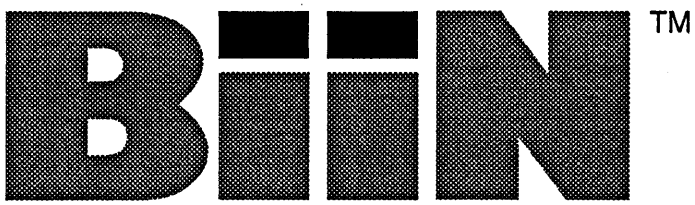
BiiN™ SYSTEMS OVERVIEW

BiiN™



BiiN™ SYSTEMS OVERVIEW

Order Code: 6AN9000-1AJ00-0BA2

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

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

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

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

PREFACE

The *BiiN™ Systems Overview* describes major features and benefits of BiiN™ computer systems.

Release Restrictions

Preliminary BiiN™ system releases do not provide some features described in this manual. See the product release notes for more information.

Organization

The *BiiN™ Systems Overview* contains seven short chapters:

- Chapter 1 describes the system's goals.
- Chapters 2 and 3 describe the hardware and software.
- Chapters 4 through 7 describe how the system achieves its major goals.

Related Publications

To begin using BiiN™ systems, see *Getting Started with BiiN™ Systems.*

To learn about programming BiiN™ systems, see the *BiiN™ Systems Programmer's Guide.*

To learn about the BiiN™ Operating System, see the *BiiN™/OS Guide.*

CONTENTS

Chapter 1. System Goals

1.1 Support for Software Development 1-1
1.2 Scalable Performance 1-1
1.3 Fault Tolerance 1-2
1.4 Data Integrity 1-2
1.5 Distributed Computing 1-3
1.6 Support for Industry Standards 1-3

Chapter 2. Hardware

2.1 Series 60 Systems 2-1
2.2 VLSI Technology 2-2
2.3 Central Processing Unit 2-3
2.4 System Bus 2-3
2.5 RAM Memory 2-4
2.6 I/O Subsystems 2-4

Chapter 3. Software

3.1 Command Language Executive 3-1
3.2 Utilities 3-2
3.3 Programming Languages 3-2
3.4 Other Programming Tools 3-3
3.4.1 Emacs Text Editor 3-3
3.4.2 Linker and Librarian 3-3
3.4.3 Debugger 3-3
3.4.4 Software Management System 3-4
3.5 Database Management System 3-4
3.6 Forms 3-4
3.7 Reports 3-4
3.8 BiiN™/UX 3-4
3.9 Graphics Support 3-5
3.10 The BiiN™ Operating System 3-5

Chapter 4. Scalable Performance

4.1 Enhancing System Performance 4-1

4.1.1 Processor Performance 4-1
4.1.2 Memory Access Time 4-1
4.1.3 Bus Performance 4-2
4.1.4 Stable Store 4-2
4.1.5 I/O Performance 4-2
4.2 Transparent Multiprocessing 4-2

Chapter 5. Fault Tolerance

5.1 Levels of Fault Tolerance 5-1
5.2 Basic System Reliability 5-1
5.2.1 Processor Reliability 5-1
5.2.2 System Bus Reliability 5-1
5.2.3 Memory Reliability 5-2
5.2.4 Disk Reliability 5-2
5.3 Fault Checking Systems 5-2
5.4 Fault Recovery for Continuous Operation 5-3
5.5 Complete Fault Coverage 5-3
5.6 Fast Troubleshooting and Online Repair 5-4

Chapter 6. Data Integrity

6.1 The Need for Protection within a Running Program 6-1
6.2 Object-Based Protection 6-1
6.2.1 What is an Object? 6-1
6.2.2 How are Objects Referenced? 6-2
6.2.3 The Inside and Outside Views of An Object 6-3
6.2.4 Switching Address Spaces Within a Program 6-3
6.2.5 Three-Fold Protection 6-4
6.3 Protecting Objects Stored on Disk 6-5
6.4 Ensuring Data Consistency with Transactions 6-6

Chapter 7. Distributed Computing

7.1 Developing Distributed Applications 7-1
7.2 How a Distributed Service Works 7-1
7.3 Distributed Disk Storage 7-2
7.3.1 General Characteristics of Passive Store 7-3
7.3.2 Naming and Reference for Passive Objects 7-3
7.3.3 Efficient Access to Passive Store 7-4
7.3.4 Handling Concurrent Access and Ensuring Consistency of Passive Objects . 7-5
7.4 Distributed Program Execution 7-6
7.5 Distributed System Administration 7-6
7.6 Support for Multiple Protocols 7-7
7.7 Flexible Network Topologies 7-7
7.8 Connecting to Non-BiiN™ Systems 7-7

List of Figures

2-1. BiiN™ Hardware Architecture 2-1
3-1. BiiN™ Software 3-1
4-1. Transparent Multiprocessing 4-3
5-1. Fault-Checking Computational Subsystem 5-3
6-1. AD and Object 6-2
6-2. Linear Address Space and Domain 6-4
6-3. Three-Fold Object Protection 6-5
7-1. Communication and Cooperation Between Filing Service Instances 7-2
7-2. Passive Store is a Distributed Object Filing Service that Unifies All Nodes in a BiiN™ System. 7-3
7-3. Single Activation Access to a Passive Object 7-5
7-4. Multiple Activation Access to A Passive Object 7-5

List of Tables

2-1. Central Processing Unit Features 2-3

1 SYSTEM GOALS

This chapter introduces the BiiN™ system's goals:

- Support for software development
- Scalable performance
- Fault tolerance
- Data integrity
- Distributed computing
- Support for industry standards.

1.1 Support for Software Development

BiiN™ systems can run many existing applications and can be used to develop new applications:

- Several widely-used and standardized programming languages are available: Ada, C, COBOL, FORTRAN, Pascal, and SQL.
- BiiN™/UX, an implementation of UNIX System V, is provided.
- BiiN™ supports object-oriented programming methods, believed to increase software productivity, reliability, and maintainability.
- The BiiN™ Operating System provides many advanced features that do not have to be implemented by application writers.
- BiiN™ applications can normally run on systems with different levels of performance, fault tolerance, or distribution.

Chapter 3 provides more information about the BiiN™ software development environment.

1.2 Scalable Performance

BiiN™ systems can be used for a wide range of applications, with different and changing performance requirements:

- The BiiN™ computer family contains a range of systems to match different user requirements. Users don't have to pay for performance that is not needed.
- The maximum performance available in BiiN™ systems is great enough for almost all applications.
- BiiN™ systems can be expanded by adding modules, preserving existing hardware investments.

- Processing power, memory, and I/O can all be expanded independently.
- There is complete software compatibility between different BiiN™ systems.

Chapter 4 provides more information about BiiN™ systems' performance.

1.3 Fault Tolerance

BiiN™ systems can be used for applications that demand extremely high reliability or even continuous operation. Some BiiN™ systems are *fault tolerant* and contain duplicated hardware units used to detect faults and possibly recover from them as well. BiiN™ systems support three levels of fault tolerance:

- Basic BiiN™ systems are very reliable and detect many faults. For example, extra check bits are built into BiiN™ memory modules to correct or detect memory errors.
- Fault-checking BiiN™ systems detect any fault and keep faults from harming data or programs.
- Continuous BiiN™ systems detect any fault and recover from any single fault by immediately switching to backup modules.

The BiiN™ approach to fault tolerance has these advantages:

- The BiiN™ computer family provides a range of fault tolerance to match different user requirements. Users don't pay for unneeded capabilities.
- There is complete software compatibility between systems with different levels of fault tolerance. Software changes are not needed to take advantage of increasing levels of fault tolerance.
- Fault-checking and continuous operation are implemented in hardware, so that such systems run almost as fast as standard systems.

Chapter 5 provides more information about BiiN™ fault tolerant systems.

1.4 Data Integrity

BiiN™ systems can be used for applications that must protect valuable data from hardware errors, software errors, or unauthorized access:

- Error Correction Codes and other checks detect and prevent errors in semiconductor or disk memory.
- Access to data representations can be restricted to particular software modules.
- Software modules can be protected so that an error in one module doesn't overwrite data in another module.
- Related changes to data can be grouped so that either all changes succeed or all changes are automatically undone.
- Access to data can be restricted to exactly those individuals with a "need to know."
- Protection boundaries are flexible and controlled by application designers.

Chapter 6 provides more information about how BiiN™ systems ensure data integrity.

1.5 Distributed Computing

A *distributed* computer system is a network of computers that can be used as if it is a single computer. BiiN™ systems can be distributed:

- In a distributed BiiN™ system, all files, applications, and other resources are transparently accessible at every computer in the network.
- BiiN™ applications are automatically distributed without special implementation effort, because operating system services are distributed.
- There is complete software compatibility between distributed BiiN™ systems and stand-alone BiiN™ computers.

Chapter 7 provides more information about distributed computing.

1.6 Support for Industry Standards

BiiN™ systems are easier to use and easier to connect with peripherals and networks because they support many industry standards:

- Programming language standards
- UNIX operating system standards
- Network communication standards
- Electrical safety and emission standards.

Chapters 2 and 3 describe what standards are supported by BiiN™ hardware and software.

HARDWARE 2

This chapter describes BiiN™ hardware systems and their major components. Figure 2-1 illustrates the hardware architecture. A BiiN™ computer uses a high-speed system bus to interconnect all modules. Multiple CPUs, multiple I/O subsystems, and multiple memory boards can be included in a single system.

