input device. The Sort_Merge and the Special_collation_sort_merge calls can sort and merge records from one or more input devices. The Sort_Merge and Special_collation_sort_merge calls take the parameter input_devices, a variable-length array of input descriptors. Each descriptor specifies an opened input device (file or pipe) and a boolean indicating whether or not the records are presorted.

You select the level of concurrency you want for an input device. An opened input device can be opened for exclusive use (no concurrency) or shared use (dirty read). Generally, you should pass in an exclusively opened device (no concurrency).

You can also specify tuning options with the `tuning_opts` parameter. These options include options for gathering statistics, for indicating the volume set to use for work files, and for indicating specific work files to be used during the sort.

Any read options or record processing operations associated with an input device are passed along with the device.

IV-11.1.9.2 Sort Ordering

The sort ordering is normally based on values of a sort/merge key composed of typed fields that you must lay out using a DDef (see Chapter IV-8 for a description of how to create DDefs). You must assign each field of the DDef in the sort/merge key a `pi_descending` value of either true (descending) or false (ascending). The default value is false.

The ordering of multiple input devices in the `input_devices` parameter can determine the ordering of records with duplicate key values when those records appear on different input devices. (Options for ordering records with duplicate key values, by insertion time or by record ID, only hold within a single file.) For example, you specify three input devices in the order `A`, `B`, and `C`. Device `A` has a record `a`; device `B` has the records `b1` and `b2`; device `C` has record `c`. All these records have identical key values. The ordering of the input devices determines the output, which can be either `a/b1/b2/c` or `a/b2/b1/c`. (If `B` was presorted or you specify `stable_sort`, the relative ordering of `b1` and `b2` are the same in the output as the input. Otherwise, their ordering is arbitrary.)

The `Special_collation_sort_merge` call takes the additional parameter `alternate_CS` for specifying an alternate collating sequence for string fields. (See the string collating sequences required by the runtime systems of certain languages such as COBOL.) The ordering of each string field is normally performed using the ASCII collating sequence. You must specify any alternate collating sequence. A collating sequence is an array of 256 bytes where `CS(x) = N` implies that a byte with a value `x` should be ordered as if it held the value `N` in the standard numeric ordering. A collating sequence can map more than one byte value to the same ordering value, that is, `CS(x) = N = CS(y)` is allowed.

IV-11.1.9.3 Stable Sorts

You can optionally specify a stable sort with the `stable_sort` parameter. A stable sort preserves the original ordering of records with duplicate key values within a file. This kind of sort is typically inefficient than an unstable sort.

The sort operations also support unique sorts, which are sorts that delete duplicates from the final output.

IV-11.2 Techniques

After reading this section, you will be able to:

- Select a set of records
- Set up a projection using an index
- Join records from two different opened devices
- Sort records from a single file
- Sort and merge records from two files.

The examples used in the following sections are in the example packages called `DBMS_Ex` and `Join_Ex`. See Appendix A for complete listings.

IV-11.2.1 Selecting a Set of Records

To select records in a file you either use a `Read` call or a `Set_oriented_read` call.

Calls Used:

`Record_Processing_Support.Set_oriented_read`
Sequentially reads an opened file and sends all data to an output device until end-of-file or a user interrupt.

Typically, you associate a read implementation with the device you you want to do the selection from, and then do the read. The read implementation you supply actually does the selection. This example uses the `Set_oriented_read` call to automatically read a file sequentially until the end-of-file. After reading the records, they are written to the output devices you specify instead of being copied into a buffer. This allows two output streams to be created from a single pass of the input file (for example, the *selected* records and the *rejected* records). This is more efficient for applications in which the records read from a file are destined to be immediately rewritten to another device.

The `Set_oriented_read` call takes two locking parameters:

`lock`	Specifies the lock mode of *every* record read.
`unlock`	Specifies the unlock mode of *every* record read.

These parameters are ignored if the file being read is not transaction-oriented.

The `unlock` parameter is the same as the one specified in `Record_AM.Ops.Read`. Records can be incrementally unlocked as the set of records is read (each previous locked record is unlocked in the specified mode).

The `timeout` parameter is also the same as the one specified in `Record_AM.Ops.Read`. The timeout is the maximum time to block waiting for a single record lock.

```
    Trusted_Record_Processing_Support.Associate_read_procedure(
        opened_dev          => opened_file,
        user_info           => System.null_address,
        read_procedure      => read_procedure);


    Record_AM.Keyed_Ops.Set_key_range(
        opened_dev     => opened_file,
        index          =>
            Employee_Filing_Ex.dept_index_name,
        select_range => (
            start_comparison => Record_AM.inclusive,
            start_value      => start_key_descr,
            stop_comparison  => Record_AM.inclusive,
            stop_value       => stop_key_descr));

    Record_Processing_Support.Set_oriented_read(
        opened_dev      => opened_file,
        modifier        => Record_AM.next,
        output_device   => Process_Globals_Support_Ex.
            Get_standard_output,
          -- Normally defaulted.
        alt_output      => System.null_word,
        no_record_lock => false,
        lock            => Record_AM.read_lock,
        unlock          => Record_AM.no_unlock,
        timeout         => Record_AM.wait_forever);
```

IV-11.2.2 Using Projection on an Index

You can project the values for selected fields in records during a read.

Calls Used:

`Record_AM.Keyed_Ops.Set_key_range`
Sets two boundaries within an index.

`Record_Processing_Support.Associate_index_projection`
Sets up a projection of the values for specified key fields in a composite index during a read.

`Record_AM.Ops.Read`
Reads only the previously specified field values in each record that is read.

```
-- Filters out all fields except those specified
-- in the DDef.
Record_Processing_Support.
    Associate_primary_data_projection(
        opened_dev       => opened_file,
        record_ID_output => false,
        primary_fields   => projection_DDef_ref);


loop
  -- Only reads the fields specified in
  -- the DDef.
  bytes_read := Record_AM.Ops.Read(
      opened_dev => opened_file,
      modifier   => Record_AM.next,
        -- Normally defaulted.
      buffer_VA  => current_record_addr,
      length     => System.ordinal(
          Employee_Filing_Ex.max_rec_size));

  -- DO ANY NEEDED PROCESSING HERE.

end loop;
```

IV-11.2.3 Joining Records from Two Devices

You join records from two (or more) different opened devices in blocks. The system does the join of the devices (files); you supply the join semantics.

Calls Used:

`Join_Interface.Block_join`
Provides a template for a user-supplied function that joins blocks of records in memory buffers.

`Join_Interface.Join`

```
Join_Interface.Join(
     participating_devices => join_devices,
     buffers_to_reserve    => buffer_reservation,
     user_info             => u_info'address,
     join_procedure        =>
         Join_ex'subprogram_value,
     join_output           => out_file,
     alternate_output      => System.null_word);
```

IV-11.2.4 Sorting Records in a File

You can easily sort records from a single opened input device to an opened output device.

Calls Used:

`Sort_Merge_Interface.Sort`
Sorts the records from a single input device to produce an ordered stream of records.

You specify the opened input device (from which the records are to be sequentially read) and the opened device (to which the sorted records are to be sequentially written). You must assure that the input device is opened in a manner that allows physical-sequential reading of records and provides for the desired level of concurrent use of the device. Physical-sequential reads begin at the current position of the input device and continue until `Device_Defs.end_of_file` is raised. After the sort, the input and output devices are automatically closed.

```
Sort_Merge_Interface.Sort(
    input_device  => inventory_file,
    DDef          => inventory_DDef_ref,
    output_device => Process_Globals_Support_Ex.
        Get_standard_output,
    stable_sort   => true,
    tuning_opts   => Sort_Merge_Interface.
        no_tuning);
```

You have the option to specify whether the sort is *stable*, that is the original ordering of records with duplicate key values within a file is preserved. A stable sort is typically inefficient than a corresponding unstable sort. You also have tuning options; these include options for gathering statistics, for indicating the volume set to use for work files, and for indicating specific work files to be used during the sort.

IV-11.2.5 Sorting and Merging Records from Two Files

In addition to sorting records from a single opened input device, you can also sort and merge records from an arbitrary number of sorted or unsorted input devices.

Calls Used:

`Sort_Merge_Interface.Sort_merge`
: Sorts the records from a single input device to produce an ordered stream of records.

`Sort_merge` takes the same parameters as `Sort`. In addition, it takes the parameter `input_devices`, which is a variable-length array of sort-merge input descriptors. Each descriptor specifies an opened input device that supports `Record_AM.Ops.Read` and a boolean indicating whether or not the records are presorted. The ordering of multiple input devices in this array determines the ordering of records with duplicate key values when those records appear on different input devices. (Options for ordering records with duplicate key values -- by insertion time or by record ID -- only hold within a single file.)

```
-- Perform the sort-merge.
Sort_Merge_Interface.Sort_merge(
    input_devices => sort_input_array,
    DDef          => sort_DDef_ref,
    output_device => Process_Globals_Support_EX.
        Get_standard_output,
    stable_sort   => true,
    tuning_opts   => Sort_Merge_Interface.
        no_tuning);

```

IV-11.3 Summary

- Use record processing support for large applications for which you want to perform operations on only selected data in files.
- Record processing support operations are associated with opened devices. Selected field values (in a file's primary data area), record IDs, and/or index key values can be supplied with each individual record during reads. Selected field values can be updated.
- DDefs are used to define the field values used by filters. These DDefs should be derived from the record DDef used to define a file's record layout.
- A record stream device can be a non-transaction-locking sequential file (without indexes) or a record-oriented pipe that consists of record IDs. Record stream devices can be associated with an opened device for a file so that records (or records fields) in the file are selected on the basis of the record IDs in the record stream device.
- Record streams can be used to perform the relational database operations difference, union, and intersection.
- You can write your custom read and custom index selection routines to associate with open devices. However, this is not possible for most users.
- You can write your own block join routine to bring together information contained in multiple files into a single file based on the values of common fields. Your custom procedure gets record location information contained in currently processed blocks of data. This, also, is not available to most users.

Part V
Human Interface Services

This part of the *BiiN™/OS Guide* discusses services used for interacting with users.

Understanding Human Interface Services
Basic concepts of Human Interface Services.

Creating a BiiN™ Application
An example of a simple, complete application program which uses Human Interface Services.

Building New Commands
Reading and processing program-defined invocation and runtime commands.

Programming with Command Language Variables
Reading, creating, and setting job and session variables.

Programming with Menus
Displaying menus and processing menu selections.

Understanding Forms
Describes data entry/display forms and the form service.

Programming with Forms
Displaying and processing forms for data entry and display.

Programming with Reports
Setting up and printing reports from data records.

Human Interface Services contains the following services and packages:

Command Service
```
CL_Defs
Command_Execution
Command_Handler
Environment_Mgt
Help_Text_Adm
```

Form Service
```
Form_Defs
Form_Handler
```

Report Service
```
Report_Handler
```

Part V Overview

UNDERSTANDING HUMAN INTERFACE SERVICES 1

Contents

This chapter provides a general overview of the BiiN™ system's user interface software.

The next eight chapters discuss how to create a BiiN™ program, including interacting with the user: using commands, variables, menus, and forms for input, and variables, forms and reports for output.

Figure V-1-1 shows the relationships between the five Human Interface Services and an application program.

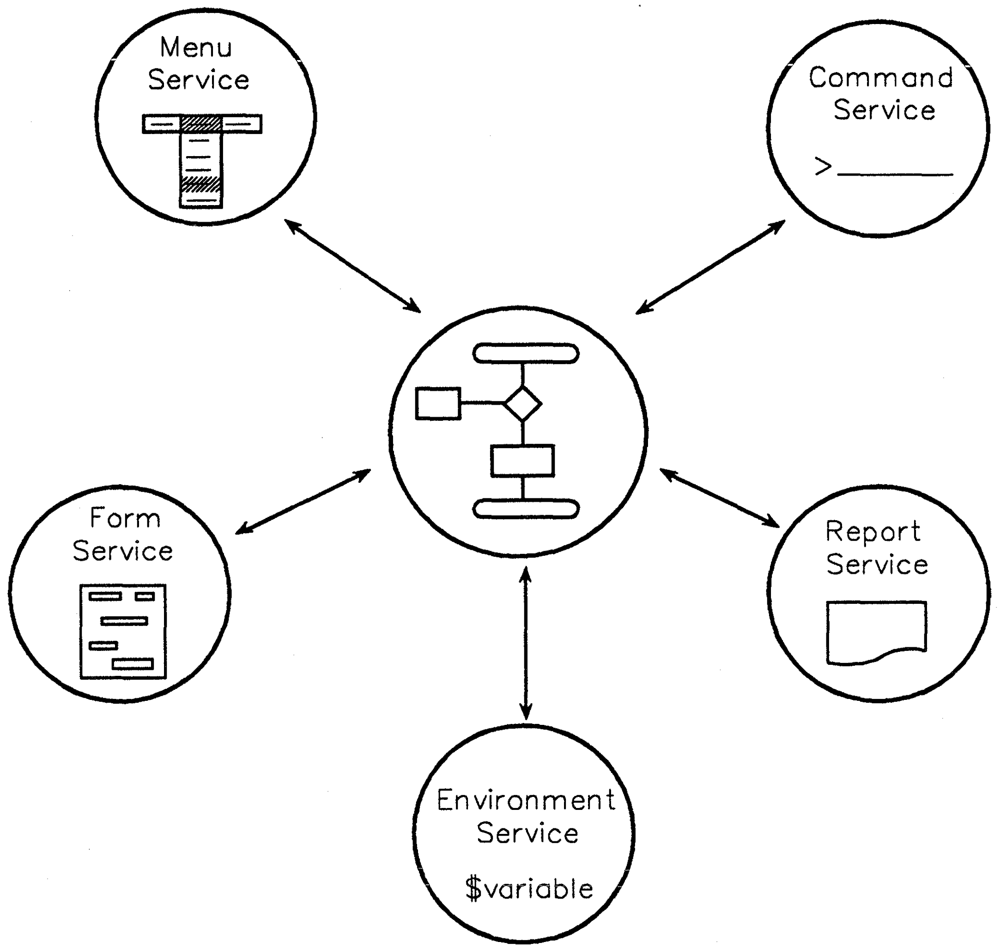


Figure V-1-1. Human Interface Services and a BiiN™ Program

- Read Chapter V-2, "Creating a BiiN™ Application", to understand the relationships between an application program and various *service areas*, including Human Interface Services.
- Read Chapter V-3, "Building New Commands", to use the *command service* to create and use program-specific commands. The *BiiN™ Command and Message Guide* describes how to create command definitions interactively.
- Read Chapter V-4, "Programming with Command Language Variables", to use the *environment service* to read and write CLEX user, and job variables.

- Read Chapter V-5, "Programming with Menus", to use the *menu service* to create menus and read menu selections.
- Read Chapter V-5.3, "Understanding Forms", for the concepts necessary to use the *form service* for user input and data display.
- Read Chapter V-7, "Programming with Forms", to use forms, change a form's appearance and structure, and get data from and put data into forms.
- Read Chapter V-8, "Programming with Reports", to use the *report service* to write reports from data records.
- Read Chapter **TBD**, "Interacting with Users in Standard Ways", for guidelines on how to use Human Interface Services in a manner consistent with the BiiN™ system's own programs and utilities.

Chapter II-5 describes how to create and display messages, including help messages. The `manage.messages` utility creates and maintains files of messages; it is described in the *BiiN™ Command and Message Guide.*

Human Interface Services is composed of five service areas:

Command Service	Manages the command interface. Also manages command language (job and session *environment*) variables.	
	`CL_Defs`	Contains declarations used by the command service, for processing command language (`CL`) arguments and variables.
	`Command_Execution`	Contains a procedural interface to command execution.
	`Command_Handler`	Contains operations for reading and processing program commands and arguments.
	`Environment_Mgt`	Contains operations to get, set, or remove local and global environment variables.
	`Help_Text_Adm`	Manages command and form help texts.
Form Service	Manages forms, for structured data input and display.	
	`Form_Defs`	Defines types and constants used by the `Form_Handler` package.
	`Form_Handler`	Provides calls to process, control, and change forms.
Report Service	Generates reports; formats data records for display.	
	`Report_Handler`	Provides calls for initializing and printing a report.

V-1.1 Concepts

Human Interface Services provide methods of interacting with human users. The model is that a user controls an application program to create, retrieve, update, and display information.

To control a program, a user has a choice of commands or menus:

commands	Provide a set of free-form commands, each with appropriate arguments and default values. The command service provides calls to prompt the user to enter a command, returning the command's index and name, and its argument values.
menus	Provide a list of choices. The menu service provides calls to enable the user to make a menu selection, then to read the selection indexes.

The environment service provides calls to read command language variables, including job variables in active memory, and user and system variables in passive store. Variables can be created and set in active memory.

The form service provides calls to display a data entry form, prompting the user to enter a complete, consistent data record.

The report service provides a call to initialize a report, given its definition and input and output devices. Another call prints the initialized report.

Some Human Interface Services are based on data definitions, or *DDefs*. There are DDefs for commands, menus, forms, and reports. The term *definition*, as in a *command definition*, is used throughout this chapter to refer to the underlying DDef. For more information on DDefs and specific DDef properties, see the `Data_Definition_Mgt` and `DDF_Utility_Support` packages in the *BiiN™/OS Reference Manual.*

Once a file's record definition (DDef) has been created, it can be used to automatically generate a default data entry form and a default report.

V-1.1.1 Why Use Human Interface Services?

These services provide consistent, standard methods of interacting with the user. A BiiN™ user, on invoking your program, is already familiar with the user interface.

The BiiN™ OS, and the system utilities, call these services to interact with the user.

The primary value of Human Interface Services is that each service performs its own processing, returning a complete result to your program's request.

Extensive "help" is provided by the services. You define the help texts, and attach each text to the appropriate point (a program, a command, a menu item, or a form's field). The help text is displayed upon the user's "help" request.

For example, after your program displays a form, the user can move through the data fields, request form-specific help, and be prompted to correct invalid or inconsistent entries, all without your program's intervention. When the user completes the form, valid input data is then available.

V-1.1.2 Utilities

Several utilities are used with Human Interface Services, to set up variable groups, and to create command, menu, form, and report definitions.

Figure V-1-2 shows the relationship between a service's utility, the definition it produces, and the service your program calls.

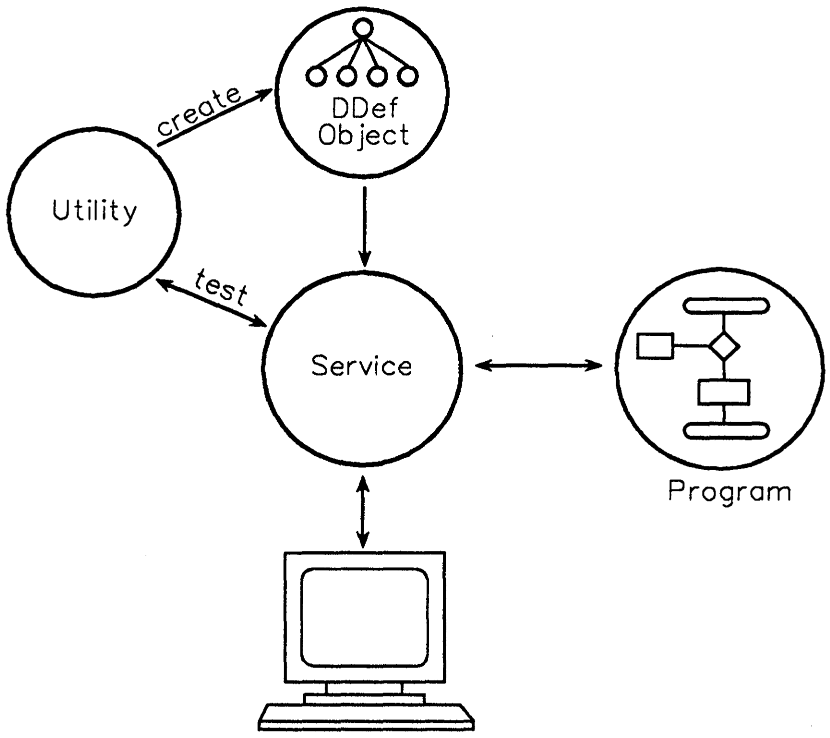


Figure V-1-2. Utility, Data Definition, and Service

Command Service Utility

`manage.commands`	Creates both invocation command and command set definitions, defining all commands and their arguments. See the *BiiN™ Command and Message Guide* for more information about using the `manage.commands` utility to create command definitions.
`manage.variable_group`	Manipulates groups of variables, creating, modifying, and storing *variable groups*. The environment service then reads and writes these variables; the environment service does not provide calls to create group variables.

Form Service Utilities

`create.standard_form`	Creates a default form definition, given a file's record definition. The default definition can then be used with the form service, or revised using the `edit.form` utility.
`edit.form`	Creates and modifies form definitions, including all fields and values. See the *BiiN™ Systems Form Editor Guide* for more information on defining forms.

`edit.key_map`	Edits a form's keyboard mapping. This becomes part of a form's definition.

Report Service Utilities

`create.standard_report`	Creates a default report definition from a file's record definition. The default definition can then be used with the report service, or revised using the `edit.report` utility.
`edit.report`	Creates and modifies report definitions, including all fields, column headings, and control breaks. See the *BiiN™ Systems Reports Guide* for more information on defining reports.

Help texts are defined within each service's utilities. Messages are defined with the `manage.messages` utility, or with the `manage.messages` command set in the `manage.program` utility. See the *BiiN™ Command and Message Guide* for more information on defining help texts and messages.

V-1.1.3 Command Service

The command service is used to parse and execute commands entered to your program. CLEX uses this interface for its own commands.

Figure V-1-3 shows the relationship of a BiiN™ program to the command service.

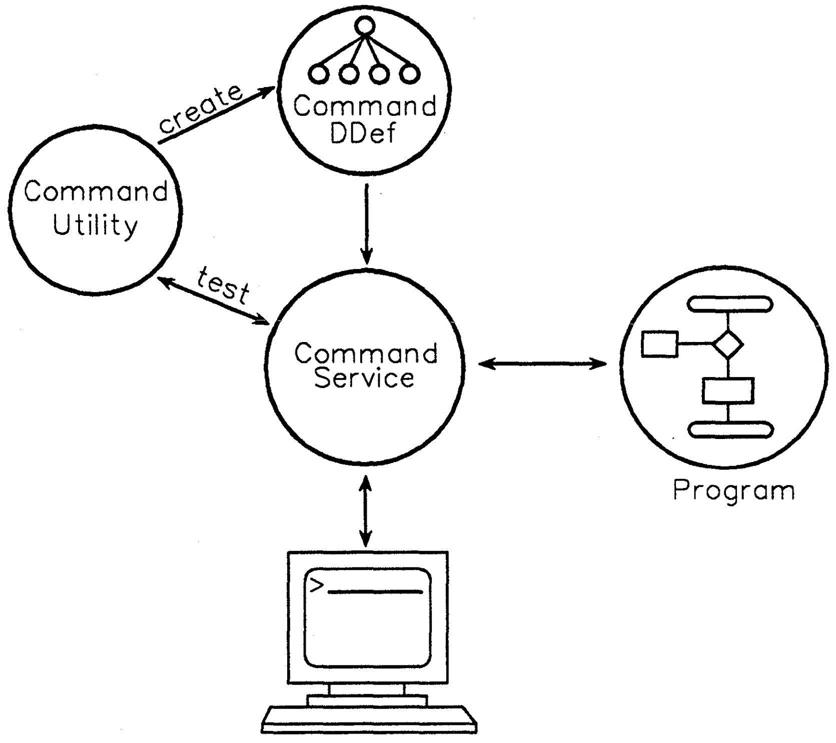


Figure V-1-3. BiiN™ Application Program and the Command Service

V-1.1.3.1 Command Concepts

- New commands are defined using the `manage.command` command set in the `manage.program` utility - see the *BiiN™ Command and Message Guide.*
- The command service uses the command definitions - to help the user correctly enter and complete a command.
- There are three types of commands - processed by the command service: *built-in commands, CLEX commands*, and *program-defined commands.*
- *built-in commands* Part of, and processed by, the command service itself. *Built-in runtime* commands directly perform some action, such as setting a variable (`set.variable`). *Built-in control* commands control the logical flow of commands (`if / then / else / endif`).

  CLEX commands Commands specific to the *command language executive* (CLEX).

  program-defined commands
  Each program has an *invocation command*, which is entered to CLEX to invoke (execute) the program. Programs using the command service define their own *runtime commands*, which control and are implemented by the program.
- A command consists of up to three parts:

  ```
  command.name    [:argument=value]...
                  [::control_option=value]...
  ```

  `command.name` The name of the command.

  `:argument = value`
  Zero, one, or more arguments may be part of a command.

  - *Arguments have a name, a type, and a value* - an argument name is a string of characters, preceded by a colon (":", for example, `:argument_name`). The argument type is one of seven: boolean, integer, range, string, string list, pointer, or "derived". A derived argument's actual value may be any of the other types. The argument value's type must match the defined argument type.
  - *Arguments may be mandatory or optional* - mandatory arguments must be entered with the command name. Optional arguments may be entered to specify an *argument value* other than the default value, if any, defined with the command.

  `::control_option = value`
  There are several control options defined in the command language, used to request input/output redirection, background execution, and so forth.
- *A sequence of commands can be stored in a command file* - for inclusion into the command input stream with the built-in `include.command` command. A command file can be made into an executable script by using the `make.script` utility.
- *An optional command history can be used* - to record commands entered. One or more recorded commands may be re-executed. The built-in command `list.last_commands` shows the recorded commands; one or more of these recorded commands are redone with the `redo.last_commands` command.
- *There is support for BiiN™/UX invocation command conventions* - such as `argv, argc, envp`; see the *BiiN™/UX User's Guide.*

Why Use Commands?

Commands provide an easy, standardized way to interact with the user:

- Commands provide a common entry format.
- Commands are entered and confirmed without program intervention. A complete, correct command is then available to the program.

Programs Using the Command Service

Most programs using the command service will be new utilities.

Commands are defined during program development. Each distinct function that the program performs should have its own command.

Commands are grouped in command sets. Separate command sets may be defined for different program tasks, or for different user groups.

Not all programs are suitable for command-oriented input. There are several alternatives to using commands to control your program:

menus	Read menu item selections; see Chapter V-5.
keyboard input	Read the keyboard directly; see Chapter IV-5.
graphics input	Read the mouse's position and state. See **TBD.**

V-1.1.3.2 Command Summary

- Commands provide a consistent user control mechanism, used by all BiiN™ utilities.
- A command consists of the command name, arguments if any, and optionally one or more control options.
- The command service requires the user to enter a complete command; help is available for each command and each argument. Calls in the `Command_Handler` package return the entered command values.
- New commands are defined with the `manage.commands` command set in the `manage.program` utility. Each command definition is stored under a pathname.
- Some commands are built-in; other commands are defined by programs using the command service (including the CLEX program itself).
- Built-in commands are part of every command set. These commands are intercepted and processed by the command service itself.
- Your program can request execution of a CLEX command, optionally in a new CLEX instance, using calls in the `Command_Execution` package.
- Menus are another method for the user to control a program. See Chapter V-5 for more information about menus.

V-1.1.4 Environment Service

The environment service manages BiiN™ CL (Command Language) variables in active memory. A BiiN™ CL variable can contain a value of any CL type. Variables can be shared between jobs and processes in a session.

Figure V-1-4 shows how variables in passive store and active memory are related, and the order of evaluation for variables.

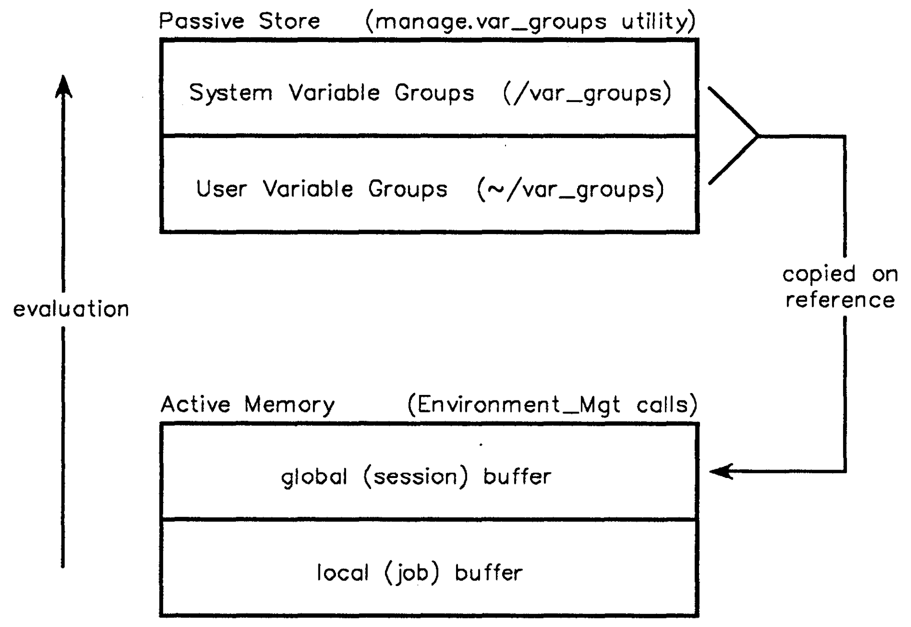

Passive Store (manage.var_groups utility)
System Variable Groups (/var_groups)
User Variable Groups (~/var_groups)
evaluation
copied on reference
Active Memory (Environment_Mgt calls)
global (session) buffer
local (job) buffer

Figure V-1-4. Command Language Variables

V-1.1.4.1 Environment Variable Concepts

- *A command language variable* - has a name, a type, a mode, and a value:

name	A CL string of letters and digits.
type	One of the six `CL_Defs` types: `boolean`, `integer`, `range`, `string`, `string list`, or `pointer`.
mode	Either `read_only`, indicating that the variable can be read but not assigned, or `read_write`, indicating that the variable can be read or assigned a value.
value	Any value of the appropriate type.

- *There are two kinds of variables* - those dynamically created in active memory, using calls in `Environment_Mgt` (and the `.variable` commands built into the command service), and those defined in passive store, using the `manage.variable_group` utility.

- *Variables in active memory* - are dynamically created in one of two buffers: either a job-specific local buffer or the session-specific global buffer.

  Variables in passive store - are in *system* and *user* groups. System variable groups are stored in the `/var_groups` directory and can only be updated by the system administrator. User variable groups are stored in each user's `~/var_groups`, and are maintained with the `manage.variable_group` utility.

-

- *Sets of system or user variables* - may be collected together in a group by giving them a group name. For example, `cli.prompt` is the `prompt` variable in the `cli.` group. Group variables may only be created with the `manage.variable_group` utility.
- *System variable groups* apply to all sessions on this node. System variables are created and maintained by the system administrator. System variables are in pre-defined groups, stored in the system directory `/var_groups`.
- *User variable groups* contain user-specific information, values, and defaults. User variables are created and maintained with the `manage.variable_group` utility. User variables are stored in a "var_groups" directory in your home directory (`~/var_groups`).
- *Job variables* are created and used by jobs, in the global buffer. Job variables may be created and used either by calls to `Environment_Mgt`, or by programs using the `Command_Handler` package (programs with runtime commands). Subsequent jobs in this session inherit all existing job variables.
- *Local variables* are used like job variables, but only exist for the duration of a job. Local variables are stored in the local buffer.

• *Variable names are evaluated upwards until found* -

1. local buffer
2. global buffer
3. user variable groups (`~/var_groups` directory)
4. system variable groups (`/var_groups` directory).

• *Each job has its own copy* - of system, user, and job variables (global buffer), and local variables (local buffer). Referencing a user or system variable causes a copy of that variable to be created in the job's global variable buffer. Job variables may be created in either the global or local buffer. Local variables are created, set, and removed in the job's local variable buffer.

• *Subsequent jobs are affected by changes to the global buffer* - since they inherit a copy of the current global buffer.

• *Changes only affect the job's copy of the variables* - changes to stored (system or user) variables are only made in the job's global variable buffer. Use the `manage.variable_group` utility to change system and user variables permanently.

• *Variables may also be created and changed* - using four built-in commands common to CLEX and the command service: ,

`create.variable`	Creates a new local or global variable, optionally with an initial value.
`set.variable`	Sets a value into an active variable which has mode `CL_Defs.read_write`.
`remove.variable`	Removes a variable from the local or global buffer. The version of the variable in passive store is not affected.
`list.variable`	Lists the type, mode, name, and current value of the specified variables or variable groups.

V-1.1.4.2 Environment Variable Summary

- A variable has a name, a type, a mode and a value. The variable's type is one of the six CL types: `boolean`, `string`, ... The variable's mode is either `read_only` or `read_write`. The variable's value is of the appropriate type.
- Command language (CL) variables control aspects of the current CLEX instance (such as message type and language) and contain information for use by jobs and programs (such as the current directory).
- System and user environment variables in passive store are maintained by the `manage.variable_group` utility.
- Variables in active memory are in either the local or global buffer. Stored variables are copied into the global buffer when referenced.
- Global variables are inherited by subsequent jobs and processes in this session.
- Variables may be read, set, and changed procedurally and interactively. Either call the environment service (`Environment_Mgt`), or enter one of the built-in `.variable` commands: `create.variable`, `set.variable`, `list.variable`, and `remove.variable`.

V-1.1.5 Menu Service

Menus are available to a BiiN™ program. They are defined by the `Window_Services` package.

Menu definitions may be created procedurally with calls to `Data_Definition_Mgt`.

Figure V-1-5 shows a *menu group* in a window, with one of the menus currently selected. This figure also shows the relationship between the window service (which provides the menus and the window), the display access method (which returns the user's selection), and your BiiN™ program.

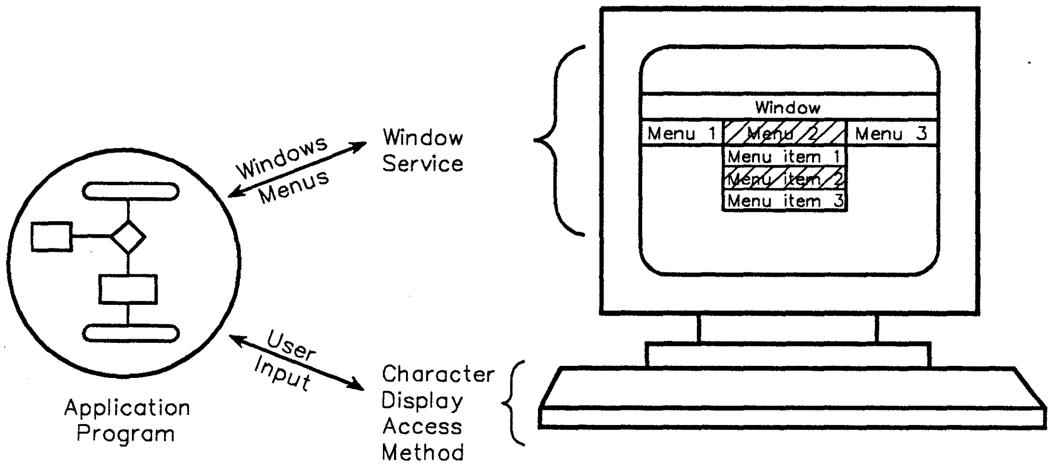


Figure V-1-5. BiiN™ Application Program and Menus

V-1.1.5.1 Menu Concepts

- *A menu has a title and one or more menu items* - the user selects a menu title, causing that menu to appear, and then selects one item from the menu.
- *Each menu is part of a menu group* - a menu group contains one or more menus.
- *A window can have only one menu group enabled* - several menu groups can be *installed* in one window, but only one menu group is *enabled* at any time.
- *Menu items may have associated "help" messages* - the user can request an explanation of any menu item.
- *Menu items may be picked by the cursor or by index* - to choose an item, the user enters the displayed item's index number.
- *Menu selection events* - can be read by calling the character display access method (`Character_Display_AM`).

V-1.1.5.2 Menu Summary

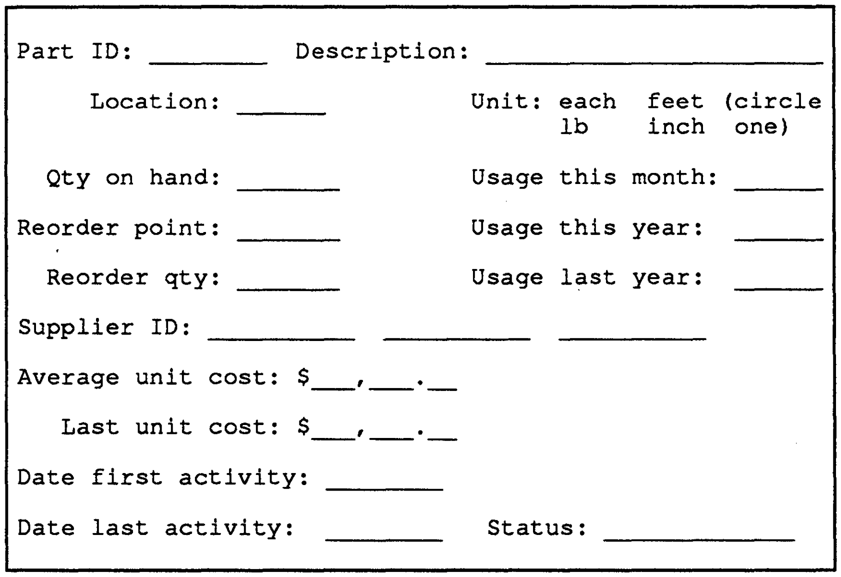
- Menus provide a consistent, easy-to-use user interface.
- A menu group contains one or more menus. Menus consist of a menu title and one or more menu items.
- A window can have several menu groups installed, but only one menu group at a time can be enabled.
- After a menu group is installed and enabled, menu selections and menu "help" messages are displayed without program intervention.

V-1.1.6 Form Service

The form service creates, modifies and executes interactive forms for structured data entry and display.

A form displayed in a window resembles a form printed on paper. Unlike paper forms, the forms created and controlled by using the form service can be dynamically changed for various data input and display requirements.

An example form is shown in Figure V-1-6.



```
Part ID: ________   Description: _______________________

     Location: ______          Unit: each   feet  (circle
                                     lb     inch   one)

  Qty on hand: _______         Usage this month: ______

Reorder point: _______         Usage this year:  ______

  Reorder qty: _______         Usage last year:  ______

Supplier ID: __________  __________  __________

Average unit cost: $___,___.__

   Last unit cost: $___,___.__

Date first activity: ________

Date last activity:  ________   Status: _____________
```

Figure V-1-6. Example Form

The cursor, which marks the current position in the form, may be moved back a space to erase an incorrect character, or back to the previous field to reenter a value. The contents of part of the form can be altered depending on the value of a previously entered field. Intermediate values can be calculated and stored transparently until needed later by the form. Even the order of execution of the form can be altered dynamically depending on the data entered.

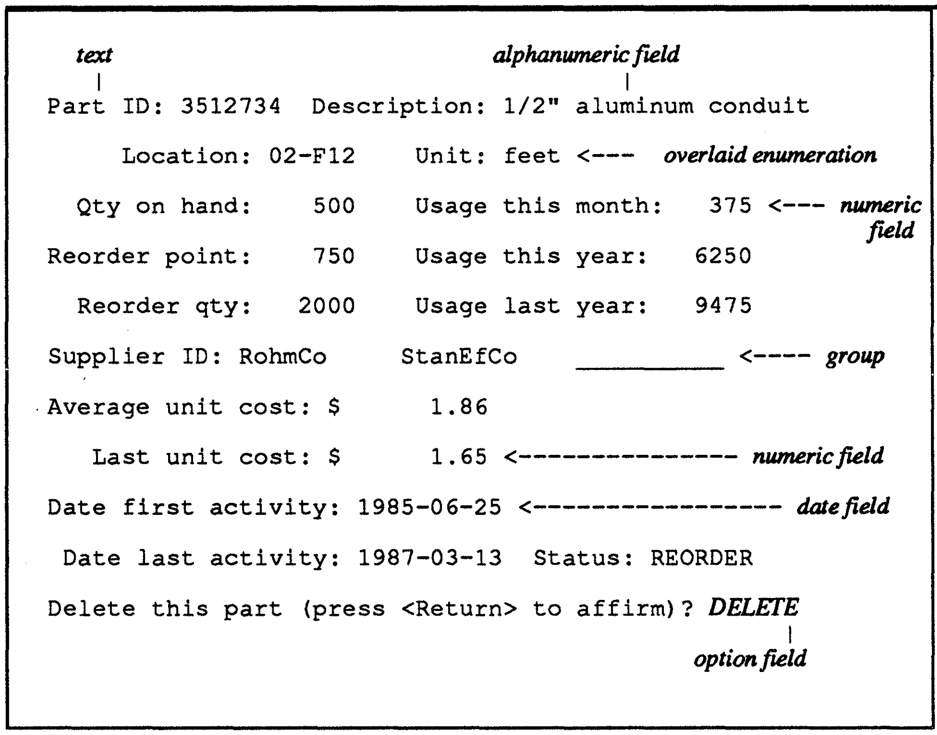
Creating an executable form involves:

- Designing the form's data and layout requirements
- Generating a form definition with `edit.form`or `define.standard_form`
- Creating a file of associated "help" messages
- Writing a program to execute a form, which can include processing routines (called before or after a field), key catchers (called by individual key sequences), and key lists (for the key catchers)
- Testing the form with the application.

The form service (`Form_Handler` package) provides calls to:

- Open and close a form
- Execute a form
- Modify data and control the execution network path
- Manage form elements
- Query the state of the form, a form element, or the last user interaction.

The Chapter V-5.3 describes the various parts of an interactive form and how they are combined into a single, executable form.



```
Part ID: 3512734  Description: 1/2" aluminum conduit

     Location: 02-F12    Unit: feet <---

  Qty on hand:     500    Usage this month:   375 <---

Reorder point:     750    Usage this year:   6250

  Reorder qty:    2000    Usage last year:   9475

Supplier ID: RohmCo     StanEfCo         __________ <----

Average unit cost: $      1.86

   Last unit cost: $      1.65 <---------------

Date first activity: 1985-06-25 <-----------------

 Date last activity: 1987-03-13  Status: REORDER

Delete this part (press <Return> to affirm)? DELETE
```

Figure V-1-7. Annotated Executable Form

Form definitions are created with the `define_standard_form` and `edit.form` utilities. A form definition may be derived from a file's record definition.

`define.standard_form`	Creates the simplest, default form definition, given a data record definition.
`edit.form`	Creates and modifies form definitions. Enables a programmer to design a form directly on the terminal screen and to define the properties for each form element as it is drawn and positioned. Detailed information for using `edit.form` is given in the *BiiN™ Systems Form Editor Guide.*

Application developers will normally use `edit.form` to create and update a form's definition. Form definitions can also be created procedurally using `Data_Definition_Mgt`.

V-1.1.6.1 Form Summary

- The form service builds upon the concept of a paper form to provide interactive forms capabilities on a terminal.
- A form can be created with the `edit.form` or `define.standard_form`.
- A form may consist of the following form elements:
  - texts
  - screen fields
  - enumerations
  - subforms: simple subforms and group subforms

- piles
- subprogram interfaces
- processing routines
- key catchers
- key lists.

- Variable length alphanumeric screen fields, and the screen elements containing them, can expand to accommodate data entered into the field.

V-1.1.7 Report Service

The report service provides formatted output from sets of records. A report is a printed or displayed document containing labelled data, often presented in hierarchical groups with subtotals and totals. A typical report is shown in Figure V-1-8.

```
Part ID    Description                Location    Unit

1234567    wiring harness              13-B27     each
3512734    1/2" aluminum conduit       02-F12     feet
4766117    5/16" hex carriage bolt     07-A02     each
7689482    flexible control cable      06-C13     inch
```

Figure V-1-8. Example Report

V-1.1.7.1 Report Concepts

The report service requires a record definition that describes the data to be printed. The report definition can be created from an existing file's record definition.

The format of a standard report page is shown in Figure V-1-9.

```
|----------------------------------------|-
| System date |              | Page Number |  |
|----------------------------------------|  |
|              Page Heading              |  | heading
|----------------------------------------|  |
| field 1 | ... | field i | ... | field n  |  |
|----------------------------------------|_
|                    .                   | |
|                    .                   | |
|                    .                   | |
|________________________________________| | page
|          |    |       |    |   |       | | body
|----------------------------------------| | area
|                                        | |
|                                        | |
|                   ...                  | |
|                                        | |
|                                        | |
|----------------------------------------|-
|              Page Footing              | | footing
|----------------------------------------|-
```

Figure V-1-9. Layout of a Standard Report Page

There are two methods for creating and modifying report definitions: by using interactive utilities (`define.standard_report` and `edit.report`) and procedurally, using `Data_Definition_Mgt`.

`define.standard_report`
: Creates the simplest, default report definition, given a data record definition.

`edit.report`
: Creates and modifies report definitions. See the *BiiN™ Systems Reports Guide* for detailed information on `edit.report`.

Application developers will normally use `edit.report` to create and update a report's definition. Report definitions can also be created procedurally using `Data_Definition_Mgt`.

Reports are composed of the following components:

Report Details A report detail is the smallest printable piece of a report. There are three kinds of details:

- Data detail
- Computed detail
- Text detail.

Report Parts A report consists of the following logical report parts, each consisting of one or more report details:

- Record print layout
- Report heading
- Page heading
- Page footing

- Control group heading (only if a control group hierarchy is defined)
- Control group footing (only if a control group hierarchy is defined).

Printing a Report

Reports are either printed or displayed depending on the type of output device, that is, depending on whether the physical output device is a printer, a terminal, or some other output device which supports character display I/O.

Two methods are available for printing (or displaying) a report: procedurally, by calling the `Report_Handler.Print` procedure, and interactively, by using the `print.file` utility.

Both of these methods read data records from the report input device, format each line of the report, and write the result to the given report output device.

V-1.1.7.2 Report Summary

- A report is produced from sets of input data records, formatted as desired, often presented in hierarchical groups with subtotals and totals.
- There are two utilities for creating and modifying report definitions: `edit.report` and `define.standard_report`.
- The report service (`Report_Handler`) provides calls to associate a report with an input and output device, print a report, and control error handling.
- A report can be printed or displayed either procedurally, with the `Report_Handler.Print` procedure, or interactively, using the `print.file` utility.

V-1.2 Summary

- Human Interface Services is composed of five services:

command service	Given a command definition, parses and returns invocation and runtime commands.
environment service	Reads, writes, and creates BiiN™ CL variables.
menu service	Given a menu definition, provides on-screen menus (lists of menu items).
form service	Given a form definition, controls execution of a data entry or display form.
report service	Given a report definition, produces a printed report from data records.

- These services perform most of the processing necessary for user interaction, including enforcement of input requirements and automatic display of "help" messages.
- A default data entry form, and a default report, can be automatically created from a file's record definition.
- Human Interface Services provide a complete, consistent user interface, shared by BiiN™ system utilities and CLEX itself.

CREATING A BiiN™ APPLICATION PROGRAM 2

Contents

This chapter describes how to use the various parts of the BiiN™ system software to create a simple application program. The example is an inventory program controlled with menus, which gets structured input from a user, updates the inventory files, displays formatted information, and prints an inventory report.

The complete source code for this example program is listed in Appendix A.

Packages Used:

`Access_Mgt`	Interface for checking or changing rights in access descriptors.
`Character_Display_AM`	Provides device-independent I/O to character display devices such as printers, plotters, and windows on character and graphics terminals.
`Data_Definition_Mgt`	Manages data definitions (DDefs). This interface is a symbol table for the development of a DDef compiler.
`Directory_Mgt`	Manages directories and directory entries.
`File_Admin`	Administers files.
`Form_Handler`	Provides calls to process, control, and change forms.
`Message_Services`	Provides calls to write messages from message files, message stacks, or message blocks.
`Process_Mgt`	Provides public operations on processes.
`Record_AM`	Provides device-independent record I/O.
`Report_Handler`	Provides calls for initializing and printing a report.
`Window_Services`	Provides windows on character and graphics terminals, including pull-down menus.

Figure V-2-1 shows the external relationships of a BiiN™ program to its terminal (windows), its control input (menus), its notes, warnings, and help information (messages), its data input and display (forms), and its printed output (reports).

As with most computer systems, the BiiN™ OS (CLEX) invokes the program and passes in invocation command arguments. And, as usual, the program interacts with files.

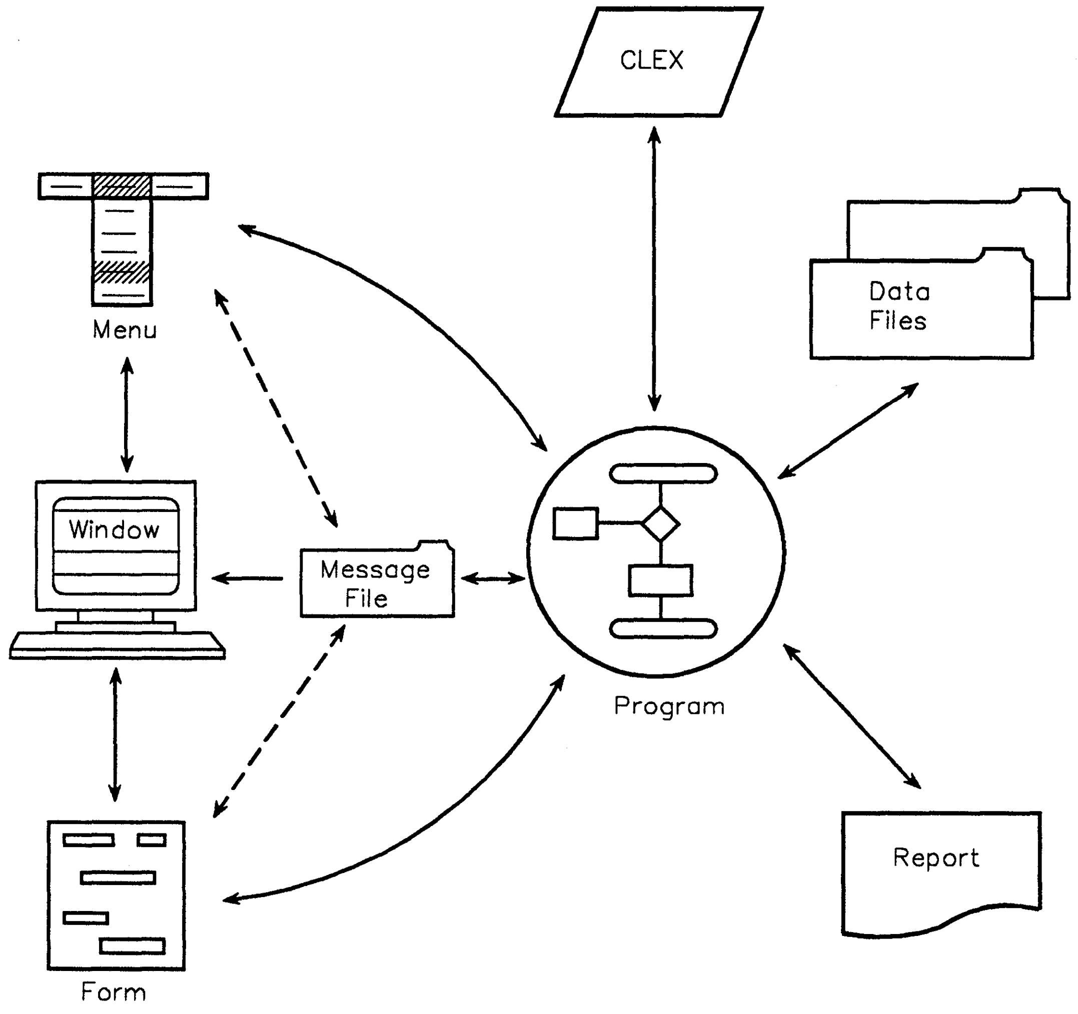


Figure V-2-1. Typical BiiN™ Application Program

A BiiN™ program relies on various system services. Each of these services are described in other chapters in this *BiiN™/OS Guide.*

BiiN™ provides several ways of interacting with users via terminals:

- Windows, to reserve an area of the terminal screen for a program. A program may open any number of windows. Usually, there is one main window for user input and data display, and optionally a small window for help and error messages. The message window may be the already existing system message window.
- Messages, to display a text with up to five parameters (such as file name and error number) in a user-selected language and format (short, long, help).
- Menus, to allow a user to select an item from a list. The menu group ID, menu ID, and menu item ID numbers are returned to the program.
- Commands, to allow the user to control a program. See Chapter V-3, "Building New Commands", for more information.

- Forms, to enforce structured data entry. Each data entry field may have an associated help message. Forms are also used to display structured data.
- Reports, to produce a formatted display of a set of data records.

A BiiN™ program itself is insulated from changes to:

- Message texts and variants - only the message file itself needs to be updated. Message variants are automatically displayed in the user's desired language and format.
- Changing languages for menus or commands - if the numeric menu and command IDs remain the same.
- Form layouts - the displayed formats may be changed; as long as no fields are added or deleted, the program will not know the difference.
- Report layouts - the report formatting, headings, control breaks, footings, and so forth, can be changed.

The following tasks are described in this chapter:

- How to create and read the program's invocation command.
- How to create and display messages.
- How to use windows, menus, and forms to interact with the user.
- How to read, write, and update records in named files.
- How to print a report from a file and how to sort records from a file and then print a report from the sorted records.

V-2.1 Concepts

V-2.1.1 Designing a BiiN™ Program

As with any program on any system, a complete program description is the first step when creating an application. For a BiiN™ program, this program description includes:

- Whether it's to be a batch or interactive program,
- Whether to use a menu-driven or command-driven user control interface,

menu selections	Simplest for the user - define menu groups, individual menus and menu items
runtime commands	Provides more control - define sets of commands, command names, types and numbers of command arguments.

- What input and output files are required, including the record layout of the program's files,
- Design of data entry and display forms, if used,
- What data manipulation is desired, and
- What types of reports are to be produced.

Once the main files' record layouts have been specified, the specification of default data input forms and reports can be done automatically (see the next section, "Defining the Application's Data Structure").

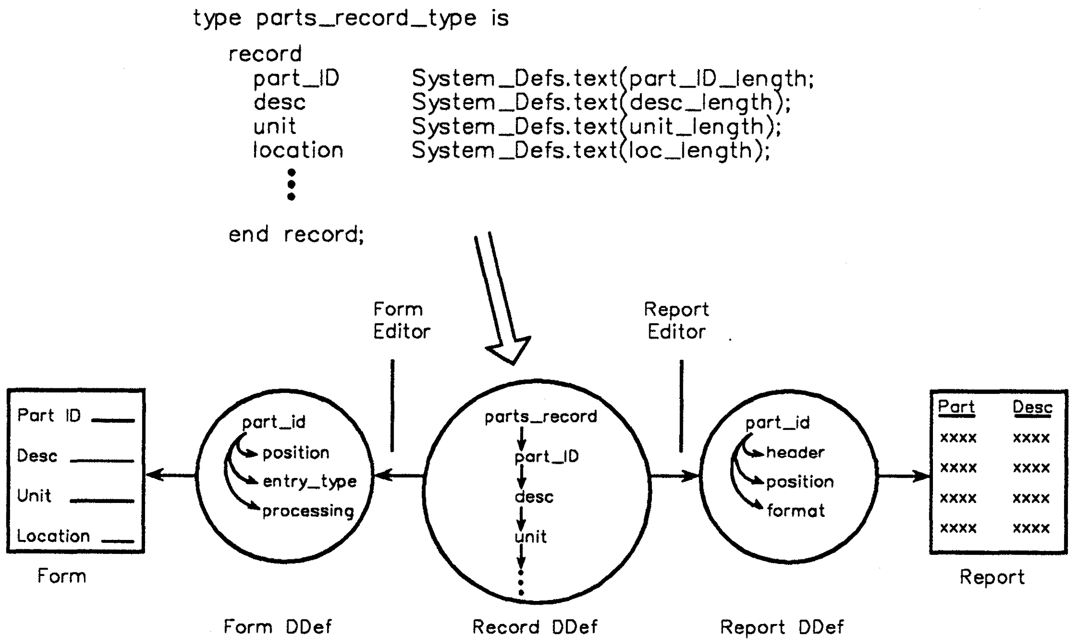
V-2.1.2 Defining the Application's Data Structure

The data definition (record layout) of a file can be used to automatically generate input and display forms and report definitions (DDefs). The resulting form and report DDefs can then be edited with the corresponding editor utilities to create a desired format.

Tools Used:

TBD	Creates a *record DDef* for a file.
`define.standard_form`	Creates a standard form DDef from a record DDef.
`edit.form`	Edits a form description's layout and screen properties.
`define.standard_report`	Creates a standard report DDef from a record DDef.
`edit.report`	Edits a report description.

Figure V-2-2 shows the relationship between a file's record DDef, the standard form layout, and the standard report layout.



```
type parts_record_type is
   record
     part_ID      System_Defs.text(part_ID_length;
     desc         System_Defs.text(desc_length);
     unit         System_Defs.text(unit_length);
     location     System_Defs.text(loc_length);
       :
   end record;
```

Figure V-2-2. File Data Definition and Associated Forms and Reports

V-2.1.3 Example Program Overview

The example program is a menu-driven inventory control program. The menu group's title line is used to select one of six menus. Using selections from the menus, the user may inquire about parts on file, enter a new part ID and description, change part information, print one of two inventory reports, and re-index the parts file.

The program has an associated message file which contains the texts of all information, warning, and error messages. Help texts are stored within the menu and form definitions.

Menus

The menus in the example program are shown in Figure V-2-3.