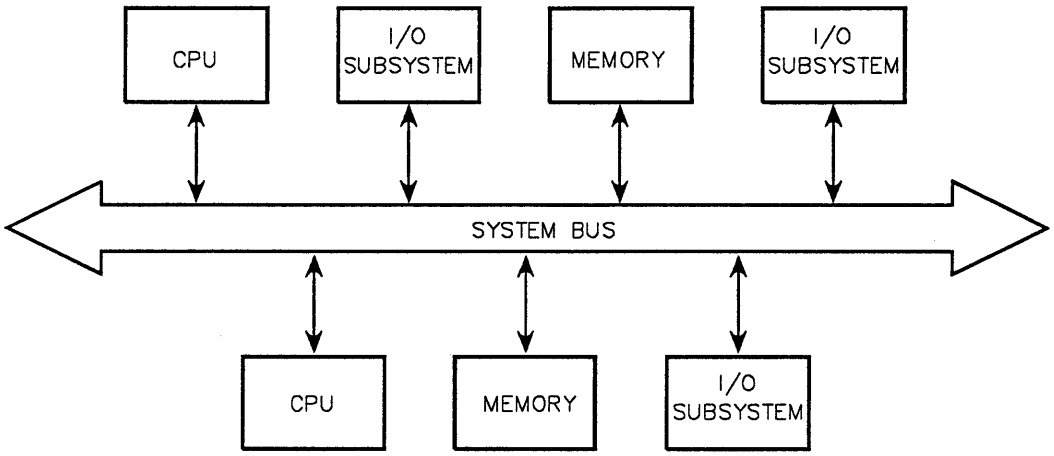


Figure 2-1. BiiN™ Hardware Architecture

Several features distinguish BiiN™ systems:

- The hardware uses Very Large Scale Integrated circuits (VLSI chips) to increase performance and reliability while reducing space requirements and power consumption.
- The hardware directly supports key software functions, such as memory management, protection, and fast file access.
- The hardware supports industry standards for connecting disk drives, tape drives, networks, terminals, printers, and other devices to systems.
- The hardware is modular, allowing subsystems to be mixed and matched to provide a range of performance and of fault tolerance.

2.1 Series 60 Systems

The BiiN™ Series 60 System provides a wide range of hardware performance. The Series 60 contains 12 available bus slots and room for four full-height and two half-height mass storage devices. The Series 60 can be configured with:

1-8 processors

32-80 Mbytes of RAM memory, if no more than five slots are used for memory

2-10 I/O controllers, if no more than five slots are used for I/O

16-128 serial lines

Up to four 380M byte Winchester disk drives (1.5 Gbytes total)

Any two half-height peripherals: 1.6Mbyte 5.25" diskette, 100Mbyte Winchester hard disk, 125Mbyte cartridge tape.

2.2 VLSI Technology

BiiN™ systems are designed around four Very Large Scale Integrated circuits (VLSI chips):

CPU	Central Processing Unit. The CPU is a 32-bit processor with on-chip floating-point arithmetic, memory management, and OS support.
CP	Channel Processor. The CP is an I/O processor with up to eight independent concurrent tasks and up to 10M bytes/second data transfer.
BXU	Bus Exchange Unit. The BXU provides an interface between a CPU or CP's Local Bus and the System Bus. Each BXU also controls up to 64K bytes of fast cache memory.
MCU	Memory Control Unit. The MCU handles a memory board's System Bus interface, controls up to 8M bytes of DRAM, and handles memory Error Correction Code (ECC).

All these VLSI chips do the work of hundreds of smaller components in older computer systems. Compared to many older computer systems, BiiN™'s VLSI solutions are:

- Much more reliable
- Faster
- Smaller, requiring much less board space
- Economical to operate, requiring much less power and cooling.

Also, all BiiN™ boards and modules are designed with surface-mount technology, further increasing reliability and reducing system size.

2.3 Central Processing Unit

Table 2-1 summarizes the BiiN™ CPU's features.

Table 2-1. Central Processing Unit Features

Feature	Description
Instruction Processing Rate	5.5 MIPS (1 MIPS = VAX 11/780)
Register Set	16 one-word local registers per call, 3 used for linkage and 13 general-purpose. 20 global registers per process, 4 floating-point, 1 frame pointer and 15 general-purpose word registers. The CPU can cache four sets of local registers.
Addressing and Memory Management	2^{58} byte virtual address space, consisting of up to 2^{26} objects each containing up to 2^{32} bytes. Physical memory is organized as 4Kbyte pages, and paged virtual memory is supported. Object addressing and virtual address translation are provided on the chip.
Protection	Each access to memory is automatically checked to see that a valid pointer is used, that the pointer has required access rights, and that the access falls entirely within the bounds of a valid object. All objects are typed. Program modules can check that an object has the proper type with a single instruction.
Arithmetic	
Integer	8-bit, 16-bit, and 32-bit signed and unsigned integer arithmetic are supported. Signed arithmetic faults on overflow; unsigned arithmetic does not.
Floating-point	32-bit, 64-bit, and 80-bit floating-point arithmetic are provided on chip. Floating-point arithmetic conforms to the IEEE standard for binary floating-point arithmetic. There is on-chip support for exponential, logarithmic, and trigonometric functions.
Multiprocessing Support	Available processors run the highest-priority waiting processes automatically, without software intervention. Low-level process scheduling is done directly by the processors. Single instructions are provided to lock and unlock semaphores and to send and receive interprocess messages. Process preemption is supported.
Subprogram Call Support	A single-instruction call can switch to a different address space for a called subprogram. These different address spaces or *domains* can be used to give each software module access to just those objects for which it needs access.
Interrupt Latency	
Typical	5 µs
Maximum	10 µs

The CPU achieves its high execution speed by using concepts found in reduced-instruction-set-computer (RISC) designs:

- Several functional units can do different tasks in parallel, such as adding two numbers while decoding the next instruction while fetching the instruction after that.
- Many simple instructions can be executed in a single cycle.
- A large register set reduces the number of memory accesses.

The CPU also excels in areas neglected by many microprocessor architectures, such as floating-point arithmetic, memory management, and process management.

2.4 System Bus

The BiiN™ System Bus is a 32-bit bus designed to support multiple processors, multiple memory modules, and multiple I/O controllers.

Each Series 60 system uses two parallel system buses, increasing system performance.

The system bus interface is handled by a VLSI chip, the BXU (Bus Exchange Unit). Each BXU also controls up to 64K bytes of fast cache memory.

2.5 RAM Memory

A BiiN™ memory board can contain 8M or 16M bytes, using 1Mbit DRAM chips. When 4Mbit DRAM chips are available, a single memory board will be able to provide 64M bytes.

The RAM memory is made very reliable by its use of an Error Correction Code (ECC) for every memory word. The ECC mechanism is able to *correct* any single-bit error without loss of data, and *detect* any double-bit error.

A memory board's control logic and bus interface is implemented by a single VLSI Memory Control Unit (MCU).

2.6 I/O Subsystems

BiiN™ I/O subsystems provide interfaces for mass storage, communications, and standard I/O peripherals, such as printers and terminals.

BiiN™ systems support two standard buses for connecting mass storage devices:

- ANSI byte-wide medium performance Small Computer Systems Interface (SCSI)
- ANSI double-byte-wide high performance Intelligent Peripheral Interface (IPI).

BiiN™ systems support two standards for connecting systems to Local Area Networks (LANs):

- IEEE 802.2/3
- Ethernet Blue Book.

BiiN™ systems support several standards for serial communication:

- High-level Data Link Control (HDLC) balanced and unbalanced bit-oriented synchronous protocols
- Synchronous Data Link Control (SDLC) unbalanced bit-oriented synchronous protocols
- 7-bit and 8-bit asynchronous protocols.

BiiN™ systems support several different physical layer interfaces that can be used for serial communication:

- Full Modem Interface (EIA Standard RS-232-C and CCITT Recommendations V.21, V.22, V.22bis, V.23, V.25bis, V.26, V.26bis, V.26ter, V.27, V.27bis, V.27ter, V.29, and V.32)
- Limited Modem Interface (EIA Standard RS-232-C and CCITT Recommendations V.21, V.22, V.22bis, V.25bis, V.26bis, and V.26ter)
- Digital Interface (EIA Standard RS-422 and CCITT Recommendations X.21 and X.24)
- TTY current loop interface.

SOFTWARE 3

This chapter describes BiiN™ software products, shown in Figure 3-1.

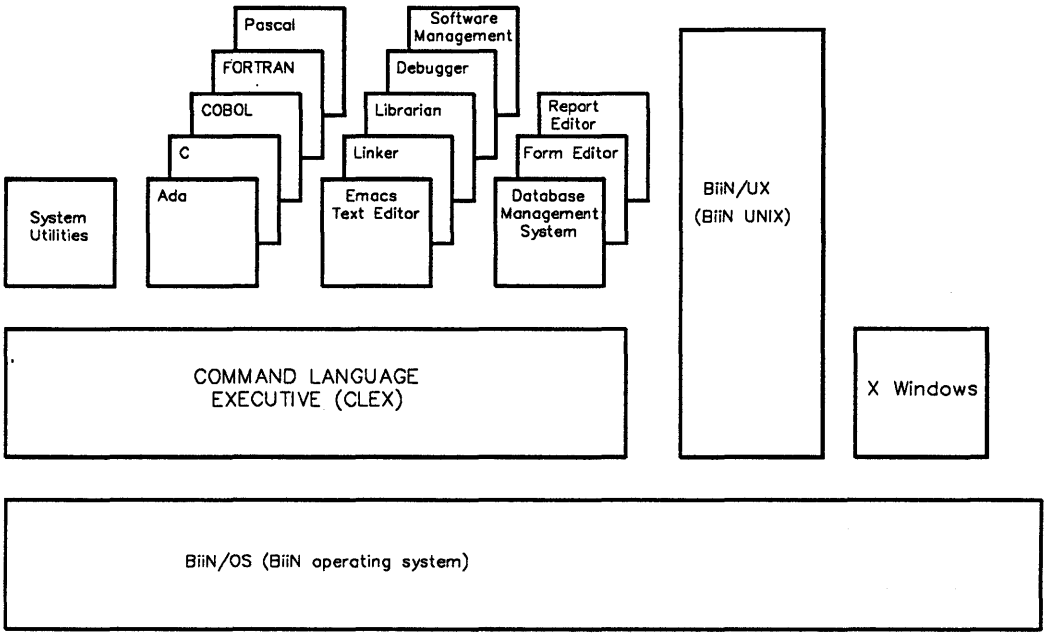


Figure 3-1. BiiN™ Software

Software products are presented in the order that a user might encounter them, beginning with the command language and system utilities.