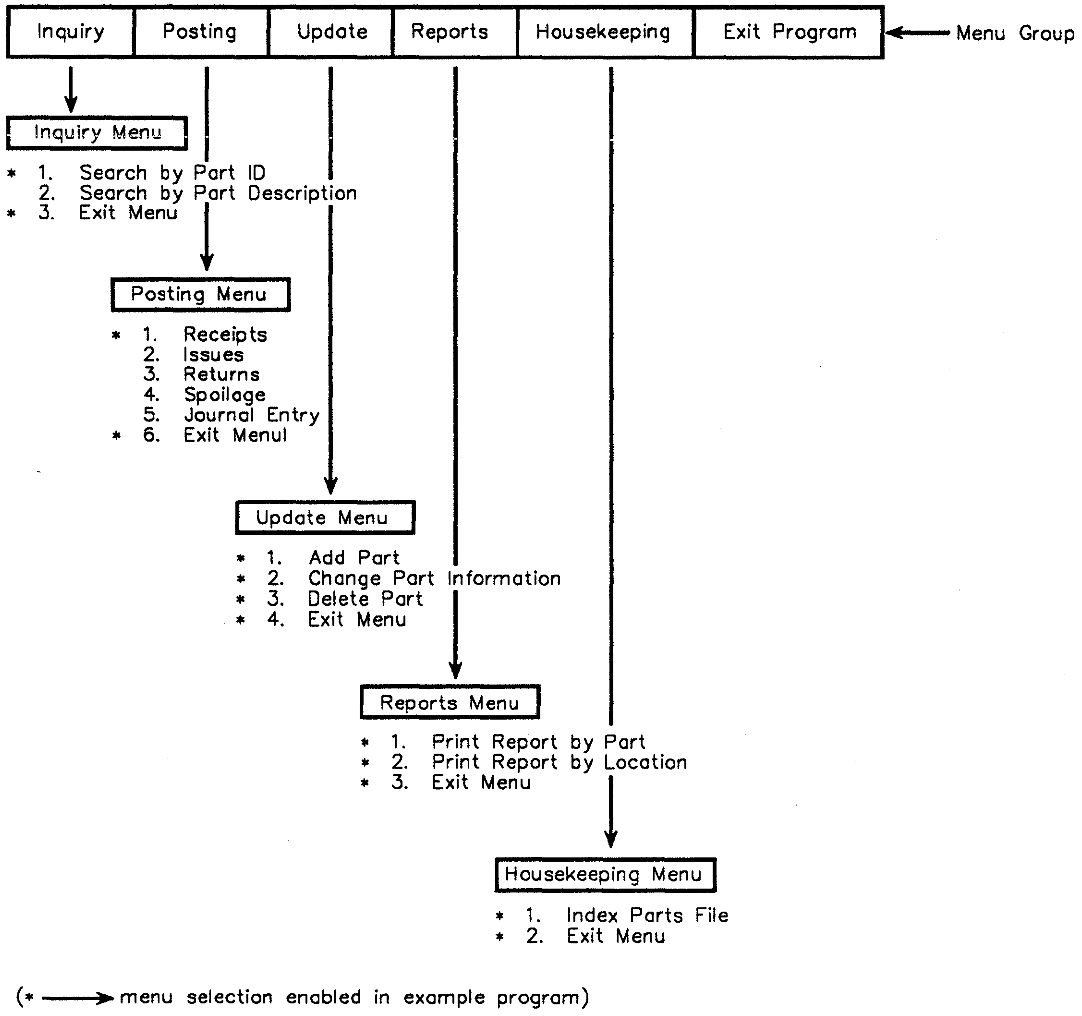


Figure V-2-3. Example Program Menus

Data Files

There are two data files used by this program: the inventory parts file, indexed by part ID, and the log file, where records of updates and changes to the parts file are written.

The parts file record layout, in Ada, is:

```
(from Inventory_Files specification)

subtype part_ID_type is System_Defs.text(
    part_ID_length);

subtype supplier_ID_type is System_Defs.text(
    supplier_ID_length);

subtype location_type is System_Defs.text(
    loc_length);
. . .
subtype qty_type is System.ordinal
          range 0..9_999_999;
. . .
subtype cost_type is float
    range 0.0..99_999_999.99;

type supplier_array_type is
  array (1..max_suppliers) of supplier_ID_type;
. . .
type parts_record_type is
. . .
  record
    part_ID:               part_ID_type;
. . .
    desc:                  System_Defs.text(
        desc_length);
. . .
    unit:                  System_Defs.text(
        unit_length);
. . .
    location:              location_type;
. . .
    qty_on_hand:           qty_type;
    reorder_point:         qty_type;
    reorder_qty:           qty_type;
    suppliers:             supplier_array_type;
. . .
    usage_this_month:      qty_type;
    usage_last_month:      qty_type;
    usage_last_year:       qty_type;
    avg_unit_cost:         cost_type;
    last_unit_cost:        cost_type;
    date_first_act:
        Timing_Conversions.numeric_time;
. . .
    date_last_act:
        Timing_Conversions.numeric_time;
. . .
    status:                System_Defs.text(
        status_length);
. . .
end record;
```

The log file record layout, in Ada, is:

```
(from Inventory_Files specification)

type action_type is (
    create,
. . .
    update,
. . .
    delete,
. . .
    receipt,
    issue,
    returns,
    spoilage,
    journal);


type log_record_type is
. . .
  record
    part_ID:      part_ID_type;
. . .
    action:       action_type;
. . .
    time:
        Timing_Conversions.numeric_time;
. . .
    doc_number:   System_Defs.text(
        doc_length);
. . .
    qty:          qty_type;
. . .
    job_ID:       System_Defs.text(
        job_length);
. . .
    supplier_ID:  supplier_ID_type;
. . .
 end record;
```

Program Source Code Files

A typical menu-driven application program has the following outline:

Setup	Open a window for this program, open the necessary files, display the main menu group.
Input	Get a menu selection from user, get necessary data for operation (input directly from user, from a form, or from a file).
Processing	Perform the selected operation.
Output	Display information in program window, update files, print a report.
Termination	Close files, deallocate objects, close the program's window.

The example program is organized into the one main procedure and six Ada packages. Each of the packages collects the procedures related to a given service area: windows, files, menus, forms, and reports. The relationship between these parts of the example program are shown in Figure V-2-4.

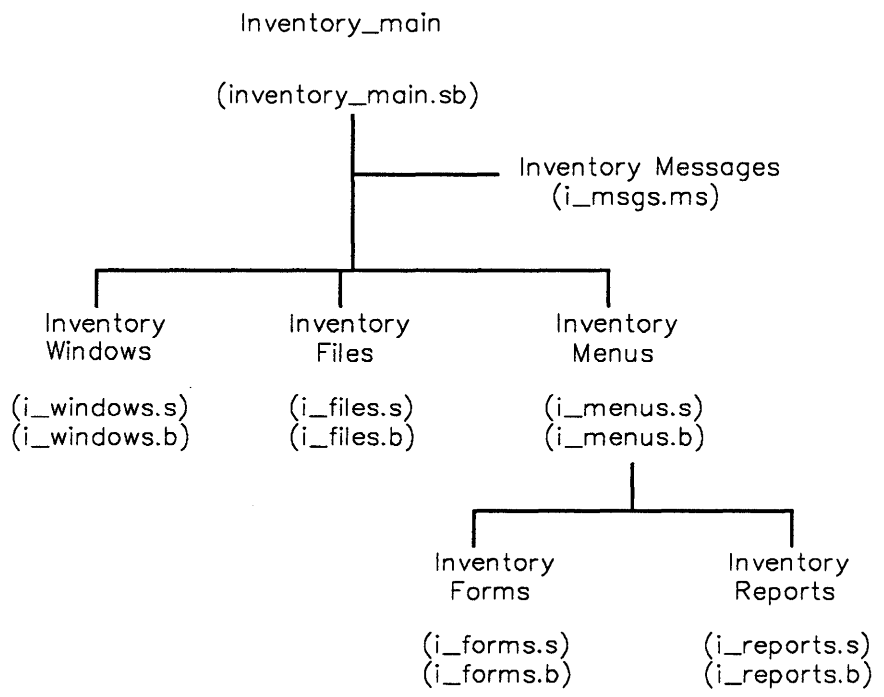


Figure V-2-4. Example Program Source Files

Setup

The following program fragment shows how the example program sets up its windows, opens its files, and prepares for menu input.

```
(from Inventory_Main)

procedure Inventory_main
. . .
is
. . .
  begin
. . .
     Inventory_Windows.Open_program_windows;
. . .
     Inventory_Files.Open_parts_file;
. . .
     Inventory_Menus.Set_up_menu_group;
. . .
     Character_Display_AM.Ops.Set_input_type_mask(
         opened_dev => Inventory_Windows.main_window,
         new_mask   => Terminal_Defs.input_type_mask'(
             Terminal_Defs.menu_item_picked => true,
             others                         => false));
```

Processing

The following program fragment shows the example program's main processing loop: read the menu selection, then perform the appropriate action (possibly just *exit program*).

```
(from Inventory_Main)

    loop
. . .
      Character_Display_AM.Ops.Read(
          opened_dev => Inventory_Windows.main_window,
. . .
      case menu_select.menu is

        when Inventory_Menus.inquiry_menu_ID =>
            Inventory_Menus.Process_inquiry_menu(
                selection  => menu_select.item);
. . .
        when Inventory_Menus.posting_menu_ID =>
. . .
        when Inventory_Menus.exit_menu_ID =>
            EXIT;
. . .
      end case;          -- "case menu_select.menu is"

    end loop;
```

Termination

The following program fragment shows how the example program closes both its windows (which disables enabled menus) and its files.

```
(from Inventory_Main)

    Inventory_Files.Close_parts_file;
. . .
    Inventory_Windows.Close_program_windows;

  end Inventory_main;
```

V-2.2 Techniques

After reading this section, you will be able to:

- Interact with the command line:
  - Create and store an invocation command definition and argument definitions
  - Read command line argument values.
- Set up a window for a program
- Accept and process user menu selections from menus
- Display a message
- Set up a form and get the user's input data
- Display a form containing program-generated information.
- Update a given record from a file
- Print a report from a file.

V-2.2.1 Creating and Processing the Invocation Command

Calls Used:

`manage.program`
: Creates command definitions.

`Command_Handler.Open_invocation_command_processing`
: Opens the invocation command device for processing.

`Command_Handler.Get_`*argument_type*
: Returns argument values.

`Command_Handler.Close`
: Closes the opened invocation command input device.

Define your program's invocation command name and the necessary arguments. Use the `manage.commands` command set in the `manage.program` utility to define the program's invocation command.

Call `Command_Handler.Open_invocation_command_processing` to open the invocation command input device. The invocation command has already been parsed, and calls to `Command_Handler.Get_`*argument_type* will return the invocation argument values.

When you are through reading argument values, close the opened invocation command input device by calling `Command_Handler.Close`.

See Chapter V-3 for more information on processing commands.

V-2.2.2 Using Windows in a Program

Calls Used:

`Character_Display_AM.Get_device_object`
: Returns the object underlying a device.

`Process_Mgt.Get_process_globals_entry`
: Returns process global variables; in this case, we are looking for the standard input device.

`Window_Services.Ops.Create_window`
: Creates a new window with a given size and position.

`Window_Services.Ops.Get_terminal`
: Returns an AD to the terminal on which an existing window is installed.

The following program fragment shows how the example program sets up its main window. The program assumes that the standard input, on entry, is from a window.

First, you need to find that opened window, then the actual window, then the terminal displaying the actual window. Then you can set up the new program window on that terminal.

```
(from Inventory_Windows specification)

  main_window_size:  Terminal_Defs.point_info := (
      80,20);
. . .
  main_buffer_size:  Terminal_Defs.point_info := (
      80,20);
. . .
  main_window_pos:  Terminal_Defs.point_info := (
      1,1);

(from Inventory_Windows body)

procedure Open_program_windows
. . .
is
      old_opened_window:    Device_Defs.opened_device;
      old_window:           Device_Defs.device;
      underlying_terminal:  Device_Defs.device;

begin
. . .
  old_opened_window :=
      Process_Mgt.Get_process_globals_entry(
          Process_Mgt_Types.standard_input);
. . .
  old_window := Byte_Stream_AM.Ops.Get_device_object(
      old_opened_window);
. . .
  underlying_terminal :=
      Window_Services.Ops.Get_terminal(
          old_window);
. . .
  main_window := Window_Services.Ops.Create_window(
      terminal            => underlying_terminal,
      pixel_units         => false,
        -- characters, not pixels
      fb_size             => main_buffer_size,
      desired_window_size => main_window_size,
      window_pos          => main_window_pos,
      view_pos            =>
          Terminal_Defs.point_info'(1,1));
. . .
end Open_program_windows;
```

V-2.2.3 Processing a Menu Selection

Calls Used:

`Character_Display_AM.Ops.Set_input_type_mask`
Sets the allowable input events for a window.

`Character_Display_AM.Read`
Reads an event from a terminal.

`Data_Definition_Mgt.Retrieve_DDef`
Retrieves an object's DDef, given the object's AD.

`Directory_Mgt.Retrieve`
Retrieves a stored object's AD, given the object's pathname.

`Window_Services.Ops.Install_menu_group`
Installs a menu group in a window.

`Window_Services.Ops.Menu_group_enable`
Enables an installed menu group.

This technique shows how to:

- create a menu (using a utility)
- install and enable a menu
- read the user's menu selection
- process a menu's selections.

Creating Menus

Define the menu titles and menu item texts, then the menu groups.

Create each menu group, using the **TBD** (`edit.menu`) utility.

1. Create a null menu group to contain the menus.
2. Add each menu and its title to the menu group.
3. Add each menu item and its text to each menu.

Store the menu group DDef under a pathname.

Installing and Enabling a Menu Group

Retrieve the menu group's DDef.

Install and enable the menu group, using calls in `Window_Services`:

1. `Install_menu_group` - install the menu group in an open window.
2. `Menu_group_enable` - enable a menu group for user selection.

```
(from Inventory_Menus specification)

        inv_menu_group_ID:  constant
            Terminal_Defs.menu_group_ID := 1;

(from Inventory_Menus body)

        menu_group_DDef_AD:  Data_Definition_Mgt.DDef_AD;
  . . .
        menu_group_node:
            Data_Definition_Mgt.node_reference;
  . . .
          menu_group_DDef_AD := DDef_from_untyped(
              Directory_Mgt.Retrieve(
                  name => menu_group_DDef_path));
  . . .
          menu_group_node := Data_Definition_Mgt.
              Retrieve_DDef(
                  DDef => menu_group_DDef_AD,
                  name => menu_group_DDef_root_name);
  . . .
              Window_Services.Ops.Install_menu_group(
                window     => Inventory_Windows.
                              main_window,
                menu_group => menu_group_node,
                ID         => inv_menu_group_ID);
  . . .
              Window_Services.Ops.Menu_group_enable(
                window     => Inventory_Windows.
                              main_window,
                menu_group => inv_menu_group_ID,
                enable     => true);
```

Reading a Menu Selection

To read a menu selection:

- Set the input type mask for the menu group's window to include `Terminal_Defs.menu_item_picked`,
- Wait for an input event at the terminal, and
- Read the menu selection values (menu group ID, menu ID, and menu item ID numbers).

```
(from Inventory_Main)

   menu_select:  Terminal_Defs.menu_selection;
. . .
   event_type:  Terminal_Defs.input_enum;
. . .
   event_num:   System.ordinal;
. . .
   Character_Display_AM.Ops.Set_input_type_mask(
       opened_dev => Inventory_Windows.main_window,
       new_mask   => Terminal_Defs.input_type_mask'(
           Terminal_Defs.menu_item_picked => true,
           others                         => false));
. . .
     Character_Display_AM.Ops.Read(
         opened_dev => Inventory_Windows.main_window,
         buffer_VA  => menu_select'address,
         max_events => 1,
         max_bytes  => 0,
         block      => true,   -- Wait . . .
         type_read  => event_type,
         num_read   => event_num);
. . .
     case menu_select.menu is

       when Inventory_Menus.inquiry_menu_ID =>
           Inventory_Menus.Process_inquiry_menu(
               selection  => menu_select.item);
```

Example Menu Processing Routine

Use the menu and item selection numbers (for example, in a `case` statement) to determine the appropriate action.

Define the menu and item numbers:

```
(from Inventory_Menus specification)

   inquiry_menu_ID:       constant
       Terminal_Defs.menu_ID := 1;

   posting_menu_ID:       constant
       Terminal_Defs.menu_ID := 2;

   update_menu_ID:        constant
       Terminal_Defs.menu_ID := 3;

   report_menu_ID:        constant
       Terminal_Defs.menu_ID := 4;

   housekeeping_menu_ID:  constant
       Terminal_Defs.menu_ID := 5;

   exit_menu_ID:          constant
       Terminal_Defs.menu_ID := 6;

 -- Inquiry menu items
   inq_by_part_item:  constant
       Terminal_Defs.menu_item_ID := 1;
   inq_by_desc_item:  constant
       Terminal_Defs.menu_item_ID := 2;
   inq_exit_item:     constant
       Terminal_Defs.menu_item_ID := 3;
```

Process the menu's selections:

```
(from Inventory_Menus body)

   procedure Process_inquiry_menu(
       selection:  Terminal_Defs.menu_item_ID)
         -- Selection made in this menu.
   is
. . .
   begin

     case selection is

       when inq_by_part_item => Inventory_Forms.
           Process_inquiry_form;

       when inq_by_desc_item =>

         Message_Services.Write_msg(
             msg_id => no_selection_code,
. . .
       when inq_exit_item =>
         return;

       when others => null;

     end case;

   end Process_inquiry_menu;
```

V-2.2.4 Displaying a Message

Calls Used:

`Message_Services.Write_msg`
Writes a specified message and its parameters to an opened device.

Messages are used to display status information and for warnings and errors.

To use messages, you must create a message file containing the texts for all your program's messages. This is done with the `manage.messages` utility (or the `manage.messages` command set in the `manage.program` utility), using your message definition commands.

The program need only know the message file's pathname:

```
(from Inventory_Messages specification)

message_file:  constant System_Defs.text_AD :=
    new System_Defs.text'(
        31,31,"/example/inventory/message_file");
. . .
message_object:  constant System.untyped_word :=
    System.null_word;

  pragma bind (message_object,
               "inventory_messages.message_file");
```

A program refers to a message by an `Incident_Defs.incident_code`, which determines the message file, the message index numbers (module number and message number), and the severity level.

The message definition commands can be stored in your source file near the `Incident_Defs.incident_code` declarations. The `manage.program` utility can extract message definition commands from your source file, to create the message file.

To define a given message:

```
(from Inventory_Messages specification)

module:  constant := 4;
. . .
-- *M*   set.language  :language = English
-- *M*   create.variable  module  :value = 4
. . .
no_selection_code:  constant
    Incident_Defs.incident_code := (
         message_object =>
             Inventory_Messages.message_object,
         module             => module,
         number             => 1,
         severity           =>
             Incident_Defs.warning);

-- *M*   store  :module = $module  :number = 1\
-- *M*          :msg_name =  no_selection\
-- *M*          :short = "Selection $p1<selection
-- *M*                    number> is not implemented."
```

To write a message to the default message device (window):

```
(from Inventory_Menus body)

    selection:  Terminal_Defs.menu_item_ID)
. . .
      Message_Services.Write_msg(
          msg_id => no_selection_code,
          param1 =>
              Incident_Defs.message_parameter(
              typ => Incident_Defs.ord,
              len => 0)'(
                  typ   => Incident_Defs.ord,
                  len   => 0,
                  o_val => selection));
```

V-2.2.5 Getting Data from a Form

Calls Used:

`Directory_Mgt.Retrieve`	Retrieves a stored object's AD, given the object's pathname.
`Form_Handler.Clear`	Clears a form from its window.
`Form_Handler.Close_form`	Closes and deallocates an opened form.
`Form_Handler.Fetch_value`	Gets a value from a field in a form.
`Form_Handler.Get`	Displays a form for user input; returns when the form is finished.
`Form_Handler.Open_form`	Opens a form for processing.
`Record_AM.Open_by_name`	Opens a device for record access, given the device's pathname.
`Record_AM.Ops.Close`	Closes an open device.
`Record_AM.Ops.Update`	Updates (writes) a record to an open device.

Define the form layout, field names, and field types. Create the form DDef with the `define.standard_form` utility, and/or the `edit.form` utility. Store the form's DDef under a pathname.

Get the stored form DDef by calling `Directory_Mgt.Retrieve` with the form's pathname, then converting the returned untyped word to a DDef AD using an instance of `Unchecked_Conversion`. Open the form by calling `Form_Handler.Open_form`.

```
(from Inventory_Forms body)

      opened_form:  Form_Defs.opened_form_AD;
. . .
      opened_form := Form_Handler.Open_form(
          DDef => DDef_from_untyped(
                  Directory_Mgt.Retrieve(
                  name => form_pathname)));
```

If the form has groups or piles to be set, call `Form_Handler.Create_group_instances` as necessary.

```
(from Inventory_Forms body)

        when Inventory_Menus.update_add_item =>

          Form_Handler.Create_group_instances(
              opened_form_a        => opened_form,
              group                => update_add,
              number_of_instances  => 1);
```

If desired, open the form for record access:

```
(from Inventory_Forms body)

opened_record_form:  Device_Defs.opened_device;
. . .
opened_record_form := Record_AM.Open_by_name(
    name         => update_form_pathname,
    input_output => Device_Defs.inout);
```

Display the form and allow user data entry, by calling `Form_Handler.Get`, specifying the form and the window where the form is to be displayed:

```
(from Inventory_Forms body)

  form_status := Form_Handler.Get(
      opened_form_a   => opened_form,
      opened_window_a =>
          Inventory_Windows.main_window);

  if form_status /= Form_Defs.finished then
```

Fetch data from fields in the form by calling `Form_Handler.Fetch_value`:

```
(from Inventory_Forms body)

    Form_Handler.Fetch_value(
        opened_form_a        => opened_form,
        element              => part_ID_field,
        subunit              => System_Defs.null_text,
          -- added subunit; value correct?
        value_buffer_VA      => part_ID'address,
        value_length         => part_ID'size/8,
        value_t              =>
            Data_Definition_Mgt.t_string,
        element_value_length => length,
        empty                => empty);

    if empty then
```

Read a data record from the form by calling `Record_AM.Ops.Read`:

```
(from Inventory_Forms body)

parts_record:
    Inventory_Files.parts_record_type;
. . .
length:        System.ordinal;
. . .
            length := Record_AM.Ops.Read(
                opened_dev => opened_record_form,
                buffer_VA  => parts_record'address,
                length     => parts_record'size/8);
```

Clear the form from the window by calling `Form_Handler.Clear_form`. Close the form by calling `Form_Handler.Close_form`:

```
(from Inventory_Forms body)

Form_Handler.Clear(
    opened_form_a => opened_form);

Form_Handler.Close_form(
    opened_form_a => opened_form);
```

Close record access to the form, if opened:

```
(from Inventory_Forms body)

Record_AM.Ops.Close(
    opened_dev => opened_record_form);
```

V-2.2.6 Displaying Data Using a Form

Calls Used:

`Directory_Mgt.Retrieve`	Gets a stored object's AD, given its pathname.
`Form_Handler.Clear`	Clears a form from its window.
`Form_Handler.Close_form`	Closes and deallocates an opened form.
`Form_Handler.Open_form`	Opens a form for processing.
`Form_Handler.Put`	Displays a read-only form.
`Form_Handler.Store_value`	Assigns a value to a field in an opened form.
`Record_AM.Open_by_name`	Opens a device for record access, given the device's pathname.
`Record_AM.Ops.Close`	Closes an open device.
`Record_AM.Ops.Update`	Updates (writes) a record to an open device.

This section describes how to use a form to display structured information.

Define and create the form, retrieve and open the form's DDef, and set any necessary groups or piles, as described in the previous section, "Getting Data from a Form".

If the form has been opened for record access (see previous section), write a data record into the form by calling `Record_AM.Ops.Update`:

```
(from Inventory_Forms body)

    parts_record:
        Inventory_Files.parts_record_type;
. . .
            Record_AM.Ops.Update(
                opened_dev => opened_record_form,
                buffer_VA  => parts_record'address,
                length     => parts_record'size/8);
```

Store data directly into individual fields in the form by calling `Form_Handler.Store_value`:

(from Inventory_Forms body)

```
            Form_Handler.Store_value(
                opened_form_a   => opened_form,
                element         => desc_field,
                subunit         => System_Defs.null_text,
                  -- added subunit; value correct?
                value_buffer_VA =>
                    parts_record.desc'address,
                value_length    =>
                    parts_record.desc'size/8,
                value_t         =>
                    Data_Definition_Mgt.t_string);
```

Display the form and its contents by calling `Form_Handler.Put`, specifying the form and the window where the form is to be displayed.

Clear the form from the window by calling `Form_Handler.Clear_form`, then close the form by calling `Form_Handler.Close_form`.

V-2.2.7 Updating a File

Packages Used:

`Access_Mgt.Permits`
    Checks an object for given access rights.

`Directory_Mgt.Retrieve`
    Returns an AD to an object, given a pathname.

`Record_AM.Keyed_Ops.Read_by_key`
    Reads a record from an opened indexed device, given the index key value.

`Record_AM.Keyed_Ops.Update_by_key`
    Updates a record in an opened indexed device, given the index key value.

`Record_AM.Ops.Close`
    Closes an opened device.

`Record_AM.Ops.Insert`
    Inserts a record into an opened device.

`Record_AM.Ops.Open`
    Opens a given device for record input or output, given a device AD.

`Record_AM.Ops.Open_by_name`
    Opens a given device for record access, given a device's pathname.

This section briefly describes how to use the filing service to update records in a file. For more information and procedural examples, see Chapter IV-9, "Using Record I/O with Structured Files".

1. Define the file's record layout and then create the file's record DDef. This can be done procedurally using calls to `Data_Definition_Mgt`. Store the file and its record DDef under pathnames.
2. Open the file by calling `Record_AM.Ops.Open` or `Record_AM.Open_by_name`.
3. Read/write/rewrite records:

- If the file is indexed:

read a record	Get the index key value of the desired record, and call `Record_AM.Keyed_Ops.Read_by_key`.
write a record	Set the record value, then call `Record_AM.Ops.Insert`.
rewrite a record	Set the record value, then call `Record_AM.Keyed_Ops.Update_by_key`.

- If the file is not indexed:

read a record	Set the record pointer with `Record_AM.Ops.Set_position`, then read the record with `Record_AM.Ops.Read`.
write a record	Write the record with `Record_AM.Ops.Insert`.
rewrite a record	Rewrite the record with `Record_AM.Ops.Update`.

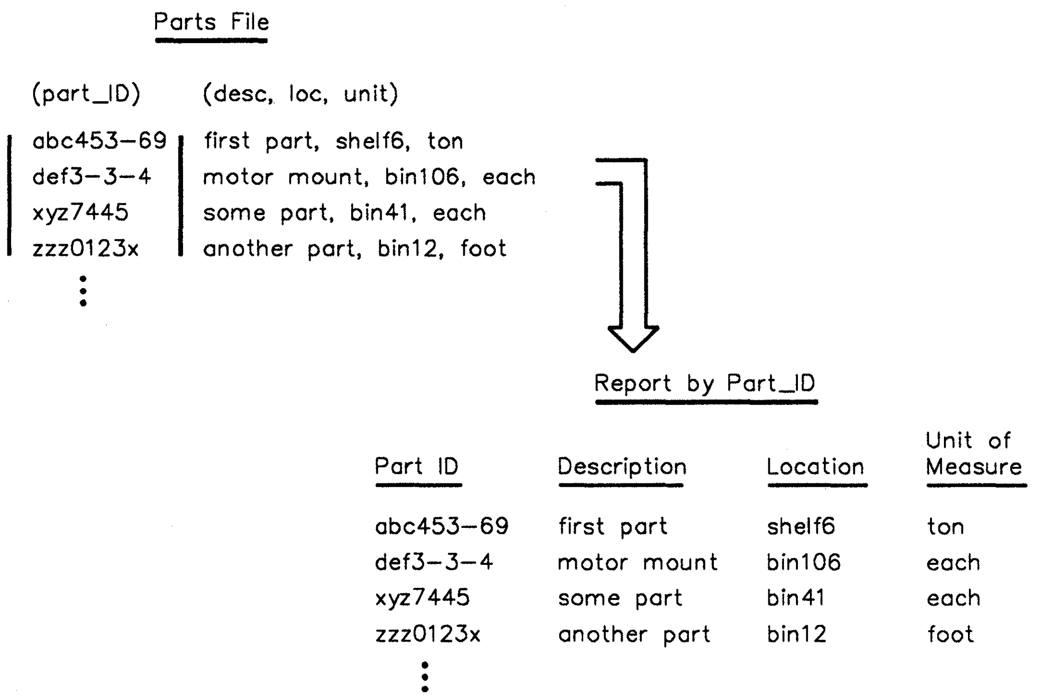
4. Close a file by calling `Record_AM.Ops.Close`.

V-2.2.8 Printing a Report from a File

Calls Used:

`Byte_Stream_AM.Open_by_name`	Opens a device for byte stream access, given the device's pathname.
`Directory_Mgt.Retrieve`	Retrieves an object, given the object's pathname.
`Record_AM.Open`	Opens a file or device for record access.
`Record_AM.Ops.Close`	Closes an opened file or device.
`Report_Handler.Initialize`	Initializes an opened report, given the report DDef and the input and output devices.
`Report_Handler.Print`	Prints an initialized report.

This technique shows how to produce a given report from a file to an output device.



Parts File

(part_ID)	(desc, loc, unit)
abc453–69	first part, shelf6, ton
def3–3–4	motor mount, bin106, each
xyz7445	some part, bin41, each
zzz0123x	another part, bin12, foot
⋮	

Report by Part_ID

Part ID	Description	Location	Unit of Measure
abc453–69	first part	shelf6	ton
def3–3–4	motor mount	bin106	each
xyz7445	some part	bin41	each
zzz0123x	another part	bin12	foot
⋮			

Figure V-2-5. File and an Associated Report

Create the report itself:

- Given the previously created file's record DDef, use the `define.standard_report` utility and/or the `edit.report` utility, to create the desired report format.
- Store the report DDef under a pathname.

Open the desired input file, by calling `Record_AM.Ops.Open`.

- To select a range of records in an indexed file for inclusion in the report, call `Record_AM.Keyed_Ops.Set_key_range` with the appropriate first and last key values.

(from Inventory_Reports body)

```
procedure Print_report_by_part(
    output_dev_pathname:  System_Defs.text)
. . .
is

opened_output:  Device_Defs.opened_device;
. . .
report_DDef:  Data_Definition_Mgt.DDef_AD;
. . .
initialized_report:  Device_Defs.opened_device;
. . .
  local_parts_file:  Device_Defs.device :=
      Record_AM.Ops.Get_device_object(
          Inventory_Files.parts_file);
. . .
  opened_local_parts_file:
      Device_Defs.opened_device;
. . .
begin
. . .
  opened_local_parts_file := Record_AM.Ops.Open(
      dev          => local_parts_file,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers);
```

Open the desired output device (must support the *byte stream access method*) by calling the appropriate `Open`:

- For terminal output, call `Character_Display_AM.Open`.
- For printer or file output, call `Byte_Stream_AM.Open`.

(from Inventory_Reports body)

```
  opened_output := Byte_Stream_AM.Open_by_name(
      name         =>
          output_dev_pathname,
      input_output =>
          Device_Defs.output);
```

Get the report DDef by calling `Directory_Mgt.Retrieve` with the stored DDef's pathname. You will then have to convert the `Retrieved` untyped word into a DDef AD, by calling an instance of `Unchecked_Conversion`.

(from Inventory_Reports specification)

```
report_by_part_DDef_str:  constant string :=
    "/example/inventory/DDefs/report_by_part";
. . .
report_by_part_DDef_pathname:
    System_Defs.text(
        report_by_part_DDef_str'length) := (
            report_by_part_DDef_str'length,
            report_by_part_DDef_str'length,
            report_by_part_DDef_str);
```

(from Inventory_Reports body)

```
  report_DDef := DDef_from_untyped(
     Directory_Mgt.Retrieve(
         name => report_by_part_DDef_pathname));
```

Initialize the report handler by calling `Report_Handler.Initialize` with the report DDef, and the opened input and output devices.

```
(from Inventory_Reports body)

  initialized_report := Report_Handler.Initialize(
      description => report_DDef,
      input       => opened_local_parts_file,
      output      => opened_output);
```

Print the report by calling `Report_Handler.Print` with the initialized report.

```
(from Inventory_Reports body)

  Report_Handler.Print(
      report => initialized_report);
```

Close the report's input file:

```
(from Inventory_Reports body)

  Record_AM.Ops.Close(
      opened_dev => opened_local_parts_file);
. . .
end Print_report_by_part;
```

V-2.2.9 Printing a Report from a Sorted File

Calls Used:

`Byte_Stream_AM.Open_by_name`	Opens the report output device.
`Directory_Mgt.Retrieve`	Retrieves the report definition, given the report's pathname.
`Event_Mgt.Wait_for_all`	Waits for all of a specified set of events to occur.
`Pipe_Mgt.Convert_pipe_to_device`	Converts a pipe AD into a device AD.
`Pipe_Mgt.Create_pipe`	Creates a new pipe.
`Process_Mgt.Deallocate`	Deallocates a spawned process.
`Process_Mgt.Get_process_globals_entry`	Gets one of the process globals entries (in this case, this process's AD).
`Process_Mgt.Spawn_process`	Creates a new process which runs concurrently with the calling process. A termination action is specified.
`Record_AM.Ops.Open`	Given a device AD, opens a file or device for record access.
`Record_AM.Open_by_name`	Given a pathname, opens a file or device for record access.
`Record_AM.Ops.Get_DDef`	Gets the DDef underlying an opened device.
`Record_AM.Ops.Close`	Closes a file.
`Report_Handler.Initialize`	Initializes an opened report, given the report DDef and the input and output devices.
`Report_Handler.Print`	Prints an initialized report.
`Sort_Merge_Interface.Sort`	Sorts records from an input device to an output device, using a sort DDef to specify the sort fields and their ordering.

This section describes how to sort a file and use the sorted records as input to a report. The example uses a pipe from the sort procedure to the report procedure.

Create a Report Definition

Create and store the report's DDef, as described in the previous section, "Creating a Simple Report".

Create a Sort Definition

Create the sort DDef, which defines the sort key fields and their ordering. This can be done interactively, with the **TBD** utility, or procedurally, using calls to `Data_Definition_Mgt`. Store the sort DDef under a pathname.

Your Sort Procedure

Implement the file-sorting procedure:

- Use the `subprogram_value` pragma to get an AD to your "Sort" procedure (process):

  (from Inventory_Reports body)

```
type connection_record is
. . .
  record
    sort_out:     Device_Defs.opened_device;
      -- Output from "Sort" to pipe.
    report_in:   Device_Defs.opened_device;
      -- Input from pipe to "Print".
    report_out:   Device_Defs.opened_device;
      -- Output device for "Print".
  end record;
. . .
procedure Sort(
  param_buffer:  System.address;
. . .
  param_length:  System.ordinal)
. . .
is

  conn_rec:  connection_record;
. . .
    FOR conn_rec USE AT param_buffer;
. . .
begin
. . .
end Sort;
  pragma subprogram_value(
      Process_Mgt.Initial_proc,
      Sort);
```

- Open the input file, with `Record_AM.Ops.Open` or `Record_AM.Ops.Open_by_name`.
- Get the sort DDef:

  (from Inventory_Reports body)

```
  opened_sort_DDef:
      Device_Defs.opened_device;
  sort_DDef_reference:
      Data_Definition_Mgt.node_reference;
. . .
  opened_sort_DDef := Record_AM.Open_by_name(
      name         =>
          sort_by_loc_DDef_pathname,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers,
      block        => true);
. . .
  sort_DDef_reference :=
      Record_AM.Ops.Get_DDef(
          opened_dev => opened_sort_DDef);
```

- Use the input end of the pipe for the output device.
- Call `Sort_Merge_Interface.Sort` with the sort DDef, and the opened input and output devices.

```
(from Inventory_Reports body)

Sort_Merge_Interface.Sort(
    input_device   =>
        opened_local_parts_file,
    DDef           => sort_DDef_reference,
    output_device  => conn_rec.sort_out,
    stable_sort    => true,
    tuning_opts    =>
        Sort_Merge_Interface.no_tuning);
```

- Close the input file, with `Record_AM.Ops.Close`.

Your Report Procedure

Implement the report procedure:

- Use the `subprogram_value` pragma to get an AD to your "Print" procedure (process).
- Get the report DDef:

```
(from Inventory_Reports body)

report_DDef:  Data_Definition_Mgt.DDef_AD;
. . .
report_DDef := DDef_from_untyped(
    Directory_Mgt.Retrieve(
        report_by_loc_DDef_pathname));
```

- Use the output end of the pipe for the report input device.
- Call `Report_Handler.Initialize` with the report DDef, and the opened report input and output devices.
- Call `Report_Handler.Print` to print the report.
- Close the report output device, by calling `Record_AM.Ops.Close`.

Creating a Pipe

Create the pipe, and open its input and output ends:

```
(from Inventory_Reports body)

sort_pipe:  Pipe_Mgt.pipe_AD;
. . .
sort_pipe := Pipe_Mgt.Create_pipe;
. . .
    sort_out     => Record_AM.Ops.Open(
        Pipe_Mgt.Convert_pipe_to_device(
            sort_pipe),
        Device_Defs.output),
    report_in    => Record_AM.Ops.Open(
        Pipe_Mgt.Convert_pipe_to_device(
            sort_pipe),
        Device_Defs.input),
```

Spawning Your Sort and Print Processes

Spawn your "Sort" and "Print" processes:

```
(from Inventory_Reports body)

     this_process_untyped:  System.untyped_word;
 . . .
     sort_process:  Process_Mgt_Types.process_AD;
 . . .
     print_process:  Process_Mgt_Types.process_AD;
 . . .
     this_process_untyped :=
         Process_Mgt.Get_process_globals_entry(
             Process_Mgt_Types.process);
 . . .
     sort_process := Process_Mgt.Spawn_process(
         init_proc    => Sort'subprogram_value,
         param_buffer => conn_rec'address,
         term_action  => (
             event       => Event_Mgt.user_1,
             message     => System.null_address,
             destination => this_process_untyped));
 . . .
     print_process := Process_Mgt.Spawn_process(
         init_proc    => Print'subprogram_value,
         param_buffer => conn_rec'address,
         term_action  => (
             event       => Event_Mgt.user_2,
             message     => System.null_address,
             destination => this_process_untyped));
```

Wait for completion of the two processes, then deallocate them:

```
(from Inventory_Reports body)

     term_events:  Event_Mgt.action_record_list(2);
 . . .
     Event_Mgt.Wait_for_all(
         events =>
             (Event_Mgt.user_1 .. Event_Mgt.user_2 =>
                  true,
              others => false),
         action_list => term_events);
 . . .
     Process_Mgt.Deallocate(sort_process);
     Process_Mgt.Deallocate(print_process);
```

V-2.3 Summary

- A BiiN™ program can be controlled by menus or commands.
- A data file's record layout can be used to generate default form and report formats. Form and report formats can be updated without changing the calling program.
- Input to a program can come from a window, a file, or a data entry form.
- Output from a program can be displayed in a form, written to a device, or printed as a report.

BUILDING NEW COMMANDS 3

Contents

This chapter describes how to process a program's invocation and runtime commands and arguments.

Packages Used:

`CL_Defs`	Contains declarations used by the command service, for processing command language (`CL`) arguments and variables.
`Command_Handler`	Contains operations for reading and processing program commands and arguments.
`Command_Execution`	Contains a procedural interface to command execution.

New commands for command-driven programs are created with the `manage.commands` command set in the `manage.program` utility. Your program calls the command service (with the command definition) to get and parse each command. Your program can then perform (implement) the returned command.

There are three ways in which you can create the command definitions for a program's commands:

1. Enter `manage.commands` runtime commands to create the command definitions (*command DDefs*) interactively.
2. Create `manage.commands` runtime commands in a command file. Submit the command file to the `manage.program` utility to create the command DDefs.
3. Include `manage.commands` runtime commands as tagged comment lines in a program's source file. Use the `:tagged_commands` argument to the `manage.program` utility to extract and process the definition commands from the tagged comment lines.

The invocation command DDef is stored with the program itself. Runtime commands sets are stored under pathnames, which your program then uses when opening command set processing.

V-3.1 Concepts

- *New commands are defined using the* `manage.commands` *command set in the* `manage.program` *utility* - see the *BiiN™ Command and Message Guide.*
- *The command service uses the command definitions* - to help the user correctly enter and complete a command.
- *There are three types of commands processed by the command service - built-in commands, CLEX commands,* and *program-defined commands*; see "Types of Commands", below.

built-in commands	Part of, and processed by, the command service itself. *Built-in runtime* commands directly perform some action, such as setting a variable (`set.variable`). *Built-in control* commands control the logical flow of commands (`if / then / else / endif`).
CLEX commands	Commands specific to the *command language executive* (CLEX).
program-defined commands	Each program has an *invocation command*, which is entered to CLEX

to invoke (execute) the program. Programs using the command service define their own *runtime commands*, which control and are implemented by the program.

- *Commands can be read from many different devices* - including terminals, pipes, and files.
- *Commands are entered and confirmed without program intervention* - a complete, correct command is then available to the program.
- *"help" is available to the user for each command and each argument* - the command service displays requested "help" texts (defined with the command DDef), without requiring any program action.
- A command consists of up to three parts:

```
command.name    [:argument=value]...
                [::control_option=value]...
```

`command.name` The name of the command. The command name may have two parts, separated by a period: `command_verb.command_noun`. See "Command Names", below.

`:argument = value`
Zero, one, or more arguments may be part of a command. See "Argument Types and Values", below.

- *Arguments have a name, a type, and a value* - an argument name is a string of characters, preceded by a colon (":", for example, `:argument_name`). The argument is one of seven types: boolean, integer, range, string, string list, pointer, or "derived" (actual value may be any of the preceding six types). The argument value's type must match the defined argument type.
- *Arguments may be mandatory or optional* - mandatory arguments must be entered with the command name; there is no default. Optional arguments may be entered to specify an argument value other than the default.
- *An optional argument's value may be entered or defaulted* - if an explicit value (`:argument=value`) is not entered, the argument has its *default* value, if any, defined with the command.

`::control_option = value`
There are several control options defined in the command language, used to request input/output redirection, background execution, and so on. See "Control Options", below.

Control options are processed by the command service. Control options entered with a command remain in effect until the next command is read (next `Command_Handler.Get_command` call).

- *An optional command history can be used* - to record commands entered.
  - The last `cli.num_last_cmds` commands entered are stored in a *command buffer* in active space.
  - Commands may be re-executed from the command buffer, either by index number, or by giving an unambiguous abbreviation (or a pattern to be matched) of a previous command.
  - The command history can be turned on and off as desired.

- *A command file is a sequence of commands stored in a text file* - command files can be included into the command input stream by using the *built-in* `include.command` command. A command file can be made into an executable script by using the `make.script` utility.
- *There are two methods for reading commands from an arbitrary file or device:*
  - Call `Command_Handler.Open_runtime_command_processing` on the file or device to return an opened command input device. Use `Command_Handler.Get_` calls with that command input device.
  - Enter the *built-in runtime* command `include.command`, specifying the file's or device's pathname. The specified file's or device's records are inserted into the command input stream. When the end of the file or device is reached, command input returns to the original device. The program reads the `included` commands as part of the original device's input command sequence.
- *The record access method is supported for runtime command input* - the command service's implementation of `Record_AM` allows non-Ada languages to read and process runtime commands. See the package description for `Command_Handler`, in the *BiiN™/OS Reference Manual.*
- *There is support for BiiN™/UX invocation command conventions* - for such constructions as `ls -ld`. The command service can set the expected invocation command variables (argv, argc, envp); see the *BiiN™/UX Commands Reference Manual.*

V-3.1.1 Developing Command-Driven Programs

This section describes how to develop a command-driven program.

Not all programs are suitable for command-oriented input. Some applications can use menu-oriented input (see Chapter V-5).

Most programs using the command service will be new utilities.

Determine the invocation command name and the arguments your program will use on entry. Create and store its command definition, as described above and in Section V-3.2.1, "Defining an Invocation Command", below.

To develop the runtime commands for your program:

- Each distinct function that the program performs should have its own command. Commands are defined during program development.
- Group related commands into *command sets*. Separate command sets may be defined for different program tasks or for different user groups.
- Create and store the command definition(s) as described above and in Section V-3.2.2, "Defining a Runtime Command Set", below.

V-3.1.2 Types of Commands

There are three types of commands processed by the command service: *built-in commands, CLEX commands*, and *program-defined commands*. All of these commands use the CL syntax.

built-in commands	Commands which are built into, and processed by, the command service.
CLEX commands	Commands which are recognized and processed by CLEX.

program-defined commands
: Commands which are specific to a program:

invocation command
: The command which is entered to CLEX to execute the program.

runtime commands
: Commands entered to and processed by your program.

  Runtime commands may be placed in a *startup command file*, to be automatically processed by your program directly after invocation, if desired.

Each of these three types of commands are described in the corresponding sections below.

V-3.1.2.1 Built-in Commands

Some commands are built into and processed by the command service. These commands can be entered to CLEX, and are part of all *command sets* (see "Program-Defined Commands", below).

There are two types of built-in commands, *built-in control commands* and *built-in runtime commands*. Each of these two types is described in the following tables.

Table V-3-1 describes the built-in control commands.

Table V-3-1. Built-in Control Commands

Control Commands	Description
`label` *label_name*	Labels a point in the command input stream; used by `goto`. The *label_name* may be any sequence of alphanumeric characters.
`goto` *label*	Transfers command input to the first command following the given label.

`if` *condition* `then`
    commands
`elsif` *condition*
    commands
`else`
    commands
`endif`
	Performs *commands* (any type), depending on the current evaluation of *condition*. A *condition* can be any boolean expression, for example, `($i < 5) OR $$exists (~/log/log_file)`.

`loop`
   commands
 `[if` *condition* `then`
    `[`*commands*`]`
    `exitloop`
 `endif]`
   `[`*commands*`]`
`endloop`
	Executes *commands* repeatedly, until an `exitloop` is executed, or forever.

`for` *range expression* `loop`
   commands
 `[exitloop]`
`endloop`
	Repeatedly executes the loop, once for each element in the *range expression*. The *range expression* may be either a range (`for $i in -3..5`), or a string list (`for $i in (a b c d e)`). An `exitloop` condition may be specified.

`while` *condition* `loop`
   commands
 `[exitloop]`
`endloop`
	While *condition* evaluates true, executes the loop. An `exitloop` may be specified.

Table V-3-2 describes the built-in runtime commands.

Built-in runtime commands may have appropriate control options, for example:

```
echo Hello ::output=x

echo Hello > x
```

Table V-3-2. Built-in Runtime Commands

Commands	Description
`echo`	Echoes a given value to the standard output. Useful for displaying information while in a command loop.
`run`	Executes a program or script.
`set.current_directory`	Sets the current directory to the given pathname.
`list.current_directory`	Lists the current directory's pathname.
`set.alias`	Defines an alias name for a given string.
`remove.alias`	Removes one or more alias names.
`list.alias`	Lists the current values of the given alias names.
`include.command`	Inserts the given file into the runtime command input. Useful for reading pre-defined command sequences stored in files.
`set.command_path`	Sets the current command path (command name space) to the given set (string list) of directory pathnames. With no argument value, updates the set of available commands, by searching through the current command name space, in order (of directories currently specified).
`list.command_path`	Lists the current pathnames in the command path.
`list.last_commands`	Lists the *last commands buffer*, out of which commands can be re-executed with `redo.last_commands`.
`redo.last_commands`	Repeats a previous command, or a sequence of commands.
`set.history_log`	Sets the scope (local or global) of history recording for subsequent jobs. If `local` (default), invoked jobs will not inherit the caller's history file. If `global`, invoked jobs do inherit the caller's history file.
`start.history_log`	Creates a user history file to be the current history, or restarts recording into the job history file after `stop.history_log` was called. If the user history file name is not specified, the default pathname is `~/$logon.history_dir/history`*timestamp*.
`stop.history_log`	Stops recording into the job history file, and into the command buffer.
`list.history_log`	Lists entries from the job history file, or from a specified history file.
`create.variable`	Creates a new CL variable in memory.
`set.variable`	Assigns a value to a CL variable.
`remove.variable`	Removes a CL variable from memory.
`list.variable`	Lists the names, types, modes, and values of the given variables.

V-3.1.2.2 CLEX Commands

Some commands are defined by the Command Language Executive (CLEX) for various system-related functions. See the *BiiN™ Command Language Executive Guide* for a complete description.

Programs and utilities are called from CLEX; each has its own invocation command.

V-3.1.2.3 Program-Defined Commands

There are two types of program-defined commands: the program's *invocation command*, entered to CLEX to invoke your program, and the program's *runtime commands*, which are processed by and control the program.

All programs (except ported C programs) should define their invocation commands. The invocation command definition contains the name of your program and defines any arguments.

Programs using the command service have *runtime commands*, grouped in *command sets*.

- A command set defines the names of all commands and their argument names, types, defaults, and permissible values. Command sets are defined during program development, using the `manage.program` utility.
- All command sets include all the *built-in* commands. See "Entering Commands to Programs", below.
- All command sets should have an `abort` or `exit` command defined, to stop command processing and terminate the program. These commands are **not** built in, but should be defined with each command set.
- The command set definition is a *command DDef*, stored under a pathname. The `Command_Handler.Open_` and `Command_Handler.Change_cmd_set` calls have a mandatory parameter for the pathname of the command set definition.
- There may be several command sets defined for one program; for example, one primary command set for general operations, some of which in turn have their own command sets. The current command set may be changed with `Command_Handler.Change_cmd_set`.
- `Command_Handler` calls use the command set definition to automatically check for correct command names, ensure completion of mandatory arguments, and supply default values for optional arguments.

All commands, except the invocation command itself, *must* have a command definition. If there is no invocation command definition, calls in this package will succeed if the invocation command uses CLEX syntax. No type checking, range checking, or consistency checking can be performed on the entered arguments (see below, "Reading the Invocation Command", in Techniques).

V-3.1.3 Review of Command Syntax

This section briefly reviews the three components of a command:

- command name
- command arguments, if any
- control options, if any

V-3.1.3.1 Command Name

The command name may be one word (for example, `echo`) or have two parts, separated by a period ("."; for example, `set.alias`). The two parts of a command name are the verb (for example, `set.`) and the noun (for example, `alias`).

For invocation commands (see "Types of Commands", above), the command name may be preceded by the absolute or relative pathname of the directory where the named program or script resides: `~/library/command.name`

There are some command verbs suggested for compatibility with the OS utilities:

`change`	For "update", "modify".

`list`	For "show", "display".
`remove`	For "delete", "kill".

V-3.1.3.2 Argument Types and Values

Argument types and values are briefly described.

Argument Types

Table V-3-3 describes each of the seven argument types.

An argument's default value may be a constant or a variable. In an argument's definition, any command language (CL) variable (of the correct type) can be used as the default value of an argument.

Table V-3-3. Argument Types

Argument Types	Description
`boolean`	Possible values are `true` or `false`. Boolean arguments normally have their defined default value `false`. Entering a boolean argument without a value is recognized as `true`. For example, `:boolean_argument` is the same as `:boolean_argument = true`.
`integer`	A sequence of numbers, possibly including underscores ("_"), and optionally preceded by a plus ("+") or minus ("-") sign. Possible values are in the range `-2^31` (`-2_147_483_648`) to `2^32-1` (`4_294_967_295`).
`range`	Two integers, separated by two periods (".."), for example, `-5..3`. Either or both of the low and high values may be defaulted.
`string`	Any sequence of characters, possibly enclosed in double quotation marks (" " ", for example, `"string of characters"`).
`string list`	One or more strings, enclosed in parentheses "`()`", separated by spaces; for example, `("string value 1" string2 "string value 3")`. There are several types (*lexical classes*) of strings, such as `blankless` and `symbolic` (for pathnames); see the *BiiN™ Command and Message Guide*.
`pointer`	A relative or absolute pathname to a stored object; for example, `~/object`.
`derived`	Any of the above six types. The actual type of a `derived` variable type is determined (derived) by the command service from the format of the entered value.

Argument Values

An argument value can be entered with or without the argument name, that is, in *named* or *positional* notation.

named notation — The argument name is followed by an equals sign ("=") and the argument value. Boolean arguments are a special case; entering the name of a boolean argument without a value (`:boolean_arg`) is the same as entering `:boolean_arg = true`.

For example:

```
:boolean_argument
   (or)
:boolean_argument = true

:integer_argument = 15
```

positional notation — Argument values entered by themselves are assigned to arguments in sequence (the sequence of arguments in the command definition). That is, the first unentered argument (one which does not already have a value entered in named notation) is assigned the entered value.

For example, the first argument of the *built-in* command `set.current_directory` is the directory name (`:directory`). Entering `set.current_directory ~/my_dir` is equivalent to entering `set.current_directory :directory = ~/my_dir`

Named and positional notation can both be used in a single command, if you are sure of the defined order of arguments.

An argument's value may be entered as a constant, variable, function, or expression:

constant	A simple value of the argument's type. For example, `:range_arg = 4..7`
variable	A variable name, of the same type as the argument, whose value is assigned to the argument. For example, `:string_arg = $user.name`
function	A CL function giving a value of the argument's type. For example, `:integer_arg = $$len($string_var)`
expression	A CL expression giving a value of the argument's type. For example, `:boolean_arg = (not $boolean_var)`

V-3.1.3.3 Control Options

Several control options are part of the command language. A control option is specified by a double colon ("`::`"), the control option's name (one of the names defined below), and possibly a value:

```
::control_option = value
```

The command service processes all control options. The given control options are set for the current command and remain in effect until the next `Command_Handler.Get_command` call.

Control options for *CLEX commands* are processed by CLEX.

Control options for *invocation commands* (see "Program-Defined Commands", above) are also processed by CLEX, and are not directly available to the called program or script.

Table V-3-4 describes each of the control options currently defined.

Table V-3-4. Control Options for Runtime Commands

Control Options	Description
`::input` or `<`	Specifies an input device's (or file's) pathname, redirecting the standard input device.
`::output` or `>`	Specifies an output device's (or file's) pathname, redirecting the standard output device.
`::output_extend` or `>>`	Whether to extend (if true) or overwrite (if false) the output device. Default is to overwrite.
`::message`	Specifies a message device's (or file's) pathname, redirecting the standard message device.
`::message_extend`	Whether to extend (if true) or overwrite (if false) the message device. Default is to overwrite.
`::window`	Requests that the command be executed in a separate window.
`::service`	Determines the *scheduling service object* (SSO) to be used for this command.
`::node`	Determines the node on which the command is to be run, for example, `::node=///my_node`.
`::history_log`	Starts a new history log file for this command, for example, `::history_log = ~/log/this_log`.
`::debug`	Specifies that this command is to be run in *debug mode* (see the *BiiN™ Application Debugger Guide*.
`::separate` or `&`	Requests that this command be executed as a separate job.

V-3.1.4 Review of Command Definitions

The following is a short example of the command definition syntax (that is, the runtime commands for the `manage.program` utility). See the *BiiN™ Command and Message Guide* for complete information.

All commands are part of a given command DDef. There will be one command DDef for the invocation command, and one for each command set.

An invocation command is defined as follows:

```
set.program  ~/example/my_program
  -- The invocation name is the same as the program's name

manage.commands  -- call the "manage.commands" command set

  create.invocation_command

    define.argument :arg_name = my_argument_1 \
                    :type     = string
      set.mandatory
    end

    define.argument :arg_name = my_argument_2 \
                    :type     = boolean
      set.mandatory
    end

  end
  exit -- exit "manage.commands"
exit -- exit "manage.program"
```

Runtime commands are defined in sets:

```
create.runtime_command_set  :cmd_def = ~/example/my_program.command_set

  define.command  :cmd_name = command.name_1
    define.argument :arg_name = command_argument_1 \
                   :type     = range
          set.mandatory
        end
      end

      define.command   :cmd_name = command.name_2
        define.argument :arg_name = command_argument_1 \
                        :type     = boolean
        end

        define.argument :arg_name = command_argument_2 \
                        :type     = range
        end
      end

      define.command   :cmd_name = exit
      end
    end
```

After creating a command DDef, it can be listed with the `list :cmd_def` runtime command.

Help Texts

Help texts are defined with each command and argument. Help texts are stored in help files. The default help file is part of a program's OEO.

Either use the default help file associated with a program, or set the help file to be used with `set.help_file` *message_file_pathname.*

Before the `end` of each defined command and argument, enter the help text, using the `set.description` command:

```
    create.runtime_command_set  :cmd_def = ~/example/my_program.command_set

      define.command   :cmd_name = command.name
        define.argument :arg_name = command_argument \
                        :type     = range
          set.mandatory
          set.description  :text = "
            Range of values for this command.
            "
        end

        set.description  :text = "
          Performs a given action, using the
          values specified with ':command_argument'.
          "
      end

      define.command   :cmd_name = exit
        set.description  :text = "
          Exits the program.
          "
      end

    end
```

V-3.1.5 Types of Command Input

Any device supporting the byte stream access method can be used for command input.

The usual interactive command input device is the standard input.

After `Command_Handler.Get_command` has been called to parse a new command (and return the command name and command index), the command may be read in several ways:

- In parts, reading each argument, with a series `Command_Handler.Get_`*argument_type* calls. See the appropriate sections under Techniques, below.
- As a string, using `Command_Handler.Get_command_string`.
- As a record, using `Record_AM.Ops.Read`. See the package description for `Command_Handler`, in the *BiiN™/OS Reference Manual.*

Startup Command File

Runtime commands can be stored in a particular `startup` file, to be read automatically by your program. After the startup file has been read, runtime commands may be entered.

Call `Command_Handler.Open_startup_command_processing` to access the startup command file, then call `Command_Handler.Get_command` to get each command.

Process each command as a runtime command (see "Reading Runtime Commands", below).

When the last command (for example, `exit`) is read, or the end of the startup file is reached, close the startup command input device by calling `Command_Handler.Close`.

Changing the Command Set

The `Command_Handler.Change_cmd_set` procedure changes the current command set definition.

Reading Commands as Records

The command service supports the record access method for command input. Programs written in non-Ada languages can use the record access method (`Record_AM.Ops.Read` call) to get a record of the current command.

See the package description for `Command_Handler`, in the *BiiN™/OS Reference Manual.*

V-3.1.6 Alternatives to Command Input

There are several alternatives to using commands to control your program:

menus	Read menu item selections. See Chapter V-5, *Programming with Menus.*
keyboard input	Read the keyboard directly. See Chapter IV-5, *Using Character Display I/O.*

V-3.1.7 Entering Commands to Programs

All command sets include all of the built-in commands. The user can enter these commands as desired; the command service processes these commands transparently to your program.

Some useful built-in commands are:

`run`	Executes any invocation command, for another program or script. For example, to display the current directory's entries, the user would enter `run "list.current_directory"`.