3.1 Command Language Executive

The BiiN™ Command Language Executive (CLEX) is a general-purpose command interpreter with these major features:

- Syntax checking and command error recovery are done by CLEX, unburdening the applications programmer.
- CLEX automatically prompts for missing mandatory arguments.
- CLEX allows abbreviation of command names and queries the user to resolve ambiguous selections.
- CLEX's built-in help facility provides information about what commands are available, the syntax of any command, or the meaning of any command argument.

- If your system has BiiN™/UX (BiiN™'s implementation of UNIX), then CLEX accepts BiiN™/UX commands directly, without needing to invoke a BiiN™/UX shell.
- CLEX provides UNIX-like piping, I/O redirection, and command language control structures.
- CLEX supports multiple jobs concurrently executing in multiple windows.
- CLEX provides "runtime" commands that are automatically part of the command set of any utility that uses CLEX. For example, the `set.current_directory` command is available within any such utility.
- CLEX can be used as a programming language. CLEX scripts can be stored in files and invoked as commands.

3.2 Utilities

The BiiN™ system provides many utilities (interactive commands) for users and for system administrators. User utilities are provided for:

- Managing directories, files, and protection information
- Controlling jobs
- Printing
- Grouping a series of commands within a transaction
- Managing CLEX variables.

System administrator utilities are provided for:

- System configuration
- Managing disks and volume sets
- Managing spool queues
- Monitoring system operations
- Controlling and accounting for system resources
- Managing user accounts
- Backing up or restoring system data
- Managing terminals, printers, and other devices.

3.3 Programming Languages

Six standard programming languages are available for the BiiN™ system:

Ada — An implementation of the Ada programming language as specified by ANSI/MIL-STD-1815A-1983.

C — An implementation of the draft ANSI standard, X3J11/86-151, that is also compatible with AT&T System V C. Two function libraries are available:

- A UNIX-derived libary that includes UNIX-compatible system calls, for customers moving UNIX applications to the BiiN™ system.

- A native function library that is not derived from UNIX and does not include UNIX-compatible system calls.

COBOL	An implementation of the COBOL 85 standard.
FORTRAN	An implementation of the FORTRAN 77 standard, with extensions specified by MIL-STD-1753.
Pascal	An implementation of the ANSI/IEEE 770 X.397-1983 standard.
SQL	An implementation of the industry-standard *structured query language* for accessing relational database management systems. SQL is provided with the BiiN™ database management system.

Ada, C, COBOL, FORTRAN, and Pascal conform to the IEEE 754-1985 standard for binary floating-point arithmetic.

3.4 Other Programming Tools

Besides programming languages, the BiiN™ system provides many other tools needed in a complete and productive programming environment.

3.4.1 Emacs Text Editor

BiiN™ Emacs is a full implementation of this popular editor. Emacs is a multi-window screen editor that can be extended using command macros and a built-in version of the MLisp programming language. Programming support includes automatic indenting and parenthesis checking for C and Pascal programs.

3.4.2 Linker and Librarian

The BiiN™ Systems Linker is a multi-language builder of runnable programs. Modules written in any BiiN™ language can be linked together. Developers can use the linker to group program modules in separate *domains*, to take advantage of the system's interdomain protection features.

The BiiN™ Systems Librarian is a programming tool for managing *libraries* of compiled program modules. Libraries are useful because all modules in a library can be made available during linking by specifying a single library name. The linker only links in those library modules actually referenced by a program being linked.

3.4.3 Debugger

The BiiN™ Application Debugger is a high-level debugging tool with these major capabilities:

- Debugging programs at the source-code level, using program-defined names.
- Debugging programs at the machine level, using memory addresses and register names.
- Debugging programs that use multiple processes.
- Logging debugging sessions to files.

3.4.4 Software Management System

The BiiN™ Software Management System (SMS) is a tool for managing software development projects. SMS includes a version control system, configuration management system, project tracking tools, and other useful tools.

3.5 Database Management System

Information about the BiiN™ database management system (DBMS) will be provided in a future release of this manual. The selected DBMS is expected to support SQL, the industry-standard Structured Query Language.

3.6 Forms

The Form Editor is used to lay out forms used for interactive data entry. Features of BiiN™ forms include:

- Conditional fields, only used if a particular condition is true.
- Group fields, for entering or displaying repeating data elements.
- Protected fields, output-only fields that cannot be modified. Such fields can be computed from other fields during program execution.
- Subforms, forms nested within other forms.
- Key catchers, subprogram calls that customize the handling of particular keys pressed by the user.
- Processing routines, subprogram calls triggered by user entry of particular fields. Processing routines can be used to do additional input validation. For example, if a source account ID field and a destination account ID field must have different values, then a processing routine can check for the difference.

A procedural interface to forms is used to display a form in a window and then read entered data as a record stream.

3.7 Reports

The Report Editor is used to lay out reports generated by your application. Features of BiiN™ reports include control of page headings and footings, control breaks, and column sums. The procedural interface to reports is used to write a stream of records to a report.

3.8 BiiN™/UX

BiiN™/UX is a software product that provides a user interface and many utilities derived from AT&T's UNIX System V operating system. BiiN™/UX includes:

- Two command interpreters, or *shells* (`sh` and `csh`), each of which can also be used as a programming language

- File and directory utilities
- The `awk` and `sed` text processing tools and a variety of text editors
- A program builder (`make`) and a source-code control system (SCCS)
- Many other utilities designed for the software engineer.

Applications written for UNIX are often written in C and rely on a set of UNIX *system calls.* A UNIX-derived function library that includes the system calls is available as a separate C function library. Some applications also depend on shell scripts, common commands in those scripts, or `make` scripts. All these parts are provided: system calls (in the C function library), the shell programs, common commands, and the `make` utility. The BiiN™/UX documentation provides a special manual, *Porting UNIX System V Applications to BiiN™ Systems*, to help customers move existing System V applications to the BiiN™ system.

3.9 Graphics Support

The BiiN™ X Window System (*X*) supports graphics applications. X is a portable standard for developing windowing applications, supported by virtually all vendors of graphics workstations. X is designed to be device-independent, portable across operating systems, and stable yet extensible. Unlike traditional windowing systems, X is designed to work well in a networked or distributed environment. Both Ada and C interfaces to X are provided.

BiiN™ systems support both character terminals and graphics terminals. Graphics applications can only run on graphics terminals. All BiiN™ software products and all character-oriented applications can run on either type of terminal.

3.10 The BiiN™ Operating System

The BiiN™ OS is a multiuser, multitasking operating system that supports:

- Multiprocessing, in which multiple CPUs simultaneously execute within a single computer.
- Flexible protection of data and programs, in which only those users or modules with a "need to know" can access a particular entity.
- Transaction processing, in which related changes to data can be grouped so that either all changes succeed or all are automatically undone.
- Distributed computing, in which multiple interconnected computers behave like a single computer system.
- Concurrent programming, in which multiple processes within a program execute concurrently.
- Real-time programming, in which resources are preallocated and programs must respond quickly to interrupts and other asynchronous events.
- Record-oriented applications, which use record-structured files and indexes for storage, forms for data entry, and reports for data output.

Subsequent chapters in this book describe some of these features in more detail.

SCALABLE PERFORMANCE 4

This chapter shows how BiiN™ systems are designed for high performance and to provide a range of performance.

4.1 Enhancing System Performance

This section describes how the BiiN™ system design enhances system performance. System features that enhance performance include:

- A high-performance 32-bit processor
- A high-speed System Bus
- Extensive caching to reduce the time needed to access both main memory and disk
- Use of separate Channel Processors to handle I/O.

4.1.1 Processor Performance

The BiiN™ system maximizes individual processor performance in several ways:

- The CPU provides on-chip floating-point arithmetic and memory management, avoiding the overhead of communicating with separate coprocessors.
- The CPU uses reduced instruction set computer (RISC) concepts to boost performance, including extensive pipelining, single-cycle execution of many instructions, simplified instruction formats, and a large register set. "Register scoreboarding" identifies registers that can be used in an instruction before a previous instruction has finished executing.
- The CPU provides special instructions and microcode sequences for very fast subprogram calling, interrupt handling, process switching, and interprocess communication.

See the *BiiN™ Systems CPU Architecture Reference Manual* for more information about the CPU.

4.1.2 Memory Access Time

Extensive *caching* is used to reduce the time needed to read or write data. Caching keeps frequently referenced data within or "close" to the CPU, to reduce access time:

- Each CPU contains a 512 byte instruction cache and a stack-frame cache. The stack-frame cache holds up to four sets of local registers.
- Each BXU manages 32K bytes of fast cache memory per CPU. Many memory reads "hit" in the cache, avoiding the extra time needed to access memory via the System Bus. When writing data, the CPU can continue execution while the BXU writes the data through to memory via the System Bus.