`include.command`	Includes a given command file into the command input stream. At the end of the file, command input returns to the default.
`.alias`	The three `.alias` commands (`set.alias`, `list.alias`, `remove.alias`) can set up aliases inside your program, for your runtime commands.
`.variable`	The four `.variable` commands (`create.variable`, `set.variable`, `list.variable`, `remove.variable`) can create and set variables for your program to read, or for use as argument values.
control	The control commands (`if / then / else / endif`, and the `loop / exitloop / endloop` constructs) can set up runtime command loops, possibly using variables.

V-3.2 Techniques

After reading this section, you will be able to:

- Define an invocation command
- Define runtime commands
- Create and store invocation and runtime commands
- Read the invocation command
- Read argument values
- Read a runtime command
- Read a command input line as entered
- Give a command to be executed by CLEX.

V-3.2.1 Defining an Invocation Command

Utility Used:

`manage.program`	Creates invocation command and runtime command set DDefs.

To define an invocation command:

1. Determine the invocation name of your program, and use that entry name for your executable program object.
2. Determine the arguments for the invocation command, their types and defaults, and whether each argument is mandatory or optional.
3. Create an invocation command definition, in one of two ways:
   - In your program source text, using tagged comment lines. Use the `:tagged_commands` argument to the `manage.program` utility to extract the command definition into a command file.
   - In a separate command file.

4. Use the `manage.program` utility, optionally with a file of definition commands as input, to create and store the invocation command's DDef.

V-3.2.2 Defining a Runtime Command Set

Utility Used:

`manage.program`
Creates invocation command and runtime command set DDefs.

Determine the names and arguments for each runtime command.

Follow the process described above, in "Defining an Invocation Command", to create the command set DDef(s).

V-3.2.3 Reading the Invocation Command

Calls Used:

`Command_Handler.Open_invocation_command_processing`
Opens a device for reading the invocation command.

To read the invocation command, just call `Command_Handler.Open_invocation_command_processing`; there is only one command, and it is already parsed.

Next, read each argument value, as described below ("Processing Command Arguments").

V-3.2.4 Processing Command Arguments

Calls Used:

`Command_Handler.Get_`*argument_type*
: Gets the value of a given argument of the current command.

  `Command_Handler.Get_boolean`

  `Command_Handler.Get_integer`

  `Command_Handler.Get_range`

  `Command_Handler.Get_pointer`

  `Command_Handler.Get_string`

  `Command_Handler.Get_enumeration_index`

  `Command_Handler.Get_string_list`

  `Command_Handler.Get_number_of_string_list_elements`

  `Command_Handler.Get_string_list_element`

`Command_Handler.Get_argument_info`
: Gets a record containing the name of an argument and the type and origin of its value. The origin of the argument's value is none, entered by the user, defined default value or defined default variable's value. This information is usually only relevant for `derived` argument types or if multiple default value sources are supported.

To read argument values, you must have an opened command input device and a *current command.* The current command is either the invocation command or has been gotten by `Command_Handler.Get_command`.

You must know at least the position (or name) of the argument:

argument name
: Defined by the command definition.

  For positional notation without a command definition (invocation commands only), the argument names default to `p1`, `p2`, `p3` and so forth, where *n* in p*n* is the argument position (`argument number`) in the command.

  The name `p0` is reserved for the command name. Its argument number is zero. The name `cmd_name` is also predefined for the invocation command's name.

argument number
: Defined by the command definition, or else by the position in the invocation command line.

  Argument number zero is the command name itself.

You may specify either the argument name or number in a `Command_Handler.Get_`*argument_type* call. For ease of program maintenance, using argument numbers is recommended (in case the names change).

- If both a name and a number are specified, the name is ignored.
- If no number is specified, the given name is used.
- If neither the argument name nor the argument number are specified, the call raises `System_Exceptions.bad_parameter`.

Call `Command_Handler.Get_argument_info` if necessary (always necessary for `derived` argument types) for the type of the argument and the origin of its current value.

Use the appropriate `Command_Handler.Get_`*argument_type* call to return the value of each argument.

If no value has been entered for an argument, and no default value is defined, `CL_Defs.no_value` is raised. This is only possible for arguments defined as "not mandatory", since the command service guarantees that all mandatory arguments have a value.

V-3.2.5 Processing Runtime Commands

Calls Used:

`Command_Handler.Get_command`
Gets and parses the next command from a given input device.

`Command_Handler.Get_`*argument_type*
Gets the value of an argument of the current command.

The most common way to read runtime commands is to open the runtime command input device, then use a loop to read and process runtime commands until an `exit` (or similar) command is entered.

V-3.2.6 Reading a Command Input Line as Text

Calls Used:

`Command_Handler.Get_line`
Gets a line of text from a given input device.

Reads a line of text, terminated by a carriage return/linefeed, directly from the command input device. This procedure can be used to read lines of data from the command input device, bypassing the command service's parsing mechanism.

An optional prompt can be specified, to alert the interactive user that the entered line will not be processed as a command.

V-3.2.7 Executing Commands from a Program

Calls Used:

`Command_Execution.Execute_command`
Executes one or more CLEX commands. Blocks until finished.

`Command_Execution.Run_program_or_script`
Executes one CLEX invocation command, in a separate job.

Set up a text record of the desired command(s), and make the appropriate call.

V-3.3 Summary

- Commands provide a consistent user control mechanism, used by all BiiN™ utilities.
- A command consists of the command name, arguments if any, and possibly some control options.
- The command service requires the user to enter a complete command; help is available for each command and for each argument.
- Commands can be read from any device supporting the byte stream access method, including files and pipes.
- Some commands are built-in; other commands are defined by programs using the command service (including CLEX itself).
- Built-in commands are part of every command set. These commands are intercepted and processed by the command service itself.
- New commands are defined with the `manage.commands` command set in the `manage.program` utility. Command DDefs are stored under a pathname.
- New commands are processed by the command service, using a command DDef. The parsed command can be read in parts, or as a string, using calls in the `Command_Handler` package. A command can be read as a record using the `Record_AM.Ops.Read` call.
- Your program can request execution of a CLEX command, optionally in a new CLEX instance, using calls in the `Command_Execution` package.
- Menus are another method for the user to control a program. See Chapter V-5, *Programming with Menus*.

PROGRAMMING WITH COMMAND LANGUAGE VARIABLES 4

Contents

This chapter describes how to create, read, set, and remove command language (CL) variables, using calls in the `Environment_Mgt` package. Some CL variables influence the application's environment; for example, the `cli.prompt` variable contains CLEX's prompt string. CL variables can also be used to save information and share it with subsequent jobs.

Packages Used:

`CL_Defs`	Contains declarations used by the command service, for processing command language (CL) arguments and variables.
`Environment_Mgt`	Contains operations to get, set, or remove local and global environment variables.

Figure V-4-1 shows how variables in passive store and active memory are related, and the order of evaluation for variables.

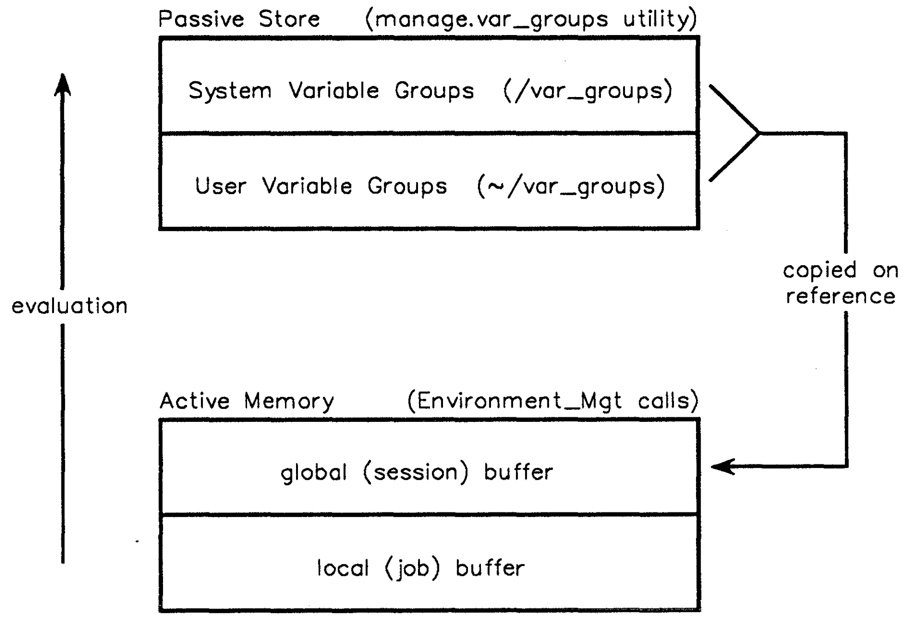

Passive Store (manage.var_groups utility)
System Variable Groups (/var_groups)
User Variable Groups (~/var_groups)
copied on reference
evaluation
Active Memory (Environment_Mgt calls)
global (session) buffer
local (job) buffer

Figure V-4-1. Command Language Variables

This chapter discusses the use and modification of variables, using calls in the `Environment_Mgt` package. For example:

- How to read a variable's value.
- How to set a variable's value. Variables are created when set, if they do not already exist.
- How to overwrite an existing variable's value.
- How to remove a variable.

V-4.1 Concepts

These concepts must be understood to use CL variables:

- *A variable has a name, a type, a mode, and a value -*

name	A CL string of letters and digits, up to `CL_Defs.max_name_sz` characters.
type	One of the `CL_Defs` types: `boolean`, `integer`, `range`, `string`, `string list`, or `pointer`.
mode	Either `CL_Defs.read_only`, indicating that the variable can be read but not assigned, or `CL_Defs.read_write`, indicating that the variable can be read or assigned a value.
value	Any value of the appropriate type.

- *There are two types of variables* - those dynamically created in active memory, using calls in `Environment_Mgt` (and the `.variable` commands built into the command service) and those defined in passive store, using the `manage.variable_groups` utility.
- *Dynamically created variables exist in one of two buffers* - either a *local* buffer or the *global* buffer. The local buffer exists for the duration of the job; the global buffer exists for the duration of the session. All processes in a job share the same variables (buffers).
- *Variables in passive store are stored in system and user groups* - system variable groups can only be updated by the system administrator. User variable groups are maintained with the `manage.variable_groups` utility.
  - *Sets of system or user variables* - may be collected together in groups, by giving them a group name. For example, `cli.prompt` is the `prompt` variable in the `cli.` group. Group variables may only be created with the `manage.variable_groups` utility.
  - *System variable groups* apply to all sessions on this node. System variables are created and maintained by the system administrator. System variables are in pre-defined groups, stored in the system directory `/var_groups`. See the "System Variables" section below.
  - *User variable groups* contain user-specific information, values, and defaults. User variables are created and maintained with the `manage.variable_groups` utility. User variables are stored in a "var_groups" directory in your home directory (`~/var_groups`).
  - *Job variables* are created and used by jobs, in the global buffer. Job variables may be created and used either by calls to `Environment_Mgt` or by programs using the `Command_Handler` package (programs with runtime commands). Subsequent jobs in this session inherit all existing job variables.
  - *Local variables* are used like job variables, but only exist for the duration of a job. Local variables are stored in the local buffer.
- *Variable names are evaluated upwards until found -*
  1. local buffer
  2. global buffer
  3. user variable groups (`~/var_groups` directory)
  4. system variable groups (`/var_groups` directory)

- *Each job has its own copy of system, user, and job variables (global buffer), and local variables (local buffer)* - referencing a user or system variable causes a copy of that variable to be created in the job's global variable buffer. Job variables may be created in either the global or local buffer. Local variables are created, set, and removed in the job's local variable buffer.
- *Subsequent jobs are affected by changes to the global buffer* - since they inherit a copy of the current global buffer.
- *Changes only affect the job's copy of the variables* - changes to stored (system or user) variables are actually made in the job's global variable buffer. Use the `manage.variable_groups` utility to change system and user variables permanently.
- *Variables may also be created and changed* - using four commands built into the command service:

`create.variable`	Creates a new local or global variable, optionally with an initial value.
`set.variable`	Sets a value into a variable which has mode `CL_Defs.read_write`.
`remove.variable`	Removes a variable from the local or global buffer. The version of the variable in passive store is not affected.
`list.variable`	Lists the type, mode, name, and current value of the specified variables.

- `"read_write"` *system and user variables can be hidden* - by creating a new variable with the same name in either of the job's buffers. `"read_only"` system or user variables cannot be hidden this way.
- *Rights for* `Environment_Mgt` *calls* - you must have *use* rights to read, *modify* rights to set, and *control* rights to create or remove variables. Group variables, being stored under pathnames, are subject to the usual access right restrictions.
- *All pre-defined variables* - are either part of a variable group, or are dynamically created (`OEO` and `status`).

`status`	`integer` variable created by each CLEX, before executing its first command. This variable is in the calling job's local buffer, and is set by CLEX from the exit status of the last executed command. The command service's built-in `run` command also creates a local `status` variable. The value of this variable is the exit status of the `run` program or script.
`OEO`	`pointer` variable created by the command service for every program. This variable is a pointer to the program's outside environment object (OEO), or null if there is no OEO. `OEO` is needed for the command service's implementation of `Record_AM`; see the `Command_Handler` package for more information.

V-4.1.1 System Variables

System variables are defined as part of CLEX. Current system variables are in six groups:

- logon
- cli (command line interface)

- pglob (process globals)
- user
- msg (message)
- ux (BiiN™/UX)

V-4.2 Techniques

After reading this section, you will be able to:

- Get a variable's type (boolean, integer, ...) and mode (`read` or `read_write`), then read the variable's value.
- Set a value into a variable. Variables are created when set, if necessary.
- Get and display all currently defined variable names.
- Display a variable's value, read a user input value, and set the variable. That is, convert to/from a string value from/to a variable's value.
- Remove a variable from active memory.

V-4.2.1 Read and Set an Environment Variable's Value

Calls Used:

`Environment_Mgt.Get_var_type`	Gets the type of a named variable.
`Environment_Mgt.Get_var_mode`	Gets the mode (`read` or `read_write`) of a named variable.
`Environment_Mgt.Get_`*type*	Gets a value from a variable of the named *type*.
`Environment_Mgt.Set_`*type*	Sets a value into a variable of the named *type*.
`Text_IO.Put`	Puts a string or character value out to the standard output.
`Text_IO.Get`	Gets a string or character value from the standard input.

The following program fragment shows how to read the type and mode of a variable, given a variable name.

If the given variable is of `CL_Defs.integer_type`, this program will read the variable's current value. If the variable is in `CL_Defs.read_write` mode, this program will increment the value by one, and then store the incremented value back into the integer variable.

```
(from Manage_Application_Environment_Ex.sb)

variable_name:  System_Defs.text(
    CL_Defs.max_name_sz);
variable_type:  CL_Defs.var_type;
variable_mode:  CL_Defs.var_mode;
. . .
integer_value:  integer;
. . .
Text_IO.Put("Enter a variable name:" );

Text_IO.Get(variable_name.value);

variable_type := Environment_Mgt.Get_var_type(
    var_name => variable_name);

variable_mode := Environment_Mgt.Get_var_mode(
    var_name => variable_name);

if variable_type = CL_Defs.integer_type then

  integer_value := Environment_Mgt.Get_integer(
      var_name => variable_name);
. . .
  if variable_mode = CL_Defs.read_write then
    integer_value := integer_value + 1;
. . .
  end if;            -- if "read_write"
. . .
end if;                      -- if "integer_type"
```

V-4.2.2 Display all Environment Variable Names

Calls Used:

`Environment_Mgt.Get_all_names`	Gets the names of all currently defined variables.
`Text_IO.Put`	Puts a string value out to the standard output.

The following program fragment shows how to read the names of all local variables, then put each name to the standard output.

(from Manage_Application_Environment_Ex.sb)

```
variable_name:  System_Defs.text(
    CL_Defs.max_name_sz);
. . .
variable_name_list:  System_Defs.string_list(1000);
. . .
Environment_Mgt.Get_all_names(
    group_name => System_Defs.null_text,
    list       => variable_name_list,
    global     => false);

Text_IO.Put_line("List of local variables:");

for i in 1 .. variable_name_list.count loop

  String_List_Mgt.Get_element(
      from    => variable_name_list,
      el_pos  => i,
      element => variable_name);

  Text_IO.Put_line(variable_name.value);

end loop;
```

V-4.2.3 Get and Set Environment Variable Values in ASCII

Calls Used:

`Environment_Mgt.Convert_and_get`	Gets an ASCII representation of a variable's value.
`Environment_Mgt.Convert_and_set`	Sets a variable from an ASCII representation of the value.
`Text_IO.Put`	Puts a string or character value out to the standard output.
`Text_IO.Get`	Gets a string or character value from the standard input.

The following program fragment asks for a variable name, then reads the type and mode of the variable. The current value of the variable is read as an ASCII representation, and displayed.

If the variable is in `CL_Defs.read_write` mode, the user is prompted to enter a new ASCII representation for the variable's value. The entered value is then set into the variable.

(from Manage_Application_Environment_Ex.sb)

```
variable_name:  System_Defs.text(
    CL_Defs.max_name_sz);
variable_type:  CL_Defs.var_type;
variable_mode:  CL_Defs.var_mode;
. . .
ASCII_value:    System_Defs.text(1000);
answer:         character;
. . .
Text_IO.Put("Enter a variable name:" );

Text_IO.Get(variable_name.value);

variable_type := Environment_Mgt.Get_var_type(
    var_name => variable_name);

variable_mode := Environment_Mgt.Get_var_mode(
    var_name => variable_name);
. . .
  Environment_Mgt.Convert_and_get(
      var_name => variable_name,
      value    => ASCII_value);

  Text_IO.Put("Value of ");
  Text_IO.Put(variable_name.value);
  Text_IO.Put(" variable is:");
  Text_IO.Put_line(ASCII_value.value);

  if variable_mode = CL_Defs.read_write then
. . .
          Text_IO.Put("Enter new value:");
          Text_IO.Get(ASCII_value.value);

          Environment_Mgt.Convert_and_set(
              var_name => variable_name,
              value    => ASCII_value,
              var_type => variable_type);
. . .
  else
    Text_IO.Put("Mode of ");
    Text_IO.Put(variable_name.value);
    Text_IO.Put_line(" variable is 'read-only'.");

  end if;                     -- if mode = read_write
```

V-4.2.4 Create and Remove an Environment Variable

Calls Used:

`Environment_Mgt.Set_`*type*
: Sets a value into a variable of the named *type*. The variable is created if it does not already exist.

`Environment_Mgt.Remove`
: Removes a variable. Locally created variables (`OEO` and `status`) and loop variables (`for i in range`) cannot be removed.

The following program fragments create a new local integer variable, named "`new_integer`". The new variable may be read or set as needed by the program. At the end of the program, the variable is removed (since it was a local variable, it would have disappeared at program termination anyway).

```
(from Manage_Application_Environment_Ex.sb)

    variable_name:  System_Defs.text(
        CL_Defs.max_name_sz);
. . .
    Text_Mgt.Set(
        dest   => variable_name,
        source => "new_integer");

    Environment_Mgt.Set_integer(
        var_name => variable_name,
        value    => 0,
        mode     => CL_Defs.read_write,
        global   => false);
. . .
    Text_Mgt.Set(
        dest   => variable_name,
        source => "new_integer");

    Environment_Mgt.Remove(
        var_name => variable_name,
        quiet    => true,
        global   => false);
```

V-4.3 Summary

- Variables have a name, a type, a mode, and a value. The variable's type is one of the six CL types: `boolean`, `string`, `. . . .` The variable's mode is either `read_only` or `read_write`. The variable's value is of the appropriate type.
- Command language (CL) variables control aspects of the current CLEX instance (such as message type and language) and contain information for use by jobs and programs (such as the current directory).
- Global variables are inherited by subsequent jobs and processes in this session. Local variables are specific to the creating job.
- System and user environment variables in passive store are maintained by the `manage.variable_groups` utility.
- Environment variables may be set and changed using commands common to CLEX and the command handler service: `create.variable`, `set.variable`, `list.variable`, and `remove.variable`.
- Environment variable values can be read procedurally in two ways: as a value of the correct type (`Get_`*type* calls) or as an ASCII representation of the value (`Convert_and_get` call).
- Environment variable values can be set procedurally in two ways: with a value of the correct type (`Set_`*type* calls) or with an ASCII representation of the value (`Convert_and_set` call).

PROGRAMMING WITH MENUS 5

Contents

This chapter describes how to use menus defined by the `Window_Services` package. Menus are created procedurally using `Data_Definition_Mgt`. The resulting menu data definition (*menu DDef*) is stored under a pathname. This chapter describes some design considerations for menus and the procedural aspects of menu usage but does not describe the menu editor utility itself.

Packages Used:

`Character_Display_AM`
: Provides device-independent I/O to character display devices such as printers, plotters, and windows on character and graphics terminals.

`Window_Services`
: Provides windows on character and graphics terminals, including pull-down menus.

This chapter describes the following tasks:

- How to define a menu group.
- How to install and enable (display) a menu group in a window.
- How to determine the menu choice made by a user.
- How to change or remove a displayed menu group.

Figure V-5-1 shows a *menu group* in a window, with one of the menus currently selected. This figure also shows the relationship between the window service (which provides the menus and the window), the terminal access method (which returns the user's selection), and your BiiN™ program.

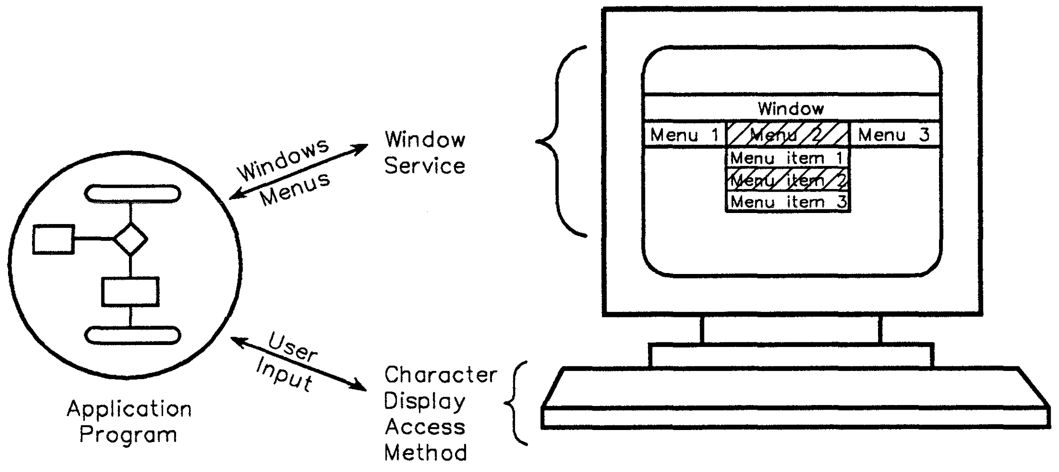


Figure V-5-1. BiiN™ Application Program and Menus

V-5.1 Concepts

- *Each menu is part of a menu group* - a menu group contains one or more menus.
- *A window can have only one menu group enabled* - several menu groups can be *installed* in one window, but only one menu group is *enabled* at any time.
- *A menu has a title and one or more menu items* - the user *selects* a menu title, causing that menu to appear, then selects one item from the menu.
- *Each menu item has a number and a string* - the menu item's string is displayed, and the menu item's number is returned when the item is selected. The returned menu item's selection record contains three numbers: the menu group, menu, and menu item numbers.
- *Menu items may be picked by the cursor or by index* - to choose an item, the user either moves the cursor onto the item and presses **<TBD>** or enters the displayed item's index number.
- *Menu items may have associated "help" messages* - the user can request an explanation of any menu item. The associated "help" message is displayed by the menu service, without program intervention.
- *Some special function keys are used with menus* - each implementation of menus defines its own.
- *Windows and this type of menu are provided by the window service* - in the `Window_Services` package and its `.Ops` nested package.
- *Menu item selections are input events* - the *character display access method* (`Character_Display_AM` package) provides a `Read` call to read such events. Your program may read a menu item selection, a keyboard or mouse event, or any of the window-related events (see `Terminal_Defs.input_enum` for a complete list of awaitable terminal input events). Note that mice are not supported by the character display access method.

V-5.1.1 Why Use Menus?

Menus provide an easy, standardized way to interact with the user:

- Menus provide a common display and user input format.
- There are no commands for the user to learn or remember.
- The user selects a menu item, without program intervention, resulting in one or more menu selections to be read.

V-5.2 Techniques

After reading this section, you will be able to:

- Define a menu group
- Install a menu group in a window
- Get the user's menu item selection
- Set a checkmark for a menu item
- Change a window's enabled menu group

- Remove a menu group from a window.

For information about creating and using windows, see Chapters IV-4 and IV-5.

V-5.2.1 Define a Menu Group

As part of program development, the menus to be used are defined. Design considerations when creating menus and menu groups include:

- Determining the logical operation to be performed by each menu selection.
- Grouping logically related items into menus. Each menu should contain from 2 to 10 menu items. (More than 10 menu items may be cumbersome for the user.)
- Grouping logically related menus into menu groups.
- Determining possible sets of menu groups: a menu selection may lead to a new menu group, under program control.

Each menu group, and its associated menus and their menu items, is created by calls to the DDef procedural interface (`Data_Definition_Mgt` package).

The `Make_Menu_DDef_Group_Ex` package, in Appendix A, shows how to procedurally create a simple menu group, containing two menus and five menu items: Menu 1 has two menu items; Menu 2 has three.

After being created, the menu group's DDef may be stored under a pathname. Your program then retrieves the menu group's DDef later by its pathname. The menu group is installed in a window and enabled, as described in the next section.

To change a menu item's text during program execution, call `Window_Services.Replace_menu_item_text`.

The following program fragment defines a menu group, menu, and some of the menu item numbers. These constants are used by the program to interpret a menu selection record (see "Get a Menu Selection", below).

```
(from Inventory_Menus specification)

  inv_menu_group_ID:  constant
      Terminal_Defs.menu_group_ID := 1;
. . .
  inquiry_menu_ID:       constant
      Terminal_Defs.menu_ID := 1;

  posting_menu_ID:       constant
      Terminal_Defs.menu_ID := 2;

  update_menu_ID:        constant
      Terminal_Defs.menu_ID := 3;

  report_menu_ID:        constant
      Terminal_Defs.menu_ID := 4;

  housekeeping_menu_ID:  constant
      Terminal_Defs.menu_ID := 5;

  exit_menu_ID:          constant
      Terminal_Defs.menu_ID := 6;

-- Inquiry menu items
  inq_by_part_item:  constant
      Terminal_Defs.menu_item_ID := 1;
  inq_by_desc_item:  constant
      Terminal_Defs.menu_item_ID := 2;
  inq_exit_item:     constant
      Terminal_Defs.menu_item_ID := 3;
```

V-5.2.2 Install a Menu Group in a Window

Calls Used:

`Window_Services.Install_menu_group`
Installs a menu group in a window.

The following program fragment retrieves the stored menu group's DDef, then installs the menu group in a window.

```
(from Inventory_Menus body)

menu_group_DDef_AD:  Data_Definition_Mgt.DDef_AD;
. . .
menu_group_node:
    Data_Definition_Mgt.node_reference;
. . .
  menu_group_DDef_AD := DDef_from_untyped(
      Directory_Mgt.Retrieve(
          name => menu_group_DDef_path));
. . .
  menu_group_node := Data_Definition_Mgt.
      Retrieve_DDef(
          DDef => menu_group_DDef_AD,
          name => menu_group_DDef_root_name);
. . .
      Window_Services.Ops.Install_menu_group(
        window     => Inventory_Windows.
                      main_window,
        menu_group => menu_group_node,
        ID         => inv_menu_group_ID);
```

V-5.2.3 Enable an Installed Menu Group

Calls Used:

`Window_Services.Ops.Menu_group_enable`
Displays the menu group and enables user menu item selection.

The following program fragment enables the menu group installed in the previous section.

```
(from Inventory_Menus body)

        Window_Services.Ops.Menu_group_enable(
          window     => Inventory_Windows.
                        main_window,
          menu_group => inv_menu_group_ID,
          enable     => true);
```

V-5.2.4 Get a Menu Selection

Calls Used:

`Character_Display_AM.Ops.Set_input_type_mask`
Determines the allowable types of user input from a window, including menu item selection.

`Character_Display_AM.Ops.Read`
Reads an input event from a window.

The following program fragment defines the types and variables for getting a menu selection.

```
(from Inventory_Main)

    menu_select:  Terminal_Defs.menu_selection;
. . .
    event_type:  Terminal_Defs.input_enum;
. . .
    event_num:   System.ordinal;
```

The following program fragment sets the window input mask to `menu_item_picked`, waits for the user's menu item selection, then calls the appropriate subprograms to perform the selection.

(from Inventory_Main)

```
Character_Display_AM.Ops.Set_input_type_mask(
    opened_dev => Inventory_Windows.main_window,
    new_mask   => Terminal_Defs.input_type_mask'(
        Terminal_Defs.menu_item_picked => true,
        others                         => false));
. . .
  Character_Display_AM.Ops.Read(
      opened_dev => Inventory_Windows.main_window,
      buffer_VA  => menu_select'address,
      max_events => 1,
      max_bytes  => 0,
      block      => true,    -- Wait . . .
      type_read  => event_type,
      num_read   => event_num);
. . .
  case menu_select.menu is

    when Inventory_Menus.inquiry_menu_ID =>
        Inventory_Menus.Process_inquiry_menu(
            selection  => menu_select.item);
```

V-5.2.5 Display a Checkmark for a Menu Item

Calls Used:

`Window_Services.Ops.Menu_item_check`
Displays a check mark (√) next to a given item in a menu.