4.1.3 Bus Performance

The BiiN™ System Bus uses several techniques to enhance performance:

- Up to 16 bytes can be read or written as a single bus operation, reducing the number of bus operations and increasing bus bandwidth.
- Other bus activity is allowed in the time between a bus request and the corresponding reply. Up to three bus operations can be active at the same time.
- Because many memory accesses are handled by cache hits, CPUs issue fewer bus requests and use less bus bandwidth. This allows more CPUs to run in one system without saturating the bus.
- A Series 60 system contains two parallel buses connected in a crossbar arrangement. Bus traffic tends to be evenly distributed over the two buses, alleviating bus bottlenecks.

See the *BiiN™ Hardware Subsystems Reference Manual* for more information about the System Bus.

4.1.4 Stable Store

Stable store is RAM memory used to cache disk blocks. A battery backup ensures that stable store contents are maintained even if system power is lost. Stable store greatly reduces the time needed for disk access:

- Blocks being read from disk are often already in stable store, avoiding a disk access.
- Blocks being written to disk are simply written to stable store, avoiding the delay of waiting for a disk access. Writing blocks to disk is done later and asynchronously by the OS.
- If an application uses small and short-lived files, those files may be created, used, and then deleted entirely within stable store, without any disk accesses. For example, the transaction service uses such short-lived files to keep information about active transactions.

4.1.5 I/O Performance

A BiiN™ system's I/O is handled by separate *Channel Processors* (CPs). The CP is a programmable I/O processor that can transfer data at up to 10M bytes/second. The CP can handle up to eight independent I/O tasks simultaneously, supporting several different peripherals. The BiiN™ approach to I/O performance has these advantages:

- The CPs free up CPU cycles and increase overall throughput. In a system with four CPUs and four CPs, there may really be eight processors running simultaneously instead of four.
- I/O performance can be scaled independently from CPU performance, by adding additional I/O boards.

4.2 Transparent Multiprocessing

A BiiN™ computer can contain from one to eight CPUs. Each processor can execute a different process in the computer's shared memory. With one processor, processes take turns executing on the single processor. With several processors, several processes can execute with true concurrency. The only difference is in execution speed.

BiiN™ provides *transparent multiprocessing*: No software changes are required if the number of processors changes. All processes and processors share a common *dispatch port*, a meeting place for processes and processors. All ready processes waiting to run are placed in the common queue. When a processor needs a process, it simply takes the process at the head of the common queue and begins to execute it.

Figure 4-1 shows two processes running (A and B) and two waiting at the dispatch port (C and D) in a two-CPU system. If process A blocks, perhaps waiting for an I/O operation to complete, then it releases its CPU. The CPU would then go to the dispatch port and "dispatch" the first waiting process, C. The dispatching action removes C from the list of waiting processes, binds it to a CPU, and begins execution. Later, when process A is again ready to run, it will be added to the list of processes waiting at the dispatch port.

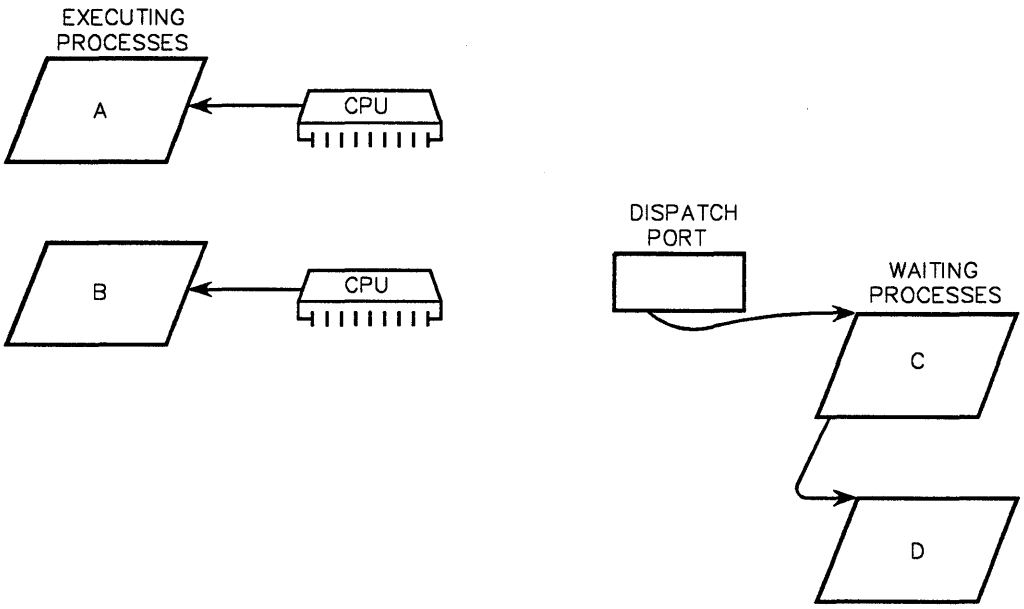


Figure 4-1. Transparent Multiprocessing

Transparent multiprocessing has these advantages:

- Programs don't depend on the number of processors. Software is independent of the multiprocessing hardware.
- Writing parallel programs is straightforward, using well-understood techniques for managing multiple processes.
- The individual processors are general-purpose computers, and each can execute a different instruction stream simultaneously. This makes a transparent multiprocessor very flexible and able to speed up the execution of almost any kind of parallel problem.

BiiN™'s implementation of transparent multiprocessing handles all process dispatching and low-level process scheduling in the CPU microcode; no software intervention is required.

FAULT TOLERANCE 5

Fault tolerant computer systems can detect errors in their own operation and automatically recover from errors whenever possible. BiiN™ systems support a range of fault tolerance options, including systems for continuous computing. Such systems include redundant hardware resources that can detect any single hardware failure and recover from such failures without interrupting system operation.

5.1 Levels of Fault Tolerance

BiiN™ systems can provide any of three levels of fault tolerance:

basic	Relies on the system's built-in fault detection and handling mechanisms, but with no duplication of hardware resources.
fault checking	Duplicates VLSI chips—CPU, CP, BXU, MCU—to provide comprehensive fault detection for those chips and their associated subsystems. If an unrecoverable fault is detected, the affected subsystem is shut down so that permanent data is not damaged.
continuous	Duplicates almost all hardware subsystems. For VLSI chips, fault-checking pairs are duplicated. Each hardware subsystem has a backup that is immediately available if it fails. Any single hardware fault automatically reconfigures the system and continues execution without a system restart.

5.2 Basic System Reliability

A fault tolerant system must have highly reliable building blocks. Each BiiN™ subsystem has been designed for high reliability.

5.2.1 Processor Reliability

These features enhance CPU reliability:

- Each CPU tests itself when it is initialized.
- CPU instructions do extensive error checking; any random error in the CPU itself is likely to trigger instruction errors that will stop program execution.

5.2.2 System Bus Reliability

These features ensure System Bus reliability:

- Each bus has two parity lines that detect any single-bit error. Parity is checked for each bus cycle. If an error occurs, the erring bus operation is retried once before reporting a permanent error.

- Because a Series 60 system has two System Buses, the system can continue to run without interruption if one bus fails. If a bus fails, the system automatically reconfigures itself to use just the one remaining bus.

5.2.3 Memory Reliability

These features ensure RAM memory reliability:

- A BiiN™'s RAM memory uses an Error Correction Code (ECC) for every memory word. The ECC mechanism is able to *correct* any single-bit error without loss of data, and *detect* any double-bit error.
- If a memory chip fails, a "spare bit" memory chip for that memory array can be switched in to replace the bad chip.

5.2.4 Disk Reliability

These optional features enhance disk reliability:

- *End-to-end checking* stores a check-code with each block of data written to disk. When an I/O unit reads a disk block, it computes a new check-code and compares it to the retrieved code value. If the retrieved and computed values do not match, then an error is signaled to the operating system.
- *Mirroring* replicates a volume set (a logical mass storage device) on different disks. Each block written to the volume set is written to both disks. Blocks are read from only one disk, the primary disk. If data is lost on the primary disk, the lost data can still be recovered from the backup "shadow" disk.

5.3 Fault Checking Systems

Fault checking detects faults in hardware modules by pairing modules and continuously comparing the output of the paired modules. The two paired physical modules then function as one "logical" module. For example, two CPU modules, each consisting of CPU, BXUs, and cache, can be paired into a single logical CPU (Figure 5-1). All inputs are routed to both CPUs while their outputs are compared by the BXUs. Any discrepancy in outputs raises a fault and shuts down the entire logical CPU.

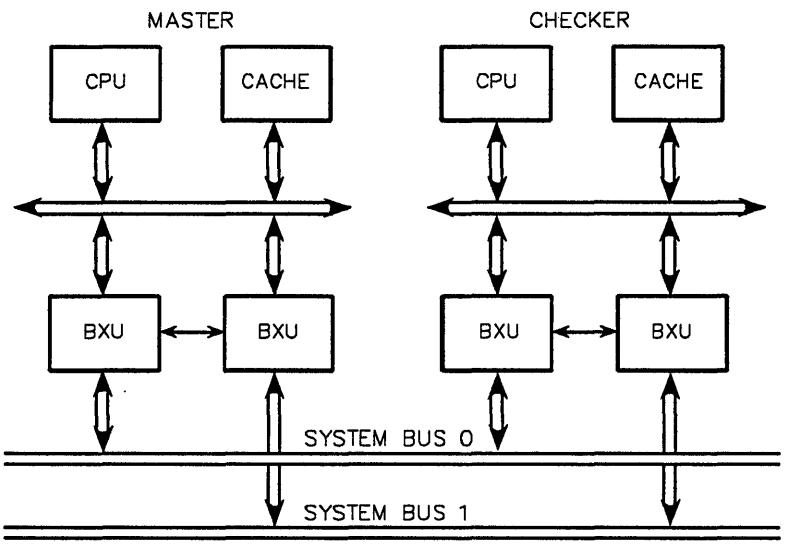


Figure 5-1. Fault-Checking Computational Subsystem

Fault checking can be used to detect faults in computational subsystems (CPUs and BXUs), I/O subsystems (CPs and BXUs), and memory subsystems (MCUs). In a fault-checking I/O subsystem, only the CPs and BXUs are paired and not the device controllers and devices. End-to-end checking can be used to improve fault coverage for some devices. In a fault checking memory subsystem, only the MCUs are paired and not the RAM memory array. Fault detection for the RAM array is already provided via Error Correcting Codes.

5.4 Fault Recovery for Continuous Operation

Quad Modular Redundancy (QMR) enables continuous operation of a properly configured Series 80 system by pairing fault-checking modules. If one fault-checking module fails, then the other can take its place. Module replacement is done automatically by the system without interfering with running applications. Replacement is typically accomplished within 100 microseconds of a fault.

QMR can be used for computational subsystems (CPUs and BXUs), I/O subsystems (CPs and BXUs), and memory subsystems (MCUs and RAM). A QMRed memory subsystem uses two fault-checking memory subsystems which each control one RAM array, so there is a primary and shadow RAM array.

5.5 Complete Fault Coverage

The fault-tolerant capabilities of the BiiN™ system extend to the system monitor EPROM chips, clocks, system support modules, power supplies, and fans. All of these subsystems can also be twinned and configured to automatically disable faulty modules and switch to backups if faults occur.

5.6 Fast Troubleshooting and Online Repair

The BiiN™ system comes with a comprehensive suite of hardware diagnostics for identifying hardware problems. These features help reduce downtime:

- System boards and modules can be replaced online while a system is running and without interrupting system execution.
- System diagnostics can be executed remotely, by a service technician at another site connected by modem to the system being tested.

DATA INTEGRITY 6

This chapter explains how BiiN™ systems protect your data from human errors, software errors, and unauthorized access. Chapter 5 explains how BiiN™ systems protect your data from hardware failures.

6.1 The Need for Protection within a Running Program

A modern application program can contain hundreds of software modules. By design, each module handles a particular task or deals with one particular type of data. In fact, a faulty module that makes an error in addressing its data—and such errors are common—can corrupt or destroy *any* data in the program, even if the destroyed data is managed by another module. Such bugs have often occurred in even well-tested, widely-distributed, commercial computer programs, forcing revisions or recalls.

Addressing errors are pernicious because the module that causes an error may be entirely unrelated to the module that detects the error, if the error is detected at all. Worse, what data gets corrupted may depend on where data is allocated in memory each time the program runs.

6.2 Object-Based Protection

BiiN™ systems use a uniform protection scheme that is secure, flexible, efficient, and enforced by the hardware.

All data and programs in a BiiN™ system are stored in *objects*. This section introduces objects and the advantages of an object-oriented architecture.

6.2.1 What is an Object?

An object is a typed, protected segment of memory.

You are probably already familiar with several object types. *Files*, *directories*, and *programs* are among the types supported by the system.

Each object can contain from 0 bytes to 4 Gigabytes. The contents of an object are called its *representation.*

Objects can be dynamically created, resized, and destroyed.

The number of objects in your system is essentially unlimited. A particular BiiN™ computer can contain up to 2^{26} (more than 64 million) objects in its active memory at any time. Object types are themselves objects and applications can create new types as needed.

6.2.2 How are Objects Referenced?

An *access descriptor (AD)* is a protected pointer to an object. The only way to reference an object is via an AD.

In most computer systems, a pointer is simply an arbitrary bit pattern used as an address. Such pointers can be corrupted without detection by the hardware or OS. ADs are specially tagged memory words that can only be created or modified in carefully controlled ways. Changing an AD in an unauthorized way invalidates the AD.

An AD contains both addressing information, used to find the object in memory, and also *access rights*, that indicate what operations are possible with the AD (Figure 6-1).

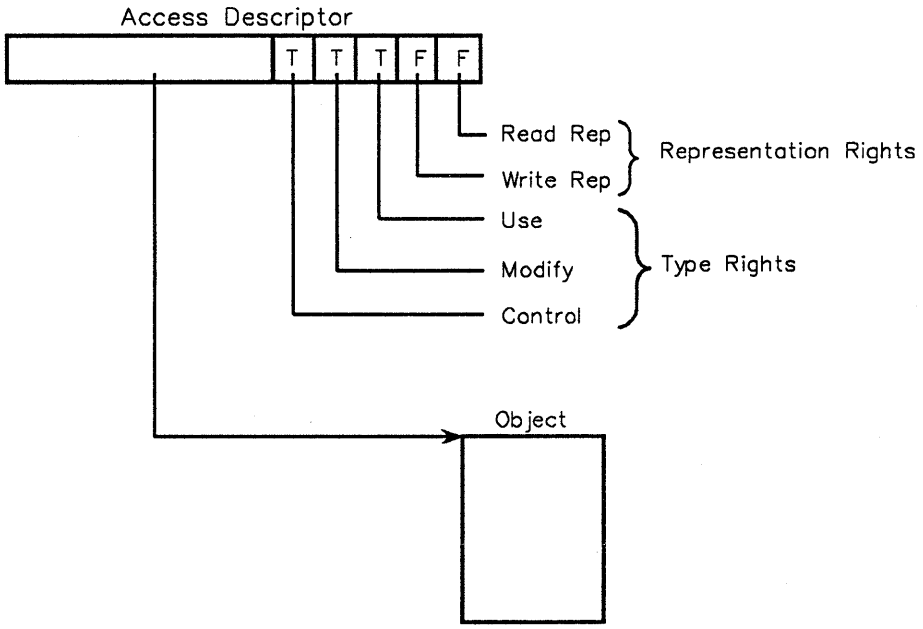


Figure 6-1. AD and Object

There are five access rights stored in each AD:

Read rep rights	Required to read an object's representation. This right is checked and enforced by the CPU on every read access.
Write rep rights	Required to write an object's representation. This right is checked and enforced by the CPU on every write access.
Three *type rights*	Required for type-specific operations, such as list rights for listing a directory. These rights can be defined differently and renamed for each type of object. These rights are checked and enforced by software.

Different users or programs may have ADs with different rights to the same object. Mary may have an AD with all rights to a directory and John may have an AD with only list rights.

To reference a particular field within an object, a program can use a two-part *virtual address*: a 32-bit offset to a byte within the object plus an AD to the object.

6.2.3 The Inside and Outside Views of An Object

A key concept in object-based protection is a strong distinction between *using* a type of object and *implementing* a type of object. For example, almost every program or subsystem running on a BiiN™ system uses directories. However just one module in the operating system implements directories. Only that module, the *type manager* for directories, needs the *inside view* of directories, with rep rights to access their internal representation. Every other module or program that uses directories only needs the *outside view*, and thus holds directory ADs with appropriate type rights but no rep rights. The outside view of directories is opaque; a directory is a black box and you cannot see inside. A directory or any other object is defined entirely by the operations provided by its type manager.

Hiding implementation details from object users has two very good effects:

- The implementation of an object type can be completely changed. So long as the same operations are provided with the same behavior, you are guaranteed that no using programs will fail, because it is not possible for any using program to know or depend on the implementation.
- It is not possible for any program module outside the type manager to corrupt the representation of a directory. In a program written as a collection of type manager modules, no module can corrupt data managed by any other module.

When another module calls an object's type manager to perform some operation, it supplies an AD to the object, an AD without rep rights. The type manager and only the type manager has a special "key" that allows it to turn on rep rights on ADs to objects that it manages. The type manager uses the key to turn on rep rights and then performs the requested operation.

6.2.4 Switching Address Spaces Within a Program

A BiiN™ program can be partitioned into multiple protected modules, each with its own *linear address space* or *domain*. See Figure 6-2. A linear address space contains up to 2^{32} bytes mapped onto four objects by the GDP: static data, instructions, stack, and a special object used only by the OS. Each domain provides access to a particular collection of objects that can be reached from the objects mapped by its linear address space. Only those program modules with a "need to know" about a particular object have access to it, and then only have the access rights that they need. For example, each type manager can be placed in its own domain.

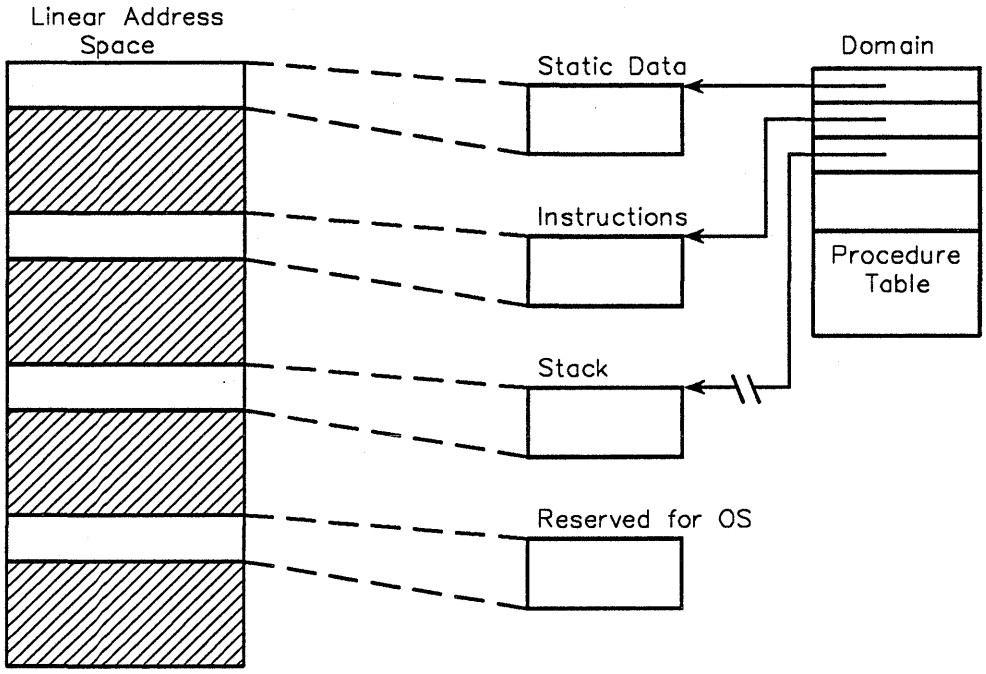


Figure 6-2. Linear Address Space and Domain

When one domain calls a routine in another domain, switching address spaces is done by the CPU, as part of the call.