Your program can display a (terminal-dependent) checkmark next to that menu item. The checkmark can indicate that the item is or was selected. For example, a menu of attributes for an object may have several attributes selected, with the selected attributes' menu items checkmarked.

V-5.2.6 Change a Window's Enabled Menu Group

Calls Used:

`Window_Services.Ops.Menu_group_enable`
Enables or disables an installed menu group in a window.

To disable the currently enabled menu group, call `Window_Services.Ops.Menu_group_enable` with the `enable` parameter false.

To enable another installed menu group in a window, just call `Window_Services.Ops.Menu_group_enable` for that menu group, with the `enable` parameter true. This implicitly disables the previously enabled menu group.

V-5.2.7 Remove an Installed Menu Group from a Window

Call Used:

`Window_Services.Ops.Remove_menu_group`
Removes an installed menu group from a window.

Call `Remove_menu_group` to remove an installed menu group. There is no change to any other installed menu groups (that is, none become enabled).

V-5.3 Summary

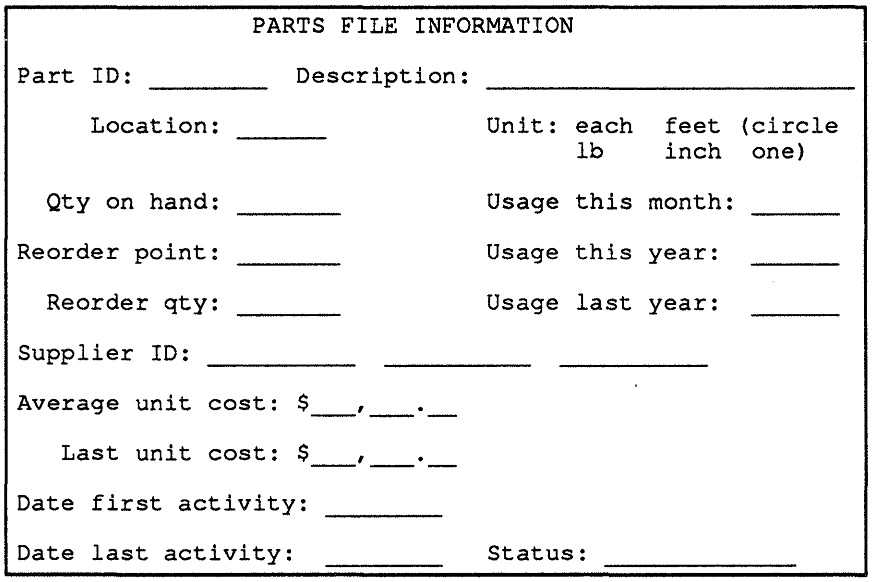
- Menus provide a consistent, easy-to-use user interface.
- A menu contains a menu title and one or more menu items.
- A menu group contains one or more menus.
- A window can have several menu groups installed, but only one menu group at a time can be enabled.
- After a menu group is installed and enabled, menu selections and menu "help" messages are displayed without program intervention.
- After the user has made a selection, an input event is available, containing the chosen menu group, menu, and menu item numbers.

UNDERSTANDING FORMS 6

Contents

The form service provides means to create, modify, test and execute forms which can be used interactively. Forms the form service displays on screens have the same general appearance as forms printed on paper that are frequently encountered in everyday transactions. Unlike paper forms, the forms created and controlled by the form service and directed by a high-level application program can perform a wide variety of functions dependent on the needs of the user and nature of the form.

Traditionally, a form has been a printed document with labelled spaces provided for writing in information. A typical paper form is shown in Figure V-6-1.

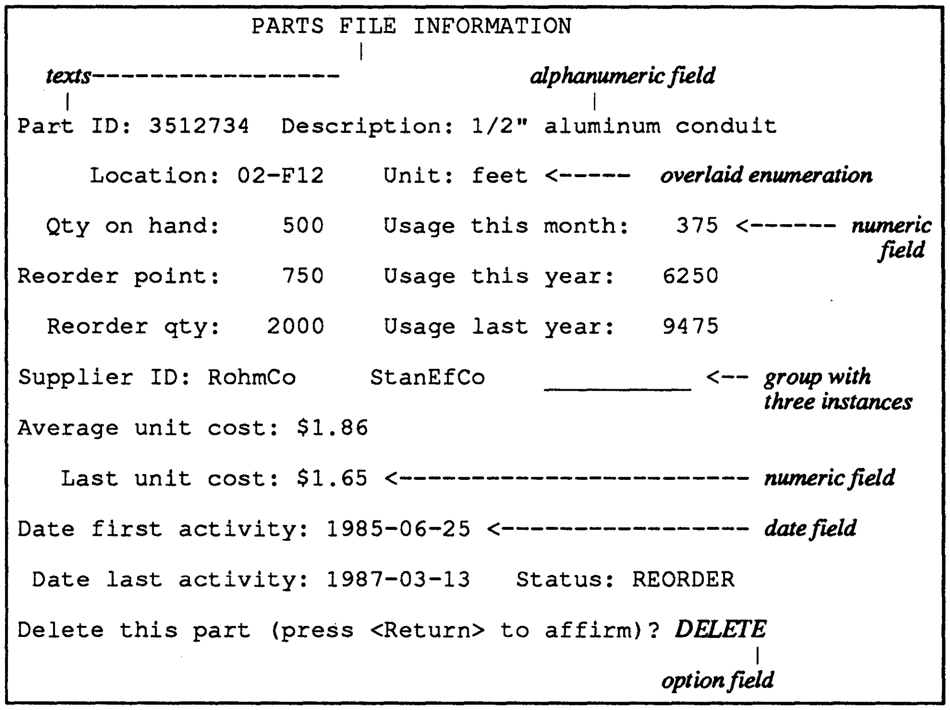


```
                 PARTS FILE INFORMATION

Part ID: ________  Description: _________________________

     Location: ______          Unit: each   feet  (circle
                                     lb     inch   one)

   Qty on hand: _______        Usage this month: ______

 Reorder point: _______        Usage this year:  ______

   Reorder qty: _______        Usage last year:  ______

Supplier ID: __________  __________  __________

Average unit cost: $___,___.__

   Last unit cost: $___,___.__

Date first activity: ________

Date last activity:  ________  Status: _____________
```

Figure V-6-1. Sample Paper Form

The form service builds on this concept to provide interactive forms capabilities on a terminal. For example, the cursor, which marks the current position in the form, may be moved back a space to erase an incorrect character, or back to the previous field to reenter a value. The contents of part of the form can be altered depending on the value of a previously entered field. Intermediate values can be calculated and stored transparently until needed later by the form. Even the order of execution of the form can be altered dynamically depending on the data entered.

This chapter describes the various parts of an interactive form and how they are combined into a single, executable form as shown in Figure V-6-2.



```
                PARTS FILE INFORMATION
                          |
 texts-----------------                alphanumeric field
  |                                         |
Part ID: 3512734  Description: 1/2" aluminum conduit

     Location: 02-F12    Unit: feet <-----  overlaid enumeration

  Qty on hand:    500    Usage this month:   375 <------ numeric
                                                           field
Reorder point:    750    Usage this year:   6250

  Reorder qty:   2000    Usage last year:   9475

Supplier ID: RohmCo     StanEfCo      __________  <-- group with
                                                      three instances
Average unit cost: $1.86

   Last unit cost: $1.65 <----------------------- numeric field

Date first activity: 1985-06-25 <---------------- date field

 Date last activity: 1987-03-13   Status: REORDER

Delete this part (press <Return> to affirm)? DELETE
                                                  |
                                            option field
```

Figure V-6-2. Annotated Executable Form

This section describes the form parts available for constructing an executable form. The next chapter, V-7 describes how an executable form is controlled.

Packages Required:

`Form_Defs`	Defines types and constants used by the `Form_Handler` package.
`Form_Handler`	Provides calls to process, control, and change forms.

`Form_Defs` contains the definitions for form properties (such as character display characteristics), symbolic keys (control keys, application keys, and information keys), and other definitions which describe the physical attributes and current operational status of a form.

`Form_Handler` provides calls to:

- Open and close a form.
- Execute a form.
- Modify data and control the execution network path.
- Query the state of the form, a form element, or the last user interaction.

V-6.1 Creating a Form Description

A form can be created with `edit.form` (the form editor), `create.form`, or procedurally with the `Data_Definition_Mgt` package.

The form editor is an interactive tool that enables a form developer to interactively create and modify form descriptions. This tool enables a form developer to design a form directly on the terminal screen and to define the properties for each form element as it is drawn and positioned. Detailed information for using the form editor is given in the *BiiN™ Systems Form Editor Guide*. Upon successful completion of an editing session, the form editor generates an executable form description.

`create.form` automatically creates the most simple, default form design based on a file's associated record description. The resulting form description can be used as input to the form editor for tailoring the form to the user's needs, or be executed as is.

Both the form editor and `create.form` generate a form description which can be executed by a user and controlled by `Form_Handler` calls.

Every form is represented as a *form DDef*. (*Form description* is a higher-level synonym for *form DDef.*) The form DDef defines the elements of a form, their order of execution, display attributes, location, etc. Application programmers will normally use the form editor to create a form description. Form descriptions can also be created procedurally using `Data_Definition_Mgt` although this method requires a detailed understanding of DDefs and is, therefore, not recommended. This low-level procedural interface is mainly of interest to implementors of interactive applications that create forms at runtime.

Form descriptions are stored with a directory entry and consequently are retrieved with `Directory_Mgt.Retrieve` when their AD is needed for calls such as `Form_Handler.Open_form`.

V-6.2 Record I/O

If record I/O is used for executing a form, a record description must be associated with the form. A record description describes the structure of a communication area used by an application program to communicate with an executing form. The primary benefit of employing record I/O is ease of use. It may not be appropriate for more complex applications for which `Form_Handler` calls are more effective.

If all or most of the screen fields that a form will use are already defined in an existing record description, associating a form with the related record description is usually the most effective means for transferring data. Given an associated record, `Form_Handler` can store and retrieve the data from the form and record with single calls. When a form is not associated with a record description, the data for each screen field must be stored and retrieved with individual `Form_Handler` calls.

V-6.3 Form Elements

A form may consist of the following *form elements*:

- Texts

- Screen fields
- Enumerations
- Subforms
- Groups
- Piles
- Data fields
- Subprogram interfaces
- Processing routines
- Key catchers
- Key lists.

The first five form elements (texts, screen fields, enumerations, subforms and groups) are called *sheet elements*. The remaining form elements affect the appearance of the form (piles), hold intermediate data values (data fields), and affect the execution of the form. The sheet elements are visible elements in the screen image of the form, called a *form sheet.* The form sheet is the rectangular area displayed on the screen.

V-6.4 Texts

Texts are strings which commonly serve as labels for screen fields. In Figure V-6-3, the string `Part ID:` is an example of a text in such a use. Texts may also be used independently for other purposes such as column headers or explanatory text.

V-6.5 Screen Fields

Screen fields are areas defined on the form sheet for receiving or storing user input. Screen fields include:

- Character
- Option.

V-6.5.1 Character Fields

Character fields are areas defined on the form sheet in which a user may enter data. Figure V-6-3 illustrates an character field with an associated *text* string.

Part ID: ________

Figure V-6-3. Character Field

A screen field is not required to have an associated text.

Character fields are of the following kinds:

- Numeric
- Alphanumeric
- Date.

Numeric fields have fixed lengths. Internally, they are represented as 4-byte integer, 8-byte integer, or 8-byte real. *Alphanumeric* fields may be of fixed or variable length depending on how they are defined.

The size of a numeric field on the screen depends on the *format definition.* There are no formats for alphanumeric fields. Table V-6-1 illustrates examples of numeric field formatting. See the *BiiN™ Systems Form Editor Guide* for detailed instruction on formatting numeric fields.

Table V-6-1. Examples of Numeric Formatting

Field Contents	Format String	Clear When Zero	Leading Text	Trailing Text	Displayed Field
10.34	99999.99	irrelevant			00010.34
10.34	*****.99	irrelevant			***10.34
5410.34	ZZ,ZZZ.99	irrelevant			5,410.34
0	99999.99	not set			00000.00
0	99999.99	set			
10.34	ZZZZ9.99	irrelevant	**	**	**10.34**
10.34	****9.99	irrelevant	$		$***10.34
12.34	ZZ9.9999999E+99	irrelevant			12.3400000E+00
-12.34	-9.999999eZZZ+	irrelevant			-1.234000E 1+
10.34	+0.999999E+99	irrelevant			+0.103400E+02

A *date* field is a special case of an character field.

Data fields require a format string. The format string is comprised of replacement characters and insertion characters. Replacement characters may include:

YY - Last two digits of a year.

YYYY - All four digits of a year.

MM - Integer value of a month.

MMM - Abbreviation of a month.

DD - Interger value of a day.

DDD - Abbreviation of a day.

HH - Integer value of hours.

II - Integer value of minutes.

SS - Integer value of seconds.

Insertion characters are printable characters that provide explanatory text or punctuation. Examples of data formatting showing the use of replacement and insertion characters are illustrated in Table V-6-2:

Table V-6-2. Examples of Date Formatting

Field Contents	Format String	Displayed Field
1984 11 28 12 01 09	YYYY-MM-DD	1984-11-28
1984 11 28 12 01 09	YYYY:Mmm:DD	1984:Nov:28
1984 11 28 12 01 09	Year:yyyy Month:Mmm Day:dd	Year:1984 Month:Nov Day:28
1984 11 28 12 01 09	DD MM YY	28 11 84
1984 12 24 12 01 09	DDD-YYYY	359-1984
1984 11 28 12 01 09	HH:II:SS	12:01:09
1984 11 28 12 01 09	/II/	/01/
1984 12 24 12 01 09	DDD days HH hours	359 days 12 hours

The contents of a date field are automatically validated after entry. If the contents of a part of the field are invalid, the local cursor is positioned at the beginning of that part.

Default formats for these varieties of screen fields are shown in Table V-6-3:

Table V-6-3. Default Screen Field Formats

Default Format	Type
-zzzzzzzzz9	int4
-zzzzzzzzzzzzzzzzzz9	int8
-9.9999999999E-99	real8
yyyy-mm-dd	date

V-6.5.2 Option Fields

An *option* field is composed of a visible text string which may be selected and deselected with a symbolic key or a mouse. The actual data transmitted is a boolean value indicating whether or not the field is selected. Figure V-6-4 illustrates an option field.

```
Delete this part (press <Return> to affirm)? DELETE
```

Figure V-6-4. Option Field

In Figure V-6-4, the `DELETE` string is an option field. An option field is highlighted when selected (true) and displayed in normal intensity when not selected (false). An option field is selected and deselected with the **<select local>** key.

V-6.6 Enumeration

Enumerations are sheet elements consisting of an ordered set of values. The values of an enumeration are represented with texts. Each enumeration is assigned a nonnegative integer value. Each tuple (value, screen representation) is called an *element* of the enumeration. If an element's representation is empty, the element is called the *null element* of the enumeration.

Enumeration elementss are selected with the **<select local>** key. When selected, the element is highlighted. Any previously selected element of the enumeration is simultaneously returned to normal intensity. The value of the currently highlighted element is the value assigned to the enumeration when the enumeration is left. Characters cannot be entered into enumeration elements.

Enumerations may be one of two kinds: overlaid and scattered.

An *overlaid* enumeration is an enumeration in which the enumeration elements are displayed one at a time. The user toggles through the enumerations by using the **<select local>** key. The enumeration element currently displayed when the enumeration is left (commonly by pressing the **<next>** key) defines the value which will be assigned to the enumeration.

Figures V-6-5 and V-6-6 illustrate the first two enumeration elements of an overlaid enumeration field with the text title *Unit:* and four units of measure as enumeration elements.

```
Unit: each
```

Figure V-6-5. Overlaid Enumeration: Initial Value

After the user presses **<select local>**, the second enumeration value, `feet`, overlays the first enumeration value, `each`.

```
Unit: feet
```

Figure V-6-6. Overlaid Enumeration: Subsequent Value

Each time **<select local>** is pressed, the next enumeration element will overlay the previous element until the last element is overlaid with the first and the cycle begins again.

A *scattered* enumeration is an enumeration in which all of the enumeration elements are displayed simultaneously. The screen representations of the elements are arranged within the rectangular area allocated to the enumeration. The element whose value represents the current value of the enumeration is highlighted while all other elements are displayed in the display attributes defined for the enumeration. Figure V-6-7 illustrates a scattered enumeration.

```
Unit: each  feet
      lb    inch
```

Figure V-6-7. Scattered Enumeration

The figure shows that the `feet` element is highlighted and, therefore, currently selected.

The local cursor can be moved from a screen field or enumeration to a scattered enumeration with the **<next>**, **<next with clear>** or **<previous>** keys. Once the cursor is in a scattered enumeration the **<next>**, **<next with clear>** or **<previous>** keys move the cursor between the enumeration elements. The element at which the cursor is positioned can be selected with the `select local>` key. The currently selected element is highlighted and the previously selected element (if any) is returned to normal intensity. The enumeration element currently highlighted when the enumeration is left defines the value which will be assigned to the enumeration.

V-6.6.1 Null Enumeration Element

A *null element* of an enumeration is an element which is associated with the value *empty*, meaning no value selected. An enumeration may or may not contain a null element. In an overlaid enumeration, the null element, if present, must have a screen representation (possibly containing only spaces).

If a scattered enumeration has a null value which is not assigned a screen representation, it can be selected with the `<delete>` key. If the enumeration has a null value that is represented on the screen, the value is selected like any other value.

V-6.7 Protecting Fields

Screen fields that are created with the *protected* property are used only for output. `Form_Handler.Store_value` is used to place the new value into the form. They are not included in the network of paths (see the "Execution Path" section for more information on a form's network of paths). These protected fields can be used, for example, to display the results of a calculation or logic decision. On the screen, they would appear the same as any other character field, but the user cannot enter data into them.

V-6.8 Data Fields

Data fields are not defined on the form sheet and are, therefore, not visible. They are used as storage areas for data used in computations or for exchanging data between processing routines, key catchers, and the application program.

V-6.9 Subforms

A *subform* is a form included in another form. This sheet element is provided as a convenience for making complex forms out of simpler forms. A subform can be created once and referenced by several other forms and subforms. Subforms may contain all the form elements allowed in a form, and may be nested.

V-6.10 Groups

A *group* is a subform which may be replicated. Each replication is called an *instance* of the group. The initial number of instances of a group is defined when the form is created, and may be modified during execution. Groups may contain any elements which may be used in a form including texts, screen fields, subforms, other groups, piles, processing routine calls, and key catcher calls.

For example, when the user enters the first *supplier ID* and leaves the field by using an application-defined key (that calls `Form_Handler.Create_group_instances`) rather than the usual key such as **<next>**, a second instance of the group is created so that the user, in this case, can enter a second *supplier ID*. In the sample form, a maximum of three instances can be displayed for character field for a supplier identification. Figure V-6-8 illustrates this group showing *supplier IDs* entered into the first two instances of the group, and the third instance created and awaiting entry of data.

```
Supplier ID: RohmCo  StanEfCo  __________
```

Figure V-6-8. Group Instances

Groups are *deployed* (displayed) horizontally to the right, or vertically and downward. Figures V-6-9 and V-6-10 illustrate the two deploying directions for group instances. (The group shown in Figure V-6-8 is deployed horizontally.)

```
instance #1 | instance #2 | instance #3
```

Figure V-6-9. Group Instances in a Horizontal Deployment

Groups may contain multiple sheet elements as shown in Figure V-6-10. These group instances contain name strings (alphanumeric fields), an enumeration of two values (Sex) and a numeric field with two insertion characters (SSN#).

```
Last Name          First Name        Sex      SSN#

_______________    ______________    M F    ___-__-____
_______________    ______________    M F    ___-__-____
_______________    ______________    M F    ___-__-____
_______________    ______________    M F    ___-__-____
```

Figure V-6-10. Group Instances with Multiple Sheet Elements

The number of instances of a group can be varied before or during the execution of the form with `Form_Handler.Create_group_instances` and `Form_Handler.Remove_group_instances` from within processing routines, key catchers or the application program.

The maximum number of instances of a group can be set by the form programmer when the form is created or modified. The number of instances of a group may be zero. In this case, the group occupies an area the size of one character (unless the group is a member of a pile in which case it occupies no space at all).

V-6.11 Piles

A *pile* is a sheet element occupying an area in a form in which other sheet elements can reside. Piles give the form designer control over the appearance of a form by specifying a fixed area of the form in which a choice of elements can be displayed.

While the locations of sheet elements not on a pile are determined directly by coordinates, piles offer an indirect method of positioning. A pile is a layout feature and cannot be executed. Processing routines or key catchers can be used to display pile elements. The order of execution of sheet elements is not affected by their being on a pile.

Piles, therefore, serve two primary functions. First, they reserve one area on the form for use by elements selected by the forms designer. Second, the pile's reserved area can be sized sufficiently for the anticipated maximum expansion of its variable-length elements so that expansion will not dislocate neighboring sheet elements thereby altering the appearance of the form. (Read about expansion and contraction of a form in the next section.)

When a pile is defined, one or more sheet elements are assigned to it. The order in which they are listed specifies their relative position or *rank*. Processing routines and key catchers determine which of the elements of pile are to be displayed. The selected pile elements are displayed in a horizontal or vertical deployment (specified when the pile is defined) in the order of their rank.

For example, a form determines which elements are to be displayed in a field depending upon whether the individual is (1) married and male, (2) married and female or (3) not married. The elements defined for the pile in this form include:

- Given Name of Spouse
- Premarital Name
- Age
- Separate Household

This example describes a form used to collect personal data. The following figure shows the form (without the specific pile elements shown).

```
       Name: ________________   Given Name: _______________

                  +------+--------+    +-------+
                  | male | female |    |married|
                  +------+--------+    +-------+

     o.............................................
     .                                             .
     .                                             .
     .              [pile]                         .
     .                                             .
     ...............................................

       Address: __________________________________________
```

Figure V-6-11. Form with a Pile

The following sheet elements is displayed in the pile of the form when "female" and "married" have been selected. (The deployment is vertical.)

```
     o.............................................
     . Given Name of Spouse:  ______________       .
     .                                             .
     . Premarital Name:       ______________       .
     ...............................................
```

Figure V-6-12. First Pile Usage

The following form element is displayed in the pile of the form when "male" and "married" have been selected.

```
o...............................................
. Given Name of Spouse: _____________          .
................................................
```

Figure V-6-13. Second Pile Usage

The following form element is displayed in the pile of the form when "married" has not been selected.

```
o...............................................
.            +-----------------+               .
. Age: __    |separate household|              .
.            +-----------------+               .
................................................
```

Figure V-6-14. Third Pile Usage

Piles may not be nested, but a group containing a pile may be located on a pile. Groups having no instances and a default instance of zero occupy an area the size of one character on a pile. Otherwise the area occupied is determined by the number of default instances.

V-6.12 Expansion and Contraction of Forms

Variable size sheet elements (alphanumeric screen fields of variable length and, optionally, piles and subforms) can *expand* beyond their default size when data is being entered into them. When they do expand horizontally, all of the sheet elements whose left boundaries are located right of the right border of the expanding element are automatically moved to the right. Likewise when they expand vertically, sheet elements whose upper boundaries are located beneath the lower boundary of the expanding element are automatically moved down. Thus, sheet elements are kept from being obscured by neighboring expanding elements. In table-like forms, column and line relationships are preserved.

If a sheet element is about to expand over the boundary of the form sheet, the form sheet expands; that is, the rectangular area occupied by the form sheet within the window's frame buffer expands. When the form sheet is constrained by the size of the frame buffer, it can no longer expand. All subsequent operations which require expansion of a sheet element are rejected.

Expansion of sheet elements not included within subforms and groups always affects subforms and groups as a whole unit. For example, a variable-length alphanumeric field that expands into the upper-left corner of a group instance will move all elements of the group as a unit regardless of whether any individual group element is in the path of the expanding field. The area that a subform or group instance occupies may grow if the group contains elements of variable size such as an alphanumeric screen field, group or pile.

When a form expands or contracts, the rectangular area containing a scattered enumeration is relocated as a whole unit.

If the deploying direction of a group is horizontal and if one or more instances of the group grows, instances with higher instance indexes are moved to the right but do not change their size. Figures V-6-15 and V-6-16 illustrate the effect of the growth of a group instance. Instance #1 grows horizontally, and instances #2 and #3, which do not change in size, move to the right.

```
instance #1  | instance #2  | instance #3
        screen field
```

Figure V-6-15. Effect of the Expansion of a Group Instance: Before Expansion

```
        instance #1       | instance #2  | instance #3
        screen field
```

Figure V-6-16. Effect of the Expansion of a Group Instance: After Expansion

The number of instances is restricted by the size of the frame buffer of the window in which the form is currently displayed. The frame buffer defines the limits of expansion for a form whether that expansion is by the addition of a group instance, or due to the expansion of a variable-length form element (such as an alphanumeric screen field).

Contraction is the opposite of expansion. When a sheet element contracts, the form may or may not contract depending on whether the contracting sheet element was the sole cause of the expansion. Contraction commonly occurs after an alphanumeric field is left. The area occupied by the field during data entry contracts to the rectangular size needed for displaying the current contents of the field or the size of the *minimum area* depending on which size is greater.

An alphanumeric screen field first expands horizontally until it reaches the maximum line length (specified when the field was created) and then expands vertically.

NOTE

When an element of a group, a field is not constrained in its expansion by the deployment direction of the group.

V-6.13 Subroutines and the Subroutine Interface

Two kinds of subroutines may be used in a form: *processing routines* and *key catchers*. Subroutines are incorporated into a form by specifying a *subprogram interface*. This interface specification contains:

- The name of the subprogram interface definition

- The name of the image module in which the subprogram is contained
- The name of the subprogram within the image module
- Link option (`link_at_bind_time`)
- A description of the formal parameters of the subprogram.

The name of the image module and the name of the subprogram within the image module are used to retrieve a subprogram value (an AD to a domain object plus a procedure entry offset). The subprogram value is retrieved by calling `Link_By_Call.Link`. The `Form_Handler` can then call the subprogram using the subprogram value.

The link option is set during form development. When set false, the subprogram is not linked to the form description at bind time. This makes it possible to create, bind and test a form whose processing routines and key catchers are not yet available. The form retrieves the subprogram value during the execution of the form when the subprogram is first called by the `Form_Handler`. When the link option is set to true, the subprogram is linked to the form description at bind time.

Subroutines must be implemented according to the *interlanguage calling conventions* (see the *BiiN™ Systems Programmer's Guide*) for the language in which the subroutine is written) in order to be callable by the form service. Also, subroutines which are to be linked to forms must be image modules (see the *BiiN™ Systems Linker Guide*).

V-6.14 Processing Routines

Processing routines are subroutines written in high-level languages which are executable form elements. They can be used to:

- Validate contents of screen fields
- Control the order in which screen fields are entered
- Modify contents of screen fields
- Modify the appearance of the form sheet depending on user input
- Perform any application-specific operations such as calculations.

Processing routines are included in the network of the form execution path, and therefore, are called when the execution of the form reaches the point where they reside in the network. Processing routines may make `Form_Handler` calls to:

- Create and remove group instances
- Change the display attributes of sheet elements
- Store and retrieve field values
- Alter the order of execution of the form
- Call other forms.

A processing routine call which has more than one successor in the form's network of paths must have a `next_path_element` parameter defined in its subprogram interface in order for it to proceed. The actual value of this parameter specifies a path element. The value is stored in the `next_path_element` path register by `Form_Handler`. Form execution will then continue at that element when the processing routine has finished executing. Processing routine calls with only a single successor must not have a `next_path_element` parameter.

Another parameter of the interface is `terminal_input`. This parameter is a byte string which is interpreted as a sequence of symbolic keys by `Form_Handler`, and inserted into the stream of input keys replacing the last key processed. Processing routines and key catchers can simulate user input by writing into this queue with this parameter.

More than one processing routine call may refer to the same subprogram interface. Therefore, a single processing routine may be called from several locations within a form's network of paths.

The subprogram interface describes the formal parameters of the subroutine and the processing routine call describes the actual parameters.

V-6.15 Key Catchers

Key catchers are subroutines written in high-level languages which are activated by pre-defined keystrokes. They can be used to trigger entire functions with a single keystroke. A key catcher is assigned to a *region* of a form. A region defines the area of effectiveness of a key catcher. It may include a single screen field or enumeration, a group, a subform or the entire form. A single key catcher may be assigned to several regions, or several key catchers may be assigned to the same region.

A subprogram interface for a key catcher is similar to a subprogram interface for a processing routine with the exception that a key catcher's interface does not include the `next_path_element` parameter, and does include the `trigger_key` parameter. `trigger_key` references an internal queue which contains the symbolic key that triggered the key catcher.

A key catcher for which a subprogram interface has been specified is included into a form by defining a *key catcher call*, and assigning it to the form, a subform, a screen field or enumeration. A key catcher call must have the following parameters:

- The name of the subprogram interface.
- A key list (a list of keys which are to be caught by the key catcher).
- The *actual parameters* (the actual values of the formal parameters specified in the subprogram interface).