BiiN™ programmers can choose either a one-domain (linear) or multiple-domain (structured) organization for their programs:

- A program that does not use object-based protection can be compiled entirely into one domain. Because there is a single linear address space for the entire program, linear addresses can be used for pointers.
- A program that uses object-based protection can be compiled into multiple domains. Because linear addresses are only valid within a particular domain, ADs or virtual addresses are normally used for pointers.

The organization of modules into domains can be varied to trade greater protection for greater execution speed. For example, related type managers can be grouped into the same domain.

6.2.5 Three-Fold Protection

Each object in a BiiN™ system is protected in three ways, as shown in Figure 6-3:

- Limited access: Only those modules with a "need-to-know" can reference the object.
- Type checking: If an object's type is not the proper type required by an operation, then the operation fails.
- Right checking: If the AD used does not have rights that allow the operation, then the operation fails.

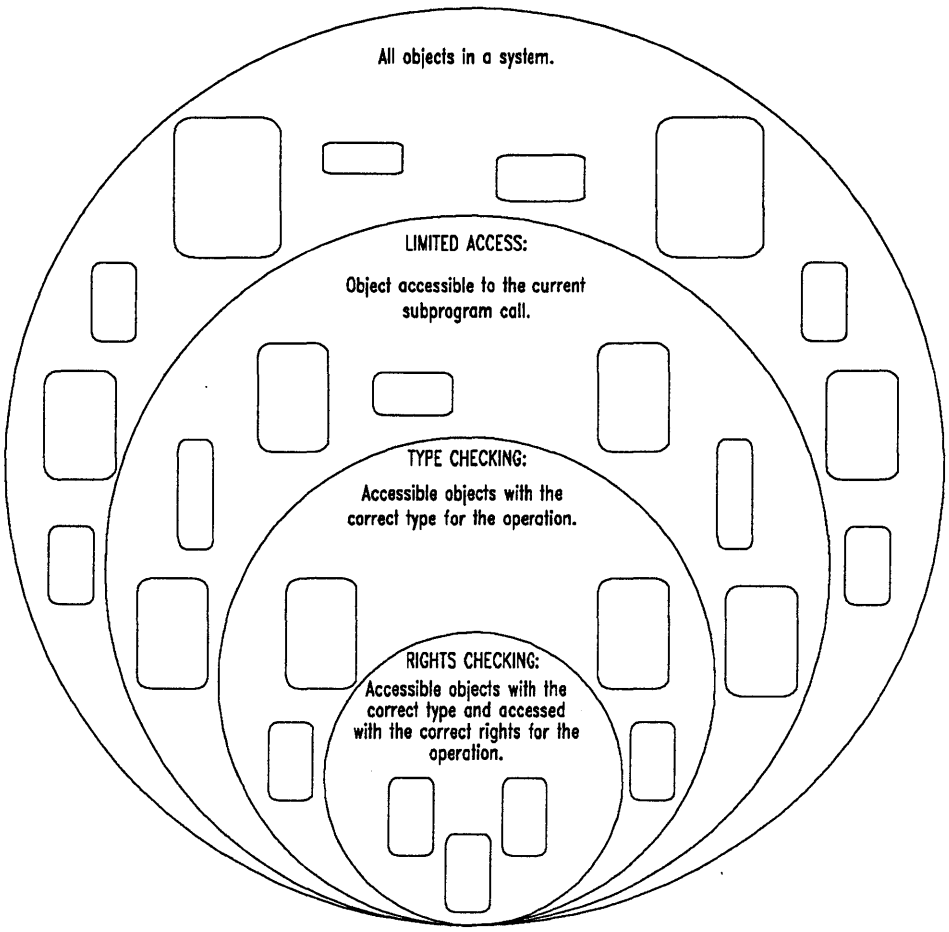


Figure 6-3. Three-Fold Object Protection

6.3 Protecting Objects Stored on Disk

BiiN™ systems are unique in extending "need-to-know" protection to all objects stored on disk, as well as objects in active memory. However, the mechanism for protecting objects on disk is somewhat different, for good reasons:

- Within a running program, intermodule protection is achieved by giving different protected modules different address spaces, each containing only those objects that the module has a need to reference. On disk, it is both impractical and undesirable to give every user a different "address space" or different directory structure. In fact, there is a system-wide hierarchical directory structure shared by all users.
- Within a running program, once an AD is handed out to a module, the AD cannot be reclaimed; access can't be revoked. This is normally not a problem because a particular program run is usually brief. ADs and objects on disk are long-lived; it is quite possible that access to an object may be granted to a user for some time and then need to be revoked.

Both these problems are solved by BiiN™'s *authority-based protection* of objects on disk. An object on disk can reference an *authority list* that describes exactly who can access the object

and with what rights. Each entry in an authority list is an <ID, type rights> pair. An ID represents either a user or a class of users, such as all users in a particular department. A user or a running program has an associated *ID list* containing several IDs, like a key ring with several keys. *Authority list evaluation* compares an ID list and authority list to find matching IDs and compute the resulting type rights.

Authority-based protection solves both problems described above:

- Access is restricted via authority lists rather than by using multiple address spaces.
- Access to an object can be revoked for a user or group by simply changing its authority list.

6.4 Ensuring Data Consistency with Transactions

Consider a banking program that is transferring funds from your checking account to your savings account. The program subtracts $1,000 from your checking account. Then the system crashes, before the money is added to your savings account. The bank's accounts are now inconsistent and funds have been lost. *Transactions* are a mechanism to prevent such inconsistency. A transaction groups a related series of operations so that either all the operations succeed or all are aborted and undone. For example, enclosing a funds transfer in a transaction ensures that either all the accounts involved are updated or that all the changes are rolled back.

Transactions are provided by the *transaction service* within the OS. Some notable characteristics of transactions are:

- Transactions are normally used to maintain the consistency of objects stored on disk, and not used for objects in active memory.
- The operations within a transaction appear to happen atomically—all together. Partial results are not visible to other transactions.
- Transactions protect against all possible reasons that a sequence of operations does not complete, including system crashes, I/O errors, and program exceptions.
- Transactions can be distributed, involving objects at many nodes in a distributed system.
- Transactions can be nested, with subtransactions within transactions.
- Timeouts can be associated with transactions, ensuring that a transaction is aborted if it does not complete within a certain time.
- Transactions can involve many different object types, not just files.
- Most OS services that manage objects on disk are transaction-oriented, including passive store, the filing service, and the directory service.
- Developers can create new transaction-oriented type managers.

7 DISTRIBUTED COMPUTING

Connecting computers with communications lines—*networking*—is becoming commonplace. Networked computers can talk with each other, but they do so at "arm's length." Networks can be difficult to use in several ways:

- Special commands must be used to transfer files or start a computing session on another system.
- Both people and programs using a network must often know *where* in the network desired data or resources are located. If data or resources are moved, then programs and procedures fail and must be updated. This is burdensome in a network of a dozen systems and is a major problem in a larger network.
- Administering a computer network can have special problems. For example, if the utility to add a user is not designed for the networked environment, then adding a new user to 50 computers in a network may require 50 separate actions by the system administrator.

Distributed computing is to networking as day is to night. A distributed computer system contains many computing *nodes*, spread out in space and networked together, but the entire collection of nodes appears to users and programs as a single geographically distributed computer system. In a distributed computer system, you don't need to know what node your program is running at, what node your file is stored at, or what the paths are between the nodes that you use. Users and programs can use the same operating procedures regardless of whether they are using a single node or are spread out across multiple nodes. The goal of distributed computing is to make the network *transparent* to users, so that networking details become invisible and unimportant.

7.1 Developing Distributed Applications

Your BiiN™ applications are automatically distributed, without any special implementation effort, because the operating system services that they use are distributed. Files, program execution, I/O devices, and users can all be on different nodes.

At the same time, the BiiN™ OS provides many ways to tailor the use of distribution, if developers choose. Developers can implement new distributed services using the same techniques used within the OS. For example, a developer creating an electronic mail application could implement it as a new distributed service.

7.2 How a Distributed Service Works

Distributed computing is implemented by the BiiN™ OS. The OS provides many distinct *services*, such as the filing service, terminal service, and transaction service. Most OS services are implemented as *distributed services* that can transparently provide service at any node of a distributed system.

There is a copy, or *instance*, of a distributed service at each node within the network. The instances communicate as necessary. For example if a user job accesses a file F, the job calls the filing service at its node. If F is stored at another node, the filing service instances at the two nodes communicate and cooperate to perform the needed access (Figure 7-1).

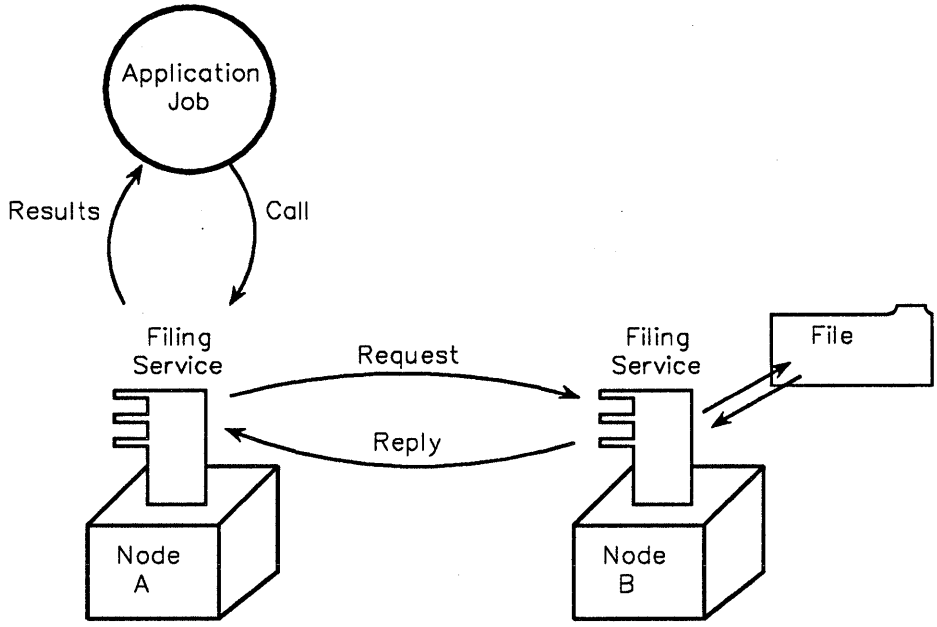


Figure 7-1. Communication and Cooperation Between Filing Service Instances

Communication between instances of a distributed service is independent of both the underlying network protocols and the paths that messages take through the network. Such communication uses a high-level *transport* protocol that hides network details.

7.3 Distributed Disk Storage

Perhaps the hardest and most important problem for designers of distributed systems is how to provide a distributed filing system. The problems to solve include:

Naming and Reference
: To be independent of the underlying network, naming objects or otherwise pointing to them must be done without using network path or location information.

Efficiency
: Because there is transparent access to objects stored at other nodes—often without the explicit knowledge of your application—it is important that access be efficient and not unduly slow down applications.

Concurrency
: Because a distributed system may have more simultaneous users and jobs than any single node, it must gracefully handle concurrent access to objects by different jobs.

Consistency
: If a distributed system allows multiple versions of an object to exist simultaneously at different nodes, then there must be mechanisms for reconciling changes and ensuring overall consistency.

A BiiN™ system's distributed filing system is called *passive store* and provides solutions for all these problems. In many ways, passive store is the glue that holds together a distributed BiiN™ system (Figure 7-2).

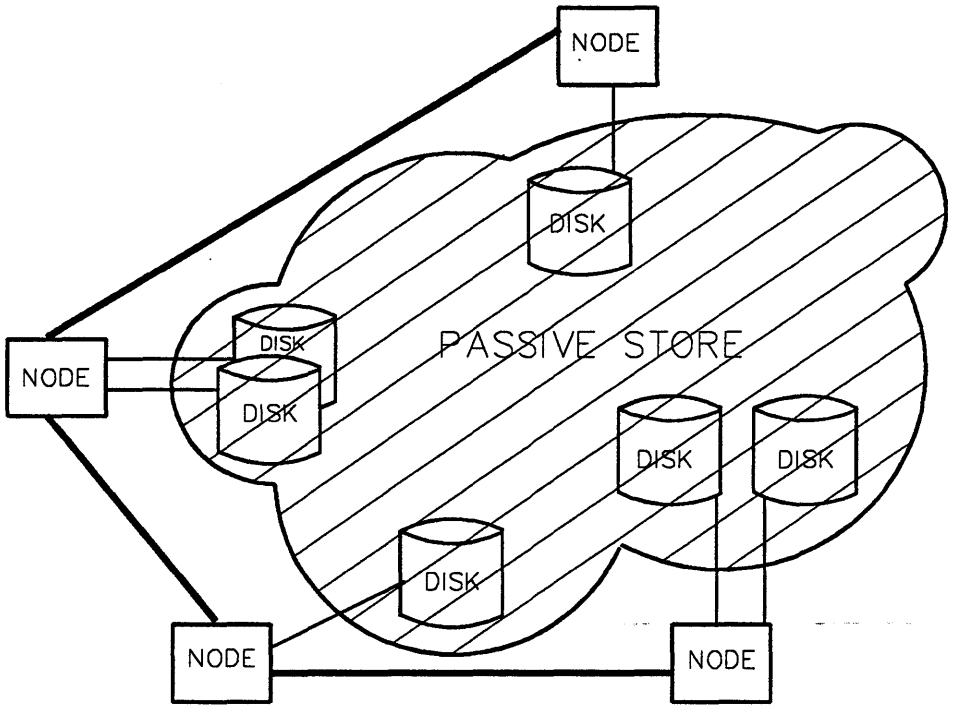


Figure 7-2. Passive Store is a Distributed Object Filing Service that Unifies All Nodes in a BiiN™ System.

7.3.1 General Characteristics of Passive Store

Passive store has these general characteristics:

- Arbitrary objects can be stored on disk. (Objects are described in Chapter 6).
- Objects are retrieved from disk or *activated* on demand, transparent to applications.
- Objects are stored on disk (*passivated* or *updated*) on command. Updating can be controlled either by applications or by a program module that manages a particular object type.
- An object can have multiple active versions simultaneously, in different jobs. An object can have only one passive version.
- Conventional files and directories are two types of objects that can be stored on disks. Not just files but any kind of object can be stored in directories.
- How passive store behaves can be customized for each type of object.

7.3.2 Naming and Reference for Passive Objects

Passive objects can be referenced in two ways, with familiar *pathnames* or with object pointers called *access descriptors* (ADs).

Pathnames are multi-level readable names that identify a stored object. For example, the full pathname of a user's home directory might be `///spirit_motors/sales/home/dagnyt`. Several forms of pathnames are supported. For example, if Dagny is in her home directory, then the *relative pathname* `widget` can be used as shorthand for the full pathname `///spirit_motors/sales/home/dagnyt/widget`. The full pathname contains a logical *naming domain*, `///spirit_motors/sales`, typically corresponding to a department within a company. Changes in the networks used by the department or the company don't invalidate such pathnames. The OS *clearinghouse* service maintains a distributed map of where to find nodes, users, and volume sets used by full pathnames.

When a passive object is created, it is given a permanent unique identifier. Part of the identifier identifies the object's volume set, the "logical disk" that contains the object. The object's unique identifier remains valid even if its volume set is moved to another disk, to another node, or to any other BiiN™ system at any time or place. An AD (pointer) to a passive object references the object by its unique identifier.

When an AD is loaded from disk into a node's virtual memory, it is transformed to a corresponding active AD. The reverse transformation takes place when an AD is written to disk. Applications only need to use one kind of AD, the hardware-supported active form. The OS transparently handles the mapping between active and passive ADs.

7.3.3 Efficient Access to Passive Store

Consider two extremes in accessing an object at another node:

1. An application wants to write a new value for a single byte in the object and will not use the object again.
2. An application repeatedly reads large amounts of data from an object and uses the object as long as it is executing.

The most efficient implementation of the first operation simply sends a message to the node where the object is stored, and performs the operation at that node. This avoids the overhead of copying the entire object over the network and then back again for a small one-time change. The most efficient implementation of the second operation activates the object over the network and creates an active version for the application. The application can then use that active version repeatedly without further network traffic. Passive store supports both sorts of access models:

single activation Operations go to the object, which is only activated on its home node. Network messages are sent to and from the home node for each operation requested from another node. See Figure 7-3.

multiple activation The object goes to operations. The object is activated over the network for each remote job that uses it. When a job updates the object's passive version, the updated version is copied back over the network to its home node. See Figure 7-4.

The choice of an activation model is concealed within the implementation of a particular distributed service, and is not visible to applications.

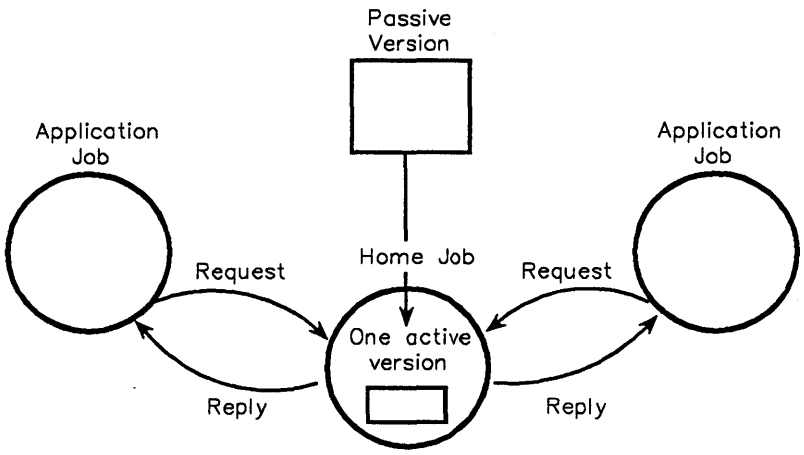


Figure 7-3. Single Activation Access to a Passive Object

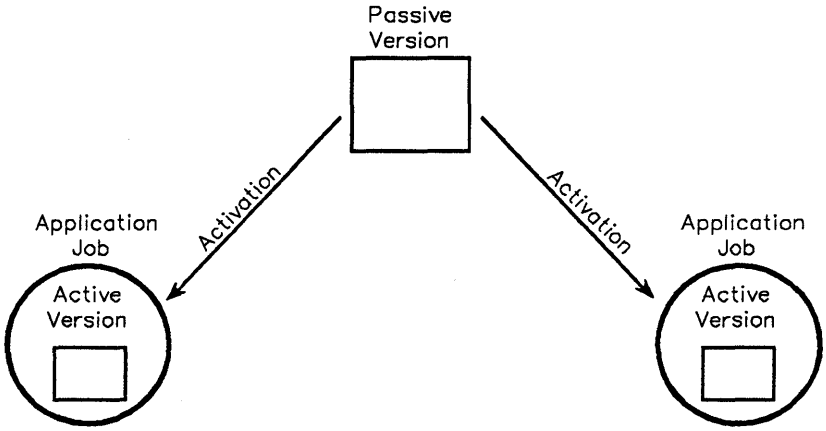


Figure 7-4. Multiple Activation Access to A Passive Object

7.3.4 Handling Concurrent Access and Ensuring Consistency of Passive Objects

In the single-activation model, an object is only activated at its home node in a "home job." That job can act as a server and maintain a queue of requests, servicing one request at a time for a particular object. This ensures that the object is not changed or used in inconsistent ways by two different clients simultaneously.