When interpreting a keystroke, key catchers are scanned in the order in which they are assigned in the form description. Generally, the first key catcher in such a list is assigned to a screen field, and secondary key catchers are assigned to the group or subform. The least significant key catcher is assigned to the region defined by the entire form.

When the form user enters a character, the character is transformed into the corresponding symbolic key. Then the key lists of the effective key catchers are searched for that symbolic key according to the ordering of the key catchers. This search is performed for every entered character. The search stops when the symbolic key is found in a key list. Then the key catcher pertaining to that list is called. If the search does not succeed, the input character is processed by the `Form_Handler` according to the type of the current sheet element.

V-6.16 Symbolic Keys

A *symbolic key* is a printable character, a control key, an application key or an information key. Printable characters correspond to the ASCII characters in the range 20 hex to 7E hex. A standard set of *control keys* are predefined in `Form_Defs` and enable the user to trigger functions used in typical form dialogues. An example of a symbolic key is **<previous>** which moves the cursor back to the previous screen field or enumeration. *Application keys*, which are also declared in `Form_Defs`, trigger application-defined functions. The application keys give the form developer the opportunity to customize the form with unique features.

Information keys differ from the other symbolic keys in that they are input events from the terminal. These keys may be included into key lists and caught by application-defined key catchers. If they are caught by the `Form_Handler`, they do not trigger any action.

Symbolic keys are used to mask differences among terminal keyboards thereby contributing to the device-independent benefits of using fornm services. The tables V-6-4, V-6-5 and V-6-6 contain definitions for the symbolic keys.

Table V-6-4. Control Keys

Mnemonic Name	Function	Value (hex)
abort_execution	Aborts execution of the form.	0100
backspace	Moves the cursor to the left by one space in the active screen field or one part in a date field.	0101
begin_of_element	Moves the cursor to the first character input position.	0102
begin_of_line	Moves the cursor to the first character position of the current line.	0103
bel	Causes an audible or visible signal on the terminal.	0104
close_requested	Requests that the form sheet window be closed.	0105
correct	Displays the data input to a numeric screen field without formatting. This key has no effect on nonnumeric screen fields.	0106
delete_character	Deletes the character (or the part of a date field) under the cursor.	0107
delete_character_left	Deletes the character to the left of the cursor.	0108
delete	Deletes active screen field's input characters, replacing them with null characters. Also, selects the null element (if any) of a scattered enumeration if the null element has no screen representation.	0109
down	Moves the cursor to the next line in a multiple-line screen field.	010A
end_of_form	Skips to the end of the form, or to the next compulsory screen field. Sets the `destination` path register to `/END`.	010B
forward_space	Moves the cursor one space (or one part in a date field) to the right.	010C
global_help	Displays help information for the form.	010D
help	Displays help information for this screen field.	010E
home	Returns the input cursor to the beginning of the form. Sets the `destination` path register to `/BEGIN`.	010F

Table V-6-4. Control Keys (cont.)		
Mnemonic Name	**Function**	**Value (hex)**
insert_space	Inserts a blank space at the cursor position. In a date field, affects only the part of the date at the cursor position.	0110
insert_overwrite	Switches between *insert* and *overwrite* mode. In insert mode, existing characters move right to make room for new characters. In overwrite mode, existing characters are replaced by new characters that are input in their position.	0111
next	When the cursor is in a screen field or overlaid enumeration, skips to the next screen field, enumeration or to the end of the form. When the cursor is in a scattered enumeration, advances the cursor to the next enumeration element. When the cursor is in a part of a date field, advances the cursor to the next part of the date field.	0112
next_with_clear	Deletes the rest of the current screen field starting at the current cursor position (does nothing in an overlaid enumeration), then skips to the next screen field, or enumeration or to the end of the form. When the cursor is in a scattered enumeration, skips to the next element of the enumeration.	0113
previous	Moves the cursor to the beginning of the previous screen field. Sets the `destination` path register to the last touched screen field. When the cursor is in an overlaid enumeration, skips to the previous screen field or enumeration. When the cursor is in a scattered enumeration, skips to the previous enumeration element. When the cursor is in a date field, moves the cursor to the previous part of the date field.	0114
refresh	Refreshes (redisplays) the form image.	0115
reset	Resets the form to its defined initial state, then restarts form entry at the first field.	0116
restore	Restores the previous value of a field.	0117
button_1_released	The first mouse button has been released.	0118
button_2_released	The second mouse button has been released.	0119
button_3_released	The third mouse button has been released.	011A
button_4_released	The fourth mouse button has been released.	011B
button_5_released	The first mouse button has been released.	011C
select_local	Selects and deselects the value of option screen fields, displays the next element in an overlaid enumeration and selects the current element of a scattered enumeration.	011D
menu_item_picked	Indicates that a menu item was selected.	011E
up	Moves the cursor to the previous line in a multiple-line screen field.	011F

Table V-6-5. Application Keys

Mnemonic Name	Function	Value (hex)
f_1	Application key 1.	0200
f_2	Application key 2.	0201
f_3	Application key 3.	0202
f_4	Application key 4.	0203
f_5	Application key 5.	0204
f_6	Application key 6.	0205
f_7	Application key 7.	0206
f_8	Application key 8.	0207
f_9	Application key 9.	0208
f_10	Application key 10.	0209
f_11	Application key 11.	020A
f_12	Application key 12.	020B
f_13	Application key 13.	020C
f_14	Application key 14.	020D
f_15	Application key 15.	020E
f_16	Application key 16.	020F
f_17	Application key 17.	0210
f_18	Application key 18.	0211
f_19	Application key 19.	0212
f_20	Application key 20.	0213

Table V-6-6. Information Keys

Mnemonic Name	Function	Value (hex)
button_1_pressed	The first mouse button been pressed.	0300
button_2_pressed	The second mouse button been pressed.	0301
button_3_pressed	The third mouse button been pressed.	0302
button_4_pressed	The fourth mouse button been pressed.	0303
button_5_pressed	The five mouse button been pressed.	0304
input_focus_gained	The window containing the form gained the input focus.	0305
input_focus_lost	The window containing the form lost the input focus.	0306
overlap_changed	The visibility of the window containing the form changed.	0307
size_changed	The size of the window containing the form changed.	0308
view_changed	The position of the view of the form changed.	0309
position_changed	The position of the window containing the form changed.	030A
scroll_requested	Kind of scrolling requested: panning, bar, or dragging.	030B
user_defined_event	A user-defined event.	030C

By default, control keys are effective over an entire form. They enable a user to trigger commonly used functions. The form programmer can disable control keys or give them other functions by catching them with a key catcher.

V-6.17 Key Lists

A *key list* contains names of printable and symbolic keys which are to be captured by the associated key catcher. Key lists can be created and modified with the form editor.

V-6.18 Form Name Environments

The name of a form element is called a *basename*. A basename is represented by a string of ASCII characters. To address all elements of a form, form service distinguishes between three name environments within a form:

Form name environment - Names of all elements of a form with the exception of those contained in a group or subform of the form.

Subform name environment - Names of all elements of a simple or group subform with the exception of those contained in a subform of that subform.

Form global name environment - Names of all elements of the form including those contained in subforms.

The names of the elements of a form or subform must be unique within the name environment of the form or subform.

Instances of group subforms are named by the basename of the group followed by the number of the instance (index) in parentheses.

To address elements within subforms, a form network pathname is constructed of one or more basenames or indexed basenames separated by a "/" (slash). For example, `/group_3(2)/screen_field_a` is the form network pathname for `screen_field_a` of the second instance of `group_3` of the form.

An *absolute* pathname starts with a "/", and is evaluated starting with the name environment of the form. The simplest absolute pathname is the slash by itself that addresses the name environment of the form.

A *relative* pathname is any pathname that does not start with a slash, and is evaluated from the name environment of the currently executing subform or group instance, or from the form name environment if no subform or group instance is executing.

The pathname "." (dot) represents the name environment of the form, subform or group instance currently executing. Similarly, the pathname ".." represents the *parent* subform or group instance of the current subform or group instance, or the form if there is no parent subform or group instance.

When no pathname (null string) is specified, the present form element is considered to be contained in the name environment of the subform or group instance currently executing, or if there is none, it is considered to be in the form name environment.

V-6.19 Execution Paths

Execution of a form follows a path or a network of paths composed of the following elements:

- Screen fields
- Processing routines
- Subforms
- Groups
- Fictive, predefined path elements: `BEGIN` and `END`.

`BEGIN` is the path element of a network, subform, or group that has no predecessor. `END` is the path element of a network, subform, or group that has no successor. Processing routines are the only path elements which may have more than one successor. A form may contain form elements which are not included in the execution network (texts, piles, subroutine interfaces, data fields, key catcher calls and key lists). Screen fields and subforms may be included in the network of paths; process routine calls must be included.

The path elements are executed in an order that is determined by:

- the network of paths, and
- the contents of the path registers.

`destination` and `next_path_element` are predefined path registers that contain an arbitrarily selected path element and the normal successor to the current path element, respectively. They may be used by an application program, processing routines, and key catchers to influence the order in which path elements are executed.

`destination` denotes a target path element to which execution will proceed. `next_path_element` contains the next path element to be executed. Normally, execution will be permitted to follow the network as defined. However, conditions arise in which execution must deviate from the defined path such as when information entered into a screen field fails to pass a validation test and execution is returned to the current path element (screen field) for re-entry of the data.

Execution begins with the current element, the first path element or the next path element depending on the following possible values of `destination`:

- If `destination` denotes a successor, execution proceeds from the current element to the destination element.
- If `destination` denotes a predecessor, execution begins with the first path element of the form and proceeds until the destination is reached.
- If `destination` is empty, execution proceeds with the next path element.

V-6.19.1 Explicit Modification of the Path Registers

`destination` is explicitly set by an application program, a processing routine or a key catcher by calling `Form_Handler.Set_destination`.

`next_path_element` can only be modified with processing routines. The purpose of this register is to enable a processing routine to select one of its direct successor path elements. Selection of a successor is required if the routine has more than one successor.

V-6.19.2 Implicit Modification of the Path Registers

`destination` is implicitly set to `empty` if:

- The target path element is reached.
- The target path element cannot be reached.
- A screen field which requires input would have been skipped.

`next_path_element` is implicitly set if:

- The most recently processed path element has a single successor, then `next_path_element` is set to the name of the successor.
- `destination` is set to a predecessor, then `next_path_element` is set to `BEGIN`.

V-6.20 Messages and Help Information

Local help information may be optionally assigned to screen fields and enumerations. The form developer must specify the name of a message and the name of the message file that contains the help information. The message and the message file need not be available when the form is created but must exist by the time the form is executed. Local help messages can be accessed during execution by pressing the **<help>** key. Information relating to the entire form can be accessed during execution by pressing the **<global help>** key.

`Message_Adm` calls are used to define and store help information. Information messages are displayed on the standard message device. See the *BiiN™ Command and Message Guide* for additional information on messages.

V-6.21 Window Management

The window in which the form is displayed must have a frame buffer that is large enough to display the form with its current contents, otherwise `Form_Handler.Get` and `Form_Handler.Put` calls will fail. The size of the frame buffer limits the expansion of forms with variable size form elements.

V-6.22 Summary

- The form service builds upon the concept of a paper form to provide interactive forms capabilities on a terminal.
- A form can be created with the form editor or the `create.form` utility.
- A form may consist of the following *form elements*:
  - Texts
  - Screen fields
  - Enumerations
  - Data fields
  - Subforms
  - Groups
  - Piles
  - Subprogram interfaces
  - Processing routines
  - Key catchers
  - Key lists.
- Variable length alphanumeric screen fields and the screen elements containing them can expand to accommodate data being entered into the field.
- Execution of a form will follow a path or a network of paths composed of the following elements:
  - Screen fields
  - Processing routines
  - Subforms
  - Groups
  - Fictive, predefined path elements: **BEGIN** and **END.**
- The path elements are executed in an order that is determined by the network of paths and the contents of the path registers (`destination` and `next_path_element`).

PROGRAMMING WITH FORMS 7

Contents

This chapter describes how to use the procedural interface of the form service to control and modify forms before and during their execution. You should read V-5.3 before reading this chapter.

Packages Required:

`Form_Defs`	Defines types and constants used by the `Form_Handler` package.
`Form_Handler`	Provides calls to process, control, and change forms.

V-7.1 Creating Executable Forms

Developing an executable form involves:

- Designing a form
- Generating a form description with the form editor (`edit.form`) or `create.form`
- Creating and binding a message file
- Writing an application to execute a form
- Writing processing routines, key catchers and key lists, as needed
- Testing the form with the application.

The following procedure is recommended for accomplishing the above steps.

Step 1 - Design the Form

Determine the primary design considerations related to the physical layout and the logic controlling the execution of the form. These considerations may include:

- Names and locations of sheet elements
- Specifications for subforms, groups and piles
- The logic defining the network of paths
- Specifications for key catcher regions
- Functional descriptions of processing routines and key catchers including parameter specifications and `Form_Handler` calls.

If the form to be designed is based on a record description and can be executed sequentially without requiring any logic decision, no path logic considerations need be determined. The form editor provides a default path network. `create.form` also automatically provides a rudimentary, sequential, nonbranching path to the form.

Step 2 - Create the Form

Use the form editor, `create.form` or the DDef procedural interface to create a form description.

Step 3 - Test the Form

Use `test.form` to test and debug the execution logic, the validity of processing routine and key catcher calls, and the validity of the contents of fields.

Step 4 - Create a Message File

Use `manage.messages` to create a message file for the application program. This file can be used for local and global help messages triggered by the `<help>` key, and for messages generated by processing routines and key catchers.

Step 5 - Bind the Message File

For a stable set of forms, use `install.outside_environment` to bind the message file to the application program. (More volatile form applications may handle messages directly.)

Step 6 - Write an Application Program

A form is called by a high-level language program.

Step 7 - Write Subroutines, Translation Tables and Key Lists

Write processing routines and key catchers referenced in the form design. Use the translation table editor to create any translation tables needed in addition to the default translation table provided by the `Form_Handler`. (Translation tables map the ASCII sequences generated by input devices to symbolic keys.)

Step 8 - Test the Form with the Application

Use `test.form` to again test the form. Include desired debug features in the application program, processing routines, and key catchers.

Step 9 - Create a Window

To execute a form, this program must create a window for the form to execute within. A window must be provided before a form can be opened. Therefore, the application program calls `Window_Services.Create_window` or `Window_Services.Ops.Create_window` to provide a window for the form.

V-7.2 Command Language Variables

The following list contains the names, descriptions, types, and initial values of the CL (Command Language) variables used by the form service. CL variables affect the appearance and performance of the form editor and `Form_Handler`. See the *BiiN™ Systems Form Editor Guide* for instructions on setting a screen field so that it can accept a CL variable as input and for a description of the CL variables used to make *general adjustments* to the form editor. The *scope* of each of the variables may be:

H - Evaluated by the `Form_Handler`.

E - Evaluated by the form editor. The variables are valid throughout the editing session or are relavant only in the initialization phase.

D - Evaluated by the form editor. These variables provide default values for editor adjustments which may be changed during an editing session.

See V-1 for a general discussion of CL variables.

`form.decimal_character`

Character which will be displayed and accepted as the decimal symbol. The possible values are `Form_Defs.point` and `Form_Defs.comma`.

```
        Scope:  H, D
         Type:  string
Initial Value:  "."
```

`form.insert_mode`

Input mode set when the user starts to edit a new form. If true, mode is `insert`, else the mode is `overwrite`. The value of this variable can be toggled with the `<insert_overwrite>` key.

```
         Scope:  H, E
          Type:  boolean
 Initial Value:  true (insert)
```

`form.visual_bell`
Indicates whether signals sent to the terminal will be visual or audible. If true, signal is visual, else the signal is audible.

```
         Scope:  H, D
          Type:  boolean
 Initial Value:  false (audible)
```

`form.key_map` Symbolic name of the translation table that is used by the `Form_Handler` to translate incoming characters into symbolic keys. If null, a standard, internal translation table is used.

```
         Scope:  H
          Type:  string
 Initial Value:  null
```

`form.expansion_step`
Contains the number of characters by which a variable-length, alphanumeric field will expand horizontally when the present size is exceeded by data being entered into the field.

```
         Scope:  H, E
          Type:  integer
 Initial Value:  1
```

`form.escape_character`
The character which is used as the escape symbol in a format string. See the *BiiN™ Systems Form Editor Guide* for information concerning formatting screen fields.

```
         Scope:  H, D
          Type:  string
 Initial Value:  \ (backslash)
```

`form.editor_key_map`
Symbolic name of the translation table used by the form editor to translate incoming characters into symbolic keys. If null, a standard, internal translation table is used.

```
         Scope:  E
          Type:  string
 Initial Value:  null string
```

`form.window_position_line`
Line number of the upper-left position of the Info window. The upper line of the Main window depends on the CL variable `form.info_window_lines` and the upper line of the Message window is likewise dependent on the values of `form.info_window_lines` plus `form.main_window_lines`.

```
         Scope:  E
          Type:  integer
 Initial Value:  1
```

`form.window_position_column`
Column number of the upper-left position of the Info window.

```
         Scope:  E
          Type:  integer
 Initial Value:  1
```

`form.window_columns`

Width in columns of the three editing windows.

```
        Scope:  E
         Type:  integer
Initial Value:  80
```

`form.info_window_lines`

Number of lines in the Info window.

```
        Scope:  E
         Type:  integer
Initial Value:  10
```

`form.main_window_lines`

Number of lines in the Main window.

```
        Scope:  E
         Type:  integer
Initial Value:  10
```

`form.message_window_lines`

Number of lines in the Message window.

```
        Scope:  D
         Type:  integer
Initial Value:  1
```

`form.pop_up_message_window`

Determines whether the Message window will open and close upon the receipt of a message or stay open. If true, a Message window is opened each time a message is to be displayed, and closed when input is entered into any of the editor windows.

```
        Scope:  D
         Type:  boolean
Initial Value:  false
```

`form.editor_adjustments`

Symbolic name of the *form editor adjustments object.* This object contains adjustments that affect the appearance and operation of the form editor. Adjustments may be made and saved with the form editor. If this string is null, default adjustments are used.

```
        Scope:  E
         Type:  string
Initial Value:  null string
```

V-7.3 Form Utilities

The following utilities are provided to automatically create a simple, standard form, to test a form, and to map symbolic keys to specific terminals:

- `test.form`
- `create.form`
- Translation tables editor.

`test.form` interactively tests and debugs forms. It provides the following functions:

- Identifies any missing processing routines or key catchers.
- Provides information about fields.

- Displays and permits changing the contents of fields.

`test.form` displays and executes a form. When during execution of the form a processing routine or key catcher is found to be missing or a field's contents are invalid, execution is suspended and a message is displayed. The form tester may change or change the contents of fields.

The form tester may change the value of the predefined path registers `destination` and `next_path_element` while the form is executing. It is not possible, however, to modify the path.

When form execution is terminated, status information for the form displays.

`create.form` automatically generates the most simple, default form design based upon the description of the associated data record. This utility is called with the name of a record description and the name to be given the new form. It can be used with the form editor to customize a form.

V-7.4 Editing Translation Tables

Translation tables map the ASCII sequences generated by input devices to symbolic keys by associating a *raw* key (a sequence of keystrokes) with a symbolic key. Translation tables can be created and edited with the translation tables editor]. A translation table is required for each terminal on which a form will be executed thus providing terminal independence.

One default translation table is always associated with the `Form_Handler.` See `Form_Defs` for a description of the elements of this default translation table.

V-7.5 Techniques

After reading this section, you will be able to:

- Open, execute, and close a form
- Insert data into a form and retrieve data from a form
- Alter the order of execution
- Add and remove group instances before and during execution of a form
- Modify the appearance of form elements
- Retrieve information about the state of a form.

The sample code segments are excerpted from the `Inventory_Forms_Ex` example package.

V-7.5.1 Opening and Closing Forms

Calls Used:

`Form_Handler.Open_form`
Opens a form.

`Form_Handler.Close_form`
Closes a form.

The application program opens the form with `Open_form`. Then the form can be activated for dialogue. The status of the form is set to `initialized`, and defaults are assigned to the form element values. `Close_form` deallocates the opened form. The following excerpt shows a form being opened and closed:

```
        opened_form:  Form_Defs.opened_form_AD;
. . .
        opened_form := Form_Handler.Open_form(
            DDef => DDef_from_untyped(
                    Directory_Mgt.Retrieve(
                    name => form_pathname)));
        .
        .
[form is executed]
        .
        .
        Form_Handler.Close_form(
            opened_form_a => opened_form);
```

V-7.5.2 Executing Forms

Calls Used:

`Form_Handler.Get`
Executes a form (displays the form and accepts input from the user).

`Form_Handler.Compute`
Executes a form without displaying the form or requiring input.

`Form_Handler.Put`
Displays a form without executing it.

When `Get` is called, the form sheet is displayed and the form awaits user input. The opened form must have status `initialized`, `suspended`, or `input_required`. This is the most common method for executing a form as shown in the following example code:

```

        form_status := Form_Handler.Get(
            opened_form_a   => opened_form,
            opened_window_a => Inventory_Windows.
                               main_window);
```

`Compute` executes a form similar to `Get` but does not display the form or require input. Execution is suspended if input is required by a field. This call can be used to validate screen field values which receive their values from processing routines rather than user input.

`Put` displays a form while refusing user input. It is commonly used for displaying forms on output-only devices such as printers, or for displaying forms on terminals when no input is required.

V-7.5.3 Setting and Resetting the Initial State of a Form

Calls Used:

`Form_Handler.Set_initial_state`
Sets the status of the form to `initialized` and marks the current contents of the screen fields and enumerations as initial values.

`Form_Handler.Reset_form`
Resets a form to the same state as immediately after its last initialization.

`Form_Handler.Clear`
Clears a form from the screen.

`Set_initial_state` sets the status of the form to `initialized` and sets the current contents of the fields, the current number of instances of groups, and the current display attributes of fields and texts at their initial values.

Initial values are particularly significant under the following conditions:

- If the `<reset>` key is entered (or `Reset_form` is called) while the form is executing, the fields, group instances, and display attributes of fields and texts are reset to their respective initial values.
- If there is more than one data entry sequence defined by the network of paths, `Form_Handler` uses the initial values as necessary to keep the form consistent with itself. For example, the form user may enter data into all the fields of a path, then use symbolic keys to return to an earlier field in the path and change its value. If the user then causes execution to proceed along another path, `Form_Handler` implicitly resets the contents of the fields, group instances, and display attributes of fields and text in the first path to their initial values.

`Reset_form` returns a form to the state immediately after its last initialization. The last initialization may have been performed implicitly with `Open_form` or explicitly with `Set_initial_state`. See the `Interrupting Execution` section for an example of the use of this call.

`Clear` removes a form sheet from the window usually in preparation for a new operation. The status of the form remains unchanged.

```

    Form_Handler.Clear(
        opened_form_a => opened_form);

    Form_Handler.Close_form(
        opened_form_a => opened_form);
```

V-7.5.4 Inserting, Storing, and Deleting the Contents of Screen and Data Fields

Calls Used:

`Form_Handler.Store_value`
: Sets the value of a screen field, data field or enumeration element.

`Form_Handler.Fetch_value`
: Retrieves the value entered into a screen field, data field or enumeration element.

`Form_Handler.Delete_value`
: Empties a screen field, data field, or enumeration.

`Store_value` assigns a value to a screen field, data field, or enumeration. If the form is displayed, the new values of screen fields appear on the screen.

```
        Form_Handler.Store_value(
            opened_form_a    => opened_form,
            element          => desc_field,
            subunit          => System_Defs.null_text,
              -- added subunit; value correct?
            value_buffer_VA =>
                parts_record.desc'address,
            value_length     =>
                parts_record.desc'size/8,
            value_t          =>
                Data_Definition_Mgt.t_string);
```

`Fetch_value` retrieves the value entered into a screen field, data field or enumeration element.

```

      Form_Handler.Fetch_value(
          opened_form_a           => opened_form,
          element                 => part_ID_field,
          subunit                 => System_Defs.null_text,
            -- added subunit; value correct?
          value_buffer_VA        => part_ID'address,
          value_length           => part_ID'size/8,
          value_t                =>
              Data_Definition_Mgt.t_string,
          element_value_length => length,
          empty                  => empty);

      if empty then
        -- null part_ID; return to menu

```

`Delete_value` removes the value of a screen field, data field, enumeration which are contained in an associated record description, and which have a representation for the null value defined.

V-7.5.5 Controlling the Execution Path

A form will execute according to the network of paths defined when the form is created or modified. This network of paths may contain branches controlled by processing routines which determine the next executable element depending upon the value of entered or calculated data. Explicit control over the execution path is provided by `Set_destination`.

Calls Used:

`Form_Handler.Set_destination`
Stores a path element name in the `destination` path register.

`Set_destination` stores a path element name in the `destination` path register. If the path element is a successor of the current path element, all path elements up to the specified path element are executed. If the specified path element is a predecessor of the current path element, execution of the form starts again with the first path element of the form and continues to the specified element. A boolean may be set to indicate whether processing routines are to be executed.

The `next_path_element` path register, unlike `destination`, cannot be modified directly by the application or by `Form_Handler` but can be modified by a processing routine.

V-7.5.6 Processing Routines and Key Catchers

A subroutine (processing routine or key catcher) is added to a form with the form editor. The form editor incorporates a subroutine into a form by specifying a subprogram interface for the subroutine. To generate the subprogram interface, the editor requires:

- Programming language
- Name of the subprogram interface
- *in* parameters
- *out* parameters
- Link option.

V-7.5.7 Defining a Processing Routine

A processing routine is added to a form by defining the following items with the form editor:

- Name of the processing routine call
- Subprogram interface
- Name of the referenced form element
- *in* parameters
- *out* parameters.

`next_path_element` is an *out* parameter that specifies the name of a path element of the currently executed subform or form, and is stored in the `next_path_element` predefined path register. A processing routine with more than one direct successor must have this parameter. Processing routines with only one successor must not specify this parameter.

`terminal_input`, an optional *in* parameter, has an actual value of a byte string that is interpreted as a sequence of symbolic keys and inserted into the stream of input keys replacing the last key processed. This parameter references an internal queue where symbolic keys arriving from the keyboard are stored. A processing routine can write directly into this queue to simulate user input.

When a processing routine call is reached during the execution of a form, `Form_Handler` calls the processing routine specified by the corresponding subprogram interface. The same processing routine can be called from several locations within a form's network of paths.

V-7.5.8 Defining a Key Catcher

A key catcher is added to a form in the same way as a processing routine except for the additional following items:

- Key list
- Region of effectiveness.

A key list is created by the form editor and contains the keys to be caught by a key catcher. The region of effectiveness is a screen field, enumeration, subform or group to which a key catcher is assigned.

`trigger_key` is an *in* parameter which receives the value of the symbolic key that triggers the key catcher. It references a predefined `Form_Handler` register. This register enables the key catcher to inquire as to which key contained in the associated key list caused the call.

V-7.5.9 Interrupting Execution

Besides altering the execution path, the application may also stop execution. With the next two calls, execution can be arbitrarily terminated, or halted. These calls can only be made by processing routines or key catchers. When the processing routine or key catcher returns, the application again gains control.

`Form_Handler.Abort_form`
Aborts execution of a form.

`Form_Handler.Suspend_form`
Suspends execution of a form.

`Abort_form` halts and exits the execution of a form. All data entered during the current execution of the form is lost. This call can be made indirectly by pressing the `<abort>` key.

`Suspend_form` suspends execution of a form without losing the currently entered data and awaits a status change before execution is resumed. `Suspend_form` can only be called by a processing routine or a key catcher. The execution of the form is suspended when the calling processing routine or key catcher returns, and the application gains control. The application must call `Get` to resume execution of the form.

V-7.5.10 Adding and Removing Group Instances

The number of group instances required for any given execution of the form can vary according to the value of data entered. Therefore, the next two calls provide a means for increasing and decreasing the number of instances of a group prior to or during the execution of a form. When a form is created, a default number of instances is assigned to each group. This number may be changed dynamically with `Create_group_instance` and `Remove_group_instance`.

Calls Used:

`Form_Handler.Create_group_instance`
Creates group instances.

`Form_Handler.Remove_group_instance`
Removes group instances.

The following code shows creating a group instance.

```
begin
  -- Add another instance of the supplier ID group.
  Form_Handler.Create_group_instances(
      opened_form_a       => opened_form,
      group               => suppliers_field,
      number_of_instances => 1);

exception
  when Form_Handler.maximum_number_reached => null;

end;
```

In this example, the `Supplier ID` is a group of three instances. The first instance is displayed when the field is executed (default instances = 1). Entering a supplier ID and pressing the `<return>` key advances the cursor to the next screen field. If this part ID has a second supplier, the user presses the `<next>` key to display a second group instance. After the third group instance is displayed, the form service knows that this group has a maximum of three instances and will continue with the next path element regardless of the key pressed. This code segment is called by a key catcher which is triggered by the `<next>` key.

V-7.5.11 Modifying the Appearance of a Form

Screen fields, enumerations and text can be given the following display attributes:

- inverse video
- underline
- half-bright
- blinking
- blank fill
- text color (color terminal only)
- font index (graphics terminal only)
- concealment (contents of a field are not displayed).

Any of these attributes may be changed, or reset to their initial values.

Calls Used:

`Form_Handler.Change_display_attributes`
Changes the display attributes of a screen field, enumeration, or text.