In the multiple-activation model, several applications may simultaneously have active versions of an object, and may make changes that are inconsistent. Two techniques for handling concurrent access are supported by the OS:

- A job using an object can *reserve* the object for its exclusive use, preventing other jobs from using the object until it is released.

- If one job updates an object's passive version, the OS marks any active versions in other jobs as out-of-date. If one of those other jobs attempts an update, the update is rejected. The other job can then "reset" its active version and redo whatever change it made to the object. This optimistic approach to concurrency and consistency does not block other jobs and incurs no overhead except when updates clash.

The service implementer chooses an approach to handling concurrent access and ensuring consistency. The choice is not visible to users of the service. Whatever choices are made, the OS never allows a passive object to be updated from an out-of-date active version.

7.4 Distributed Program Execution

Users can take advantage of distribution to balance processing loads between hardware nodes on a Local Area Network (LAN). You may be logged into node A but can easily request that a job run on a particular node. For example, if a user knows that a major job will make heavy use of files on a particular node, then the user might request that node.

Because BiiN™ systems are distributed, the OS distinguishes between *jobs* and *processes.* Both jobs and processes are units of program execution. A job is a virtual computer; each job has its own address space, memory resource, and processing resource. Just as a BiiN™ computer can contain multiple processors, a job can contain multiple processes running concurrently and sharing data and resources. Jobs do not share memory with each other and interact "at arms length." Jobs interact in the same way whether they are at the same node or different nodes; thus jobs are the units of distributed execution. All processes within a job run at one node and can share memory. Processes in a job can interact quickly and efficiently using shared data structures.

7.5 Distributed System Administration

The BiiN™ system provides administrative utilities so that a central system administrator can manage an entire distributed system with hundreds or thousands of nodes:

- Add a user or manage user information for the entire system.
- Control access to stored data throughout the entire system.
- Set resource limits and control resource accounting for the entire system.
- Manage spool queues and I/O devices for the entire system.
- Manage backups for the entire system.
- Manage the Clearinghouse that locates nodes, users, and volume sets within the system.

Because a distributed system can span many computers, BiiN™ system administration supports *decentralized administration,* in which multiple administrators each administer a local part of a system. For example:

- A particular user can be made the administrator of a particular volume set.
- A particular user can be given control rights for a particular spool queue.
- An administrator can be assigned for a particular organization or naming domain in the Clearinghouse, representing a particular division or department within the company.

7.6 Support for Multiple Protocols

The BiiN™ system initially supports these industry-standard network protocols:

- Ethernet Local Area Network (LAN) or IEEE 802.2 and 802.3 LAN (up to 10 Mbit/sec)
- HDLC, RS-422 (up to 1 Mbit/sec)
- X.25 standard for packet-switched wide area networks (speed depending on the underlying network).

Future product offerings may add support for additional protocols.

The BiiN™ architecture for network communication and for distributed computing adheres to the International Standard Organization's Reference Model of Open Systems Interconnection.

7.7 Flexible Network Topologies

A single BiiN™ distributed system can use several networking protocols. For example, LANs can be used within each department at a company site, HDLC lines can interconnect the LANs at the site, and X.25 connections can be used to interconnect the site and other sites world-wide.

7.8 Connecting to Non-BiiN™ Systems

Because BiiN™ systems support industry-standard protocols, data can easily be transferred to and from non-BiiN™ systems. Two products ease such transfers:

- The File Transfer utility is an open network application that supports standardized file exchange between BiiN™ and non-BiiN™ systems. The utility follows the ISO standard on File Transfer, Access, and Management (FTAM).
- The COBOL High-level-language Communication Facility (HCF) implements and extends a subset of the COBOL facilities for data exchange between programs communicating via a network.

INDEX

A

Access descriptor 6-2
Access rights 6-2
Activation model 7-4
AD 6-2
Ada 3-2
Administrator utilities 3-2
Asynchronous communication 2-4
Authority list 6-5
Authority list evaluation 6-6

B

Basic fault tolerance 5-1
BiiN/OS 3-5
BiiN/UX 1-1, 3-1, 3-4
Bus 2-3, 4-2, 5-1
Bus Exchange Unit 2-2, 2-4, 4-1
BXU 2-2, 2-4, 4-1

C

C 3-2, 3-5
Caching 4-1
Central Processing Unit 2-2, 2-3, 4-1, 5-1
Channel Processor 2-2, 4-2
Chips 2-2
Clearinghouse 7-4
CLEX 3-1
COBOL 3-3
Command interpreter 3-1, 3-4
Command language 3-1, 3-4
Command Language Executive 3-1
Concurrent execution 3-2
Concurrent programming 3-5
Configuration management 3-4
Continuous operation 1-2, 5-1, 5-3
CP 2-2, 4-2
CPU 2-2, 2-3, 4-1, 5-1
Current loop interface 2-4

D

Data integrity 1-2, 6-1
Data protection 1-2, 6-1
Database 3-4
Database management system 3-4
DBMS 3-4
Debugger 3-3
Diagnostics 5-4
Disk 5-2
Disk mirroring 5-2
Dispatch port 4-3
Distributed computing 1-3, 3-5, 7-1
Distributed filing system 7-2
Distributed services 7-1
Domain 6-3
DRAM 2-4, 5-2

E

ECC 2-2, 2-4, 5-2
Editor 3-3, 3-5
Emacs 3-3
End-to-end check 5-2
Error Correction Code 2-2, 2-4, 5-2
Ethernet 2-4

F

Fault checking 1-2, 5-1, 5-2
Fault coverage 5-3
Fault recovery 5-3
Fault tolerance 1-2, 5-1
Fault tolerance levels 5-1
File transfer 7-7
Floating-point arithmetic 3-3
Form editor 3-4
Forms 3-4
FORTRAN 3-3

G

Goals 1-1
Graphics 3-5

H

Hardware 2-1
Hardware architecture 2-1
HDLC 2-4
Help facility 3-1
High-level Data Link Control 2-4

I

I/O 2-4, 4-2
I/O subsystem 2-4
ID
IEEE 802.2/3 2-4
Intelligent Peripheral Interface 2-4
IPI 2-4

K

L

Librarian 3-3
Linear address space 6-3
Linker 3-3
Local area networks 2-4

M

MCU 2-2
Memory 2-4, 5-2
Memory Control Unit 2-2
Mirroring 5-2
MLisp 3-3
Modem interface 2-4
Multiple activation 7-4
Multiprocessing 4-2, 3-5

N

Networks 1-3, 7-1

O

Object 6-1
Object type 6-1
Object-based protection 6-1
Online repair 5-4
Operating system 3-5
OS 3-5

P

Pascal 3-3
Passive store 7-2, 7-3
Pathnames 7-3
Performance 1-1, 4-1
Performance range 1-1
Program libraries 3-3
Programming languages 1-1, 3-2
Programming tools 3-3, 3-5
Project tracking 3-4

Q

QMR 5-3
Quad Modular Redundancy 5-3

R

RAM 2-4, 5-2
Real-time programming 3-5
Record-oriented applications 3-5
Rep rights 6-2
Report editor 3-4
Reports 3-4
Representation rights 6-2
Rights 6-2
RS-232-C 2-4
RS-422 2-4
Runtime commands 3-2

S

SCSI 2-4
SDLC 2-4
Serial communication 2-4
Series 60 2-1, 2-3
Single activation 7-4
Small Computer Systems Interface 2-4
SMS 3-4
Software 3-1
Software development 1-1
Software Management System 3-4
Spare bit 5-2
SQL 3-3, 3-4
Stable store 4-2
Standards 1-3
Structured Query Language 3-3, 3-4
Surface mount 2-2
Synchronous communication 2-4
Synchronous Data Link Control 2-4
System administrator utilities 3-2
System Bus 2-3, 4-2, 5-1
System goals 1-1

T

Tagged memory 6-2
Text editor 3-3, 3-5
Text processing 3-5
Transaction processing 3-5
Transactions 4-2, 6-6
Transparent multiprocessing 4-2
Troubleshooting 5-4
TTY current loop interface 2-4
Type manager 6-3
Type rights 6-2

U

Unique identifier 7-4
UNIX 1-1, 3-1, 3-4
User utilities 3-2, 3-4
Utilities 3-2, 3-4

V

V.21 2-4
V.22 2-4
V.22bis 2-4
V.23 2-4
V.25bis 2-4
V.26 2-4
V.26bis 2-4
V.26ter 2-4
V.27 2-4
V.27bis 2-4
V.27ter 2-4
V.29 2-4
V.32 2-4
Version control 3-4, 3-5
Virtual address 6-2
VLSI chips 2-2
Volume set mirroring 5-2

X

X Window System 3-5
X.21 2-4
X.24 2-4