`Form_Handler.Restore_display_attributes`
Restores the display attributes of a screen field, enumeration, or text.

Display attributes include:

- inverse video
- underlining
- half-bright
- blinking
- blank fill
- font index
- concealing (not displaying) the contents of a field.

An example use of these two calls is to blink a field's contents to alert the user that the entered data is erroneous and must be reentered. The field is restored to its original attributes after the user enters a valid value.

V-7.5.12 Inquiring About an Element, Form Sheet, and Form Status

An application can more effectively control the execution of a form when it is able to access the identity and current state of form elements. The availability of information about the form sheet during execution ensures that device-dependent considerations stay transparent to the user. Form status and other current state information gives a valuable snapshot of the executing form. All this information is made available to the application by the following calls.

Calls Used:

`Form_Handler.Get_current_number_of_group_instances`
Gets the current number of instances of a group.

`Form_Handler.Get_current_path_element`
Gets the name and type of the path element currently being executed.

`Form_Handler.Get_current_subunit`
Gets the pathname of the current subunit.

`Form_Handler.Get_element_info`
Returns information about an element.

`Form_Handler.Get_index_sequence_of_current_subunit`
Gets the index of each subunit (group instances and subforms) comprising the current subunit.

`Form_Handler.Get_last_edited_sheet_element`
Gets the pathname of the last edited screen field or enumeration.

`Form_Handler.Last_input_event`
Gets information about the last input event.

`Form_Handler.Get_selected_sheet_element`
Gets information about the sheet element selected by the last mouse event.

`Form_Handler.Get_sheet_info`
Returns information about the currently displayed form sheet.

`Form_Handler.Get_status_info`
Returns information about the status of a form.

Get_element_info returns the type of the element and whether the contents of the element have been changed since the last initialization.

Get_sheet_info returns size information about the currently displayed form sheet and whether it is designed to be displayed on a character or graphics terminal.

Get_selected_sheet_element returns similar information as Get_element_info except that the element is selected by a mouse event. This call is used by key catchers which catch mouse events.

This information is typically evaluated by the application program when the execution of a form is suspended or aborted.

V-7.5.13 Inquiring About the Last Edited Sheet Element and Input Event

These calls return information identifying a previous action.

Calls Used:

Form_Handler.Get_last_edited_sheet_element
Provides the form network pathname of the screen field or enumeration last edited.

Form_Handler.Get_last_input_event
Returns the type of the last input event.

Get_last_edited_sheet_element returns the form network pathname of the last screen field or enumeration that was edited. This call is commonly used by processing routines to aid in determining which successor to choose as the next path element.

Get_last_input_event returns the type of the last input event. This call is used similarly to get_last_edited_sheet_element.

V-7.6 Summary

- Form_Handler enables an application to dynamically control a form.
- After a form has been created, it can be executed and controlled by Form_Handler calls which perform the following functions:
  - Open and close a form
  - Execute a form
  - Insert and store data
  - Modify the order of execution
  - Add and remove group instances
  - Modify the appearance of the form
  - Inquire about the state of the form.
- Developing an executable form involves:

- Designing a form
- Generating a form description
- Creating and binding a message file
- Writing an application to execute the form
- Writing processing routines, key catchers and key lists, as needed
- Testing the form with the application.

• The following steps comprise a recommended procedure for accomplishing these tasks:

  Step 1 - Design a Form Layout.

  Step 2 - Create the Form.

  Step 3 - Test the Form.

  Step 4 - Create a Message File.

  Step 5 - Bind the Message File.

  Step 6 - Write an Application Program.

  Step 7 - Write Subroutines, Translation Tables and Key Lists.

  Step 8 - Test the Form with the Application.

  Step 9 - Create a Window.

• A screen field can be set to accept a CL variable as input.

• The following utilities are provided to automatically create a simple, standard form, to test a form and to map symbolic keys to specific terminals:

  - `test.form`
  - `create.form`
  - Translation tables editor.

GENERATING REPORTS 8

Contents

V-8.1 Concepts

This chapter discusses the ways to create and modify a report description and print a report.

Packages Used:

`Report_Handler`
Provides calls for initializing and printing a report.

A *report* is a printed or displayed document containing labelled data, often presented in hierarchical groups with subtotals and totals. A simple report is shown in Figure V-8-1.

```
                    INVENTORY REPORT

Part ID    Description               Location    Unit

1234567    wiring harness             13-B27     each
3512734    1/2" aluminum conduit      02-F12     feet
4766117    5/16" hex carriage bolt    07-A02     lb
7689482    flexible control cable     06-C13     inch
```

Figure V-8-1. Sample Report

V-8.1.1 Report Characteristics

A report is made up of various combinations of the following report parts:

- Report heading
- Report footing
- Page heading
- Page footing
- Control group footing
- Control group heading
- Record print layout.

A typical report consists of one or more pages of data, a report heading, and a report footing (see Figure V-8-2).

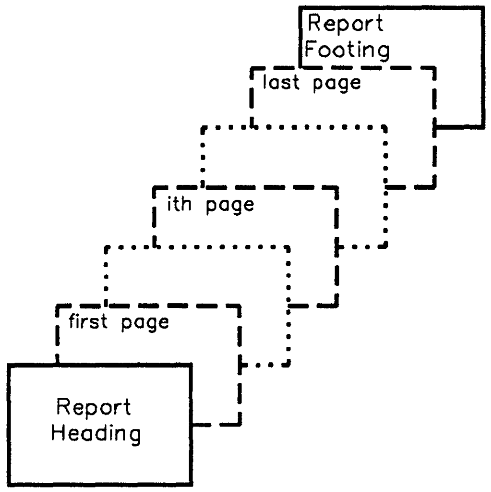


Figure V-8-2. Page Series of a Report

The *report heading* prints on a separate page and may contain explanatory information similar to the title of a book or the burst page of a print job. The *report footing* may print on the last page or a separate page and can contain summary statistical information pertaining to the report. Both are optional.

Data appears on the report pages other than the report heading and footing pages. The layout of a page is defined by the *page body area* shown in Figure V-8-3.

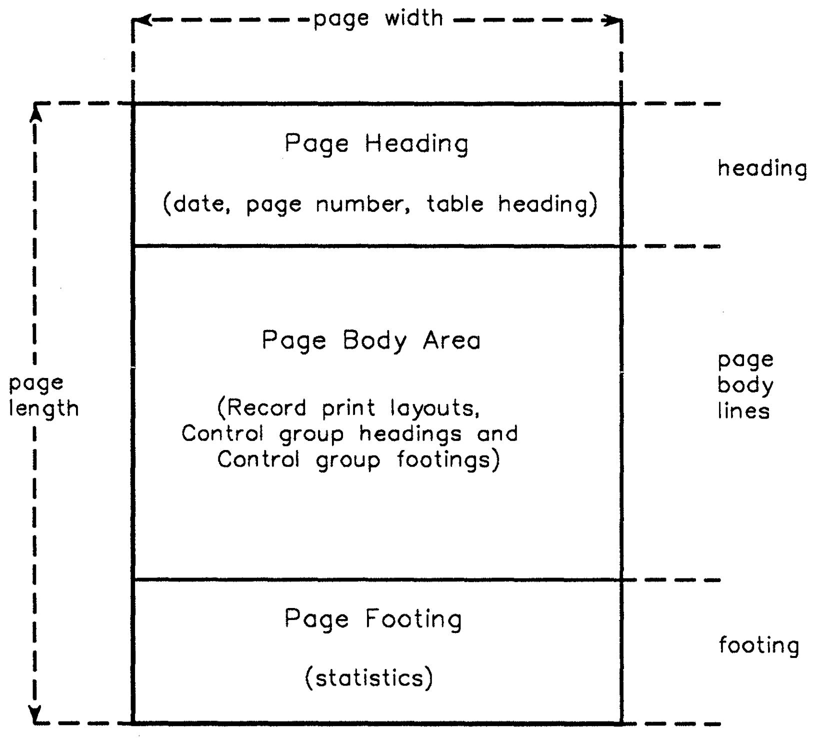

page width
Page Heading
(date, page number, table heading)
heading
Page Body Area
(Record print layouts, Control group headings and Control group footings)
page body lines
page length
Page Footing
(statistics)
footing

Figure V-8-3. Parts of a Report Page

A *page header* typically contains the date, page number, and headings for the columns of data. The *page footer* typically contains statistics or is empty. Both are optional.

The *record print layout* defines how the records are to be printed. This layout includes information about:

- the record fields selected for printing
- additional, explanatory information
- user-defined expressions
- position, display attributes, and formatting of the record fields.

The *page body area* contains one or more kinds of items defined by the record print layout:

- A *data* corresponds to a field of the record from which the report is derived. It contains a reference to this field and all information about the layout of the data. When the detail is printed, the content of the corresponding record field is printed.
- *Computed data* contains a mathematical expression. The expression may contain references to data in the same report part or in another report part. It contains all information about the layout of the data. When the report is printed, the expression is calculated and the result of the calculation is printed.
- A *text* represents an explanatory text string.

The data in the page body area may be an unstructured stream of records or may be structured in groups of records framed by intermediate headings and footings called *control groups*. The records within control groups have in common a particular field which contains the same value.

Figure V-8-4 illustrates control groups in three consecutive pages of a report. Two control groups are defined. Control group 2 is nested within control group 1. The dominant/subdominant relationship of the control groups defines the *control hierarchy.*

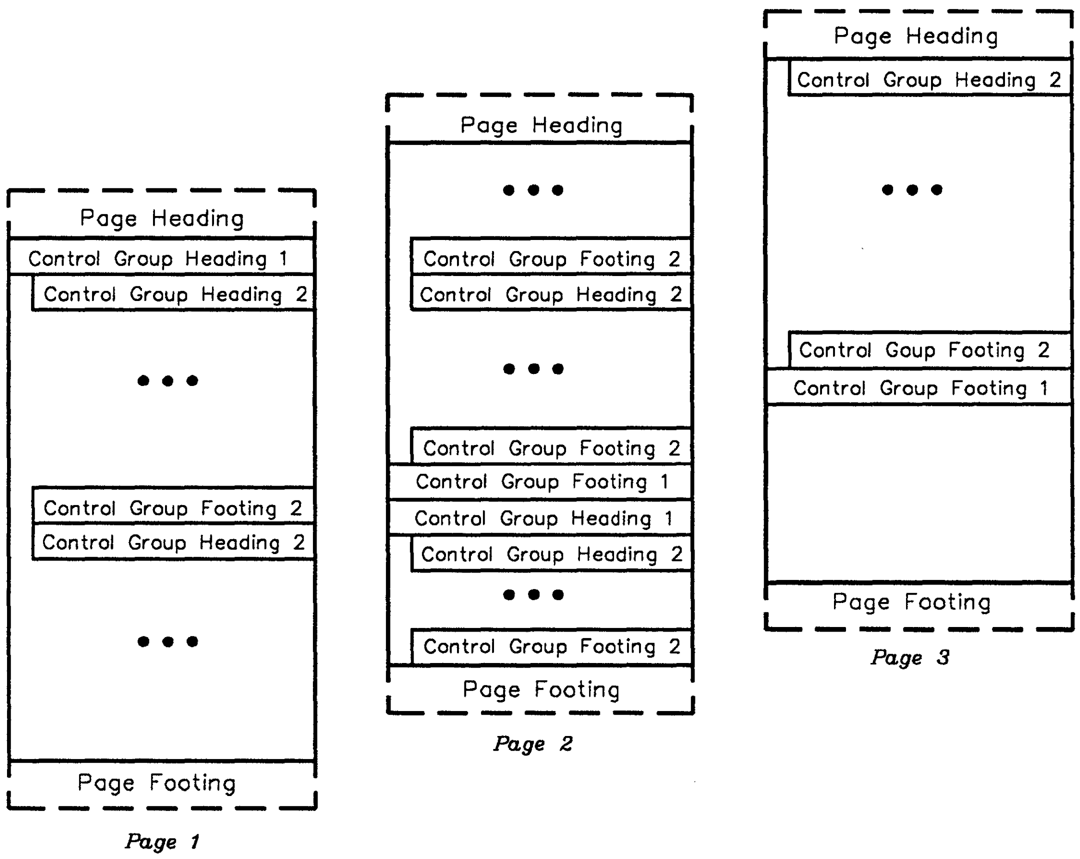


Figure V-8-4. Report With Nested Control Groups

V-8.1.2 Control Groups

Records printed in the page body area can be grouped into *control groups.* If a stream of records contain groups of records having at least one field with the same value, this collection of records can be printed as a group. The field with the common value can be designated as a *control group field.*

Control groups may be nested. When nested, control groups define a hierarchical structure called the *control group hierarchy* that controls the sequence of printing the records.

Each time a record of the file is read, the contents of all control fields are evaluated. The change of the value of a record field designated as a control field causes a *control break.* On a control break, printing is suspended until the following actions have been performed:

- All control group footings are printed beginning from the lowest level of control group hierarchy up to the level associated with the highest level of the control field which caused the control break.

- All control group headings are printed beginning from the level associated with the highest level of the control field which caused a control break down to the lowest level.

Figure V-8-5 shows a report with control breaks on the `location` field of the Parts Master File record of the Inventory Program example. The `Cost` column is defined as computed data which is the product of `qty_on_hand` (not reported) and `ave_unit_cost` (not reported).

```
Date: 12/31/87   Inventory Location Report      Page:  1

Location   Part ID   Description            Unit    Cost

02-F12     3512734   1/2" aluminum conduit  feet    121.98
           3571998   5/8" aluminum conduit  feet    317.69
           3521195   3/4" aluminum conduit  feet     79.50

                                          Total:    519.17

07-A02     4766117   5/16" hex  bolt        lb       17.69
           4619984   3/8" stove bolt        lb       37.55
           4722390   1/2" crenellated nut   lb        7.05

                                          Total:     62.26

                                    Grand Total:    581.43
```

Figure V-8-5. Report With Control Breaks

V-8.1.3 Representation of Report Descriptions

A report description is composed of report parts. Figure V-8-6 shows an example of the report parts that are combined in a report description.

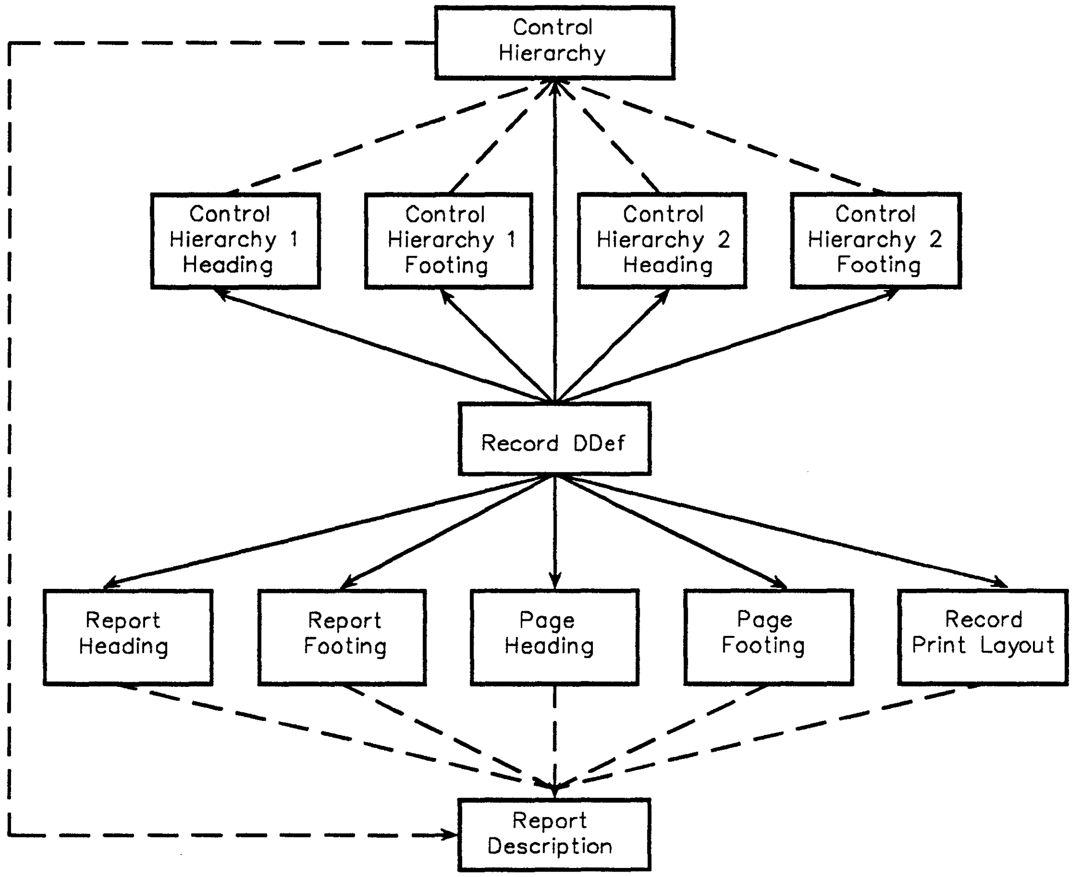


Figure V-8-6. Report Parts of a Report Description

The report service requires a record DDef that describes the data to be printed. The record DDef can be created using `Data_Definition_Mgt`, or an existing record DDef for a file can be used.

V-8.1.4 Creating and Modifying a Report Description

Three methods are available for creating and modifying report descriptions: interactively with `edit.report`, dynamically with `create.report`, and procedurally using `Data_Definition_Mgt`. The report editor, `edit.report`, is the most commonly used method. `create.report` is the easiest method for generating a simple report. `Data_Definition_Mgt` is the most fundamental and complex method and is primarily a tool for utility writers.

Application programmers will normally use `edit.report` to create a report description. Report descriptions can also be created procedurally using `Data_Definition_Mgt`, although this method requires a detailed understanding of DDefs. This low-level procedural interface is mainly of interest to implementors of utilities such as `edit.report`.

The report editor, `edit.report`, is an interactive utility for creating and modifying report descriptions. Upon successful completion of a report design or update, the report editor generates a report description that can be used to print the report. See the *BiiN™ Systems Reports Guide* for detailed information on report editor.

`create.report` automatically creates the most simple, default report design based upon the description of an associated data record. See the *BiiN™ Systems Reports Guide* for instructions on using this editor. The layout of a standard report page is shown in Figure V-8-7.

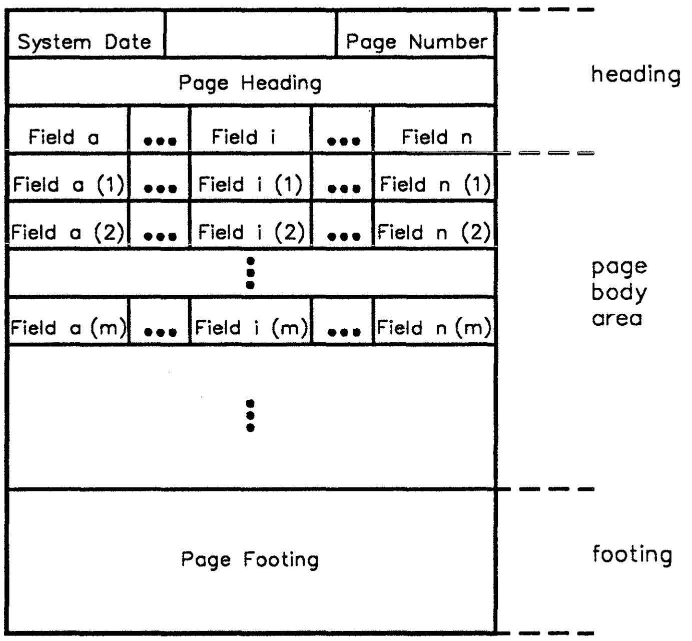


Figure V-8-7. Layout of a Standard Report Page

The report parts for a standard report assume the following default properties:

Record print layout The data of the record print layout is taken from the corresponding fields of the record.

Within the page body area, the data is printed line by line (according to the records read) and positioned beneath the matching column.

The width of a column is determined by the length of the name of the field in the heading and by the length of the field (by the format string for numeric fields), whichever is larger. The smaller one is centered within the column.

Default formats for numeric and date data are shown in Table 4-1.

Table V-8-1. Standard Report Default Formats

Default Format	Type
-zzzzzzzzz9	int4
-zzzzzzzzzzzzzzzzzz9	int8
-9.9999999999E-99	real8
yyyy-mm-dd	date

Numeric fields are right-justified; byte string fields are left-justified.

Control group hierarchy
When the report is associated with a variant record, control hierarchies are defined by the standard layout for readability.

If the record description does not contain variant parts, no control group hierarchy is defined.

Control group headings	No control group heading is defined.
Control group footings	No control group footing is defined.
Page heading	The page heading prints the date on the left and the current page number on the right. A tabular heading line is printed in the third line of the heading. For variant records, the tabular heading line reflects the contents of the first record to print on any given page.
Page footing	The page footing is defined as a single, empty line.
Report heading and footing	The report heading and footing are not defined.

V-8.1.5 Report CL Variables

The following lists contains the names, descriptions, types, and initial values of the CL (Command Language) variables used by the report service. CL variables affect the appearance and performance of the report editor and report handler. See the *BiiN™ Systems Reports Guide* for instructions on the use of CL variables. See V-1 for a general discussion of CL variables.

The *scope* of a variable is defined as *E* or *D*. *E* means that the variable is valid throughout the editor session or is only relevant in the initialization phase. *D* means that the variable provides default values for editor adjustments which may be changed during an editor session.

Report Editor-Specific CL Variables

`report.editor_key_map`
Symbolic name of the translation table that is used by the report handler to translate incoming characters into symbolic keys. If null, a standard, internal translation table is used.

```
        Scope:  E
         Type:  string
Initial Value:  null string
```

`report.window_position_line`
Line number of the upper-left position of the Info window. The upper line of the Main window depends on the CL variable `report.info_window_lines` and the upper line of the Message window is likewise dependent on the values of `report.info_window_lines` plus `report.main_window_lines`.

```
        Scope:  E
         Type:  integer
Initial Value:  1
```

`report.window_position_column`
Column number of the upper-left position of the Info window.

```
        Scope:  E
         Type:  integer
Initial Value:  1
```

`report.window_columns`
Width in columns of the three editing windows.

```
        Scope:  E
         Type:  integer
Initial Value:  80
```

`report.info_window_lines`
Number of lines in the Info window.

```
           Scope:  E
            Type:  integer
   Initial Value:  10
```

`report.main_window_lines`
Number of lines in the Main window.

```
           Scope:  E
            Type:  integer
   Initial Value:  10
```

`report.message_window_lines`
Number of lines in the Message window.

```
           Scope:  D
            Type:  integer
   Initial Value:  1
```

`report.pop_up_message_window`
Determines whether the Message window will open and close upon the receipt of a message or stay open. If true, a Message window is opened each time a message is to be displayed, and closed when input is entered into any of the editor windows.

```
           Scope:  D
            Type:  boolean
   Initial Value:  false
```

`report.editor_adjustments`
Symbolic name of the *report editor adjustments* object. This object contains adjustments that affect the appearance and operation of the form editor. Adjustments may be made and saved with the form editor. If this string is null, default adjustments are used.

```
           Scope:  E
            Type:  string
   Initial Value:  null string
```

General CL Variables Used by the Report Editor

`form.decimal_character`
Character which will be displayed and accepted as the decimal symbol. The possible values are `Form_Defs.point` and `Form_Defs.comma`.

```
           Scope:  D
            Type:  string
   Initial Value:  "." (point)
```

`form.escape_character`
The character which is used as the escape symbol in a format string. See the *BiiN™ Systems Reports Guide* for information concerning formatting screen fields.

```
           Scope:  D
            Type:  string
   Initial Value:  \ (backslash)
```

`form.visual_bell`
Defines whether the editor user will be informed visually or audibly of incorrect input. If true, the signal is visual, else the signal is audible.

```
           Scope:  D
            Type:  boolean
   Initial Value:  false (audible)
```

`user.verbose`
Indicates whether status messages should be displayed.

```
          Scope:  D
           Type:  boolean
  Initial Value:  false (not displayed)
```

`user.language`
Defines whether the editor user will be informed visually or audibly of incorrect input. If true, the signal is visual, else the signal is audible.

```
          Scope:  D
           Type:  string
  Initial Value:  null string
```

`msg.long_text`
Used by the message service to determine whether the long or short version of a message is to be used.

```
          Scope:  D
           Type:  boolean
  Initial Value:  false (short)
```

V-8.1.6 Printing a Report From the Command Line

`print.file` is a general purpose utility with which reports can be printed or displayed. It reads the input file and writes the report to a spool queue. See the *BiiN™ Systems Administrator's Guide* for more information about this utility.

V-8.2 Techniques

After reading this section, you will be able to:

- Print a report from your program
- Optionally sort a file and print the sorted entries
- Change global assignments.

The examples used are excerpted from the `Inventory_Reports_Ex` example listed in Appendix X-A.

V-8.2.1 Printing a Report From Your Program

Calls Used:

`Report_Handler.Initialize`
Initializes a report for printing.

`Report_Handler.Print`
Prints an initialized report.

Initialization associates a report description with an input device opened for record stream input, and an output device opened for character display output to which the report is printed. `Report_Handler.Print` prints an initialized report.

The entire input stream is printed in input order; that is, by record number for relative files or by index for indexed files. The input file may be the entire, original file associated with the report description or a subset of this file.

If the file is an indexed file, a subset of the original file can be selected with `Record_AM.Keyed_Ops` calls. If the report control hierarchy fields differ from the key fields of which the file index is composed, then `Sort_Merge_Interface.Sort` can be called to generate a record stream with the required record order.

The following sample code demonstrates the use of `Report_Handler.Print` in which a range of records is printed in indexed order.

```
  local_parts_file:  Device_Defs.device :=
      Record_AM.Ops.Get_device_object(
          Inventory_Files.parts_file);
   -- AD to parts file.

  opened_local_parts_file:
      Device_Defs.opened_device;
    -- AD to locally opened parts file.

  part:  System_Defs.text(4) := (4,4,"part");
    -- Parameter to "report_printing" message,
    -- since this report is by "part".

begin

  -- Open parts file for reading, so no
  -- concurrent updates will interfere:
  --
  opened_local_parts_file := Record_AM.Ops.Open(
      dev          => local_parts_file,
      input_output => Device_Defs.input,
      allow        => Device_Defs.readers);


  -- Open output device:
  --
  opened_output := Byte_Stream_AM.Open_by_name(
      name         =>
          output_dev_pathname,
      input_output =>
          Device_Defs.output);


  -- Get report definition (DDef):
  --
  report_DDef := DDef_from_untyped(
     Directory_Mgt.Retrieve(
         name => report_by_part_DDef_pathname));
  -- Assume "Report_Handler.Is_report".


  -- Initialize report:
  --
  initialized_report := Report_Handler.Initialize(
      description => report_DDef,
      input       => opened_local_parts_file,
      output      => opened_output);


  -- Print report:
  --
  Report_Handler.Print(
      report => initialized_report);


```

```
-- Display "report_printing" message:
--
Message_Services.Write_msg(
    msg_id => report_printing_code,
    param1 => Incident_Defs.message_parameter(
        typ => Incident_Defs.txt,
        len => part.length)'(
            typ     => Incident_Defs.txt,
            len     => part.length,
            txt_val => part),
    param2 => Incident_Defs.message_parameter(
        typ => Incident_Defs.txt,
        len => output_dev_pathname.length)'(
            typ    => Incident_Defs.txt,
            len    => output_dev_pathname.length,
            txt_val => output_dev_pathname),
    device => Inventory_Windows.message_window);


-- Close locally opened parts file:
--
Record_AM.Ops.Close(
    opened_dev => opened_local_parts_file);
```

The report service also implements record I/O as another method for printing reports. This method enables printing reports from applications written in languages such as COBOL that provide record I/O but do not support ADs. Using record I/O to print a report is similar to writing to a file. The application program opens a device specifying the report description. Each `Insert` call supplies a record to the report service. The report is sent to the application program's current output device; that is, the *standard output* specified in the process globals.

The report service allows report descriptions for files which contain variant records.

V-8.2.2 Setting Global Assignments

Calls Used:

`Report_Handler.Set_global_assigns`
Assigns the error handling controls for an initialized report.

Several global properties may be set by report editor. Two of these, *error decision* and *line end decision*, may be changed with `Set_global_assigns`.

The *error decision* defines the action to be taken when a numeric error (overflow, underflow, or division by zero) occurs during the evaluation of an arithmetic expression. Possible actions include:

- Printing the error symbol (default is ?) instead of the erroneous value
- Suspending the evaluation of the current item, and continuing printing with the next item
- Terminating the report (closing output and returning).

The *line end decision* defines the action to be taken when the width of the mounted sheet is too small for printing the report lines. Possible actions include:

- Printing the remaining characters on the next line

- Discarding the remaining characters
- Terminating the report.

V-8.3 Summary

The report service and related utilities provide methods for creating and modifying a report description and for printing a report.

- A report is a printed or displayed document containing labelled data, often presented in hierarchical groups with subtotals and totals.
- A report description is composed of report parts.
- Methods for creating and modifying report descriptions include the report editor, `create.report` and the `Data_Definition_Mgt` procedural interface.
- Methods for printing or displaying a report include `Report_Handler.Print`, `print.file` and record I/O.
- `Report_Handler` includes calls to associate a report description with an input and output device, print an initialized report and control error handling.