CLIX™

System Guide

INTERGRAPH

SE

CLIX™

System Guide

INTERGRAPH

DSYS18510

CLIX System Guide

January 1990

DSYS18512
Edition 1

Copyrights

Additional References

The following UNIX System V documentation is required reference material. These documents can be purchased individually or in sets from Intergraph:

Title	Release V.3
AT&T UNIX System V User's Reference Manual	DSYS08110
AT&T UNIX System V User's Reference Addendum	DSYS19410
AT&T UNIX System V Administrator's Reference Manual	DSYS08310
AT&T UNIX System V Administrator's Reference Addendum	DSYS19710
AT&T UNIX System V Programmer's Reference Manual	DSYS08510
AT&T UNIX System V Programmer's Reference Addendum	DSYS19510

The following UNIX System V documentation is suggested reference material. The following documents can be purchased individually or in sets from Intergraph:

Title	Release V.3
AT&T UNIX System V User's Guide	DSYS08010
AT&T UNIX System V Programming Guide	DSYS08410
AT&T UNIX System V Administrator's Guide	DSYS08210

Ordering Information

To order any of these documents:

- Within the United States contact your Customer Engineer or Sales Account Representative.
- For International locations, contact the Intergraph subsidiary or distributor where you purchased your workstation.

Support Information

If you have trouble with the workstation/server or the procedures described in this guide, contact Intergraph Customer Support at 1-800-633-7248. International customers should contact the Intergraph subsidiary or distributor where the workstation was purchased.

Introduction

The *CLIX System Guide* contains procedures and tutorials designed to give instructions in how to perform tasks. It also provides background information explaining when and why these tasks are needed.

The following documents provide related information:

- The *CLIX System Administrator's Reference Manual* describes the commands and special interfaces used by those who administer a CLIX system.
- The *CLIX Programmer's & User's Reference Manual* describes the commands that constitute the basic software running on an Intergraph workstation or server, as well as system calls, library routines, file formats, and miscellaneous facilities.

The *CLIX System Guide* is divided into the following sections:

Part 1: System Administrator's Tutorials

1. FFS Tutorial
2. FFS Check Tutorial
3. BSD LP Spooler Tutorial
4. NQS Tutorial
5. YP Tutorial

Part 2: System Administrator's Procedures

1. System Rebuild
2. New Product Delivery
3. System Reconfiguration
4. FFS Installation
5. BSD Network Configuration
6. NFS/YP Installation
7. NQS Installation

Part 3: Programmer's & User's Tutorials

1. Technical Programming Tutorial
2. PROC Debugging Tutorial
3. Network Programming Tutorial
4. BSD Porting Tutorial
5. Introductory Socket Tutorial
6. Advanced Socket Tutorial
7. NQS Tutorial
8. RCS Tutorial
9. RPC/XDR Tutorial

The *CLIX System Administrator's Reference Manual* is divided into the following sections:

(1M) System Administrator Commands

(7) Special Interfaces

- (7S) Special Files
- (7B) BSD Network Interfaces
- (7A) Asynchronous Interfaces

The *CLIX Programmer's & User's Reference Manual* is divided into the following sections:

(1) Commands

(2) System Calls

- (2B) BSD System Calls
- (2I) Intergraph System Calls

(3) Library Routines

- (3C) and (3S) C Programming Language Utilities
- (3B) BSD Library Routines
- (3N) Intergraph Network Library Routines
- (3R) RPC/XDR/YP Library Routines
- (3A) Intergraph Synchronous/Asynchronous Library Routines

(4) File Formats

(5) Miscellaneous

Format

The following conventions are used throughout this document:

Boldface	User input such as commands, options and arguments to commands, directories, and files appear in **bold**.
Italic	Substitutable values or new terms appearing for the first time appear in *italics*.
`constant width`	Text that is printed on your terminal or program code appears in a `constant width` font.
command(number)	A command name followed by a number in parentheses refers to the corresponding part of the appropriate CLIX or UNIX System V reference manual as follows:

- Look up references followed by (1M), (7S), (7B), or (7A) in this document.
- Look up references followed by (1), (2B), (2I), (3C), (3B), (3N), (3R), (3A), (4), or (5) in the *CLIX Programmer's & User's Reference Manual.*
- Look up all other references in the appropriate CLIX document.

If the references are not in the CLIX document, refer to the appropriate UNIX System V manual.

Part 1: System Administrator's Tutorials

1. FFS Tutorial

Chapter 1: FFS Tutorial

Introduction

Fast File System is a reimplementation of the UNIX® file system. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies that allow better locality of reference and can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access to large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the standard UNIX file system are experienced. Long needed enhancements to the programmers' interface are discussed. These include a mechanism to place advisory locks on files, extensions of the name space across file systems, the ability to use long file names, and provisions for administrative control of resource usage.

This tutorial presents the motivations for changes in the file system, the methods used to effect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results obtained, directions for future work, and the additions and changes made to the facilities available to programmers.

The original UNIX system has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512-byte blocks, which can be placed arbitrarily within the data area of the file system. Virtually no constraints other than available disk space are placed on file growth [Ritchie74], [Thompson78]. (In practice, a file's size is constrained to be less than about one gigabyte.)

When used on the VAX™-11 with other UNIX enhancements, the original 512-byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications such as VLSI design and image processing do a small amount of processing on large quantities of data and need to have a high throughput from the file system. High throughput rates are also needed by programs that map files from the file system into large virtual address spaces. Paging data in and out of the file system is likely to occur frequently [Ferrin82b]. This requires a file system providing higher bandwidth than the original 512-byte UNIX, one that provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

The UNIX file system has been modified to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and changed the underlying implementation to increase its throughput. Consequently, users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. Previous work to improve the UNIX file system performance has been done by [Ferrin82a]. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a LISP environment [Symbolics81]. A good introduction to the physical latencies of disks is described in [Pechura83].

Old File System

In the file system developed at Bell™ Laboratories (the "traditional" file system), each disk drive is divided into one or more partitions. Each of these disk partitions may contain one file system. A file system never spans multiple partitions. (By "partition" here we refer to the subdivision of physical space on a disk drive. In the traditional file system as in the new file system, file systems are really located in logical disk partitions that may overlap. This overlapping is available, for example, to allow programs to copy entire disk drives containing multiple file systems.) A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to the *free list*, a linked list of all free blocks in the file system.

Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may be directories. Every file has a descriptor associated with it called an *i-node*. An i-node contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For this section, we assume that the first eight blocks of the file are directly referenced by values stored in an i-node. (The actual number may vary from system to system, but is usually in the 5-13 range.) An i-node may also contain references to indirect blocks containing further data block indices. In a file system with a 512-byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further singly indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A 150-megabyte traditional UNIX file system consists of four megabytes of i-nodes followed by 146 megabytes of data. This organization segregates the i-node information from the data; thus, accessing a file normally incurs a long seek from the file's i-node to its data. Files in a single directory are not typically allocated consecutive slots in the four megabytes of i-nodes, causing many nonconsecutive blocks of i-nodes to be accessed when executing operations on the i-nodes of several files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512-byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by staging modifications to critical file system information so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase happened because of two factors: each disk transfer accessed twice as much data, and most files could be described without needing to access indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system.*

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that, although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually, the free list became entirely random, causing files to have their blocks allocated randomly over the disk. This forced a seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of this randomization of data block placement. There was no way of restoring the performance of an old file system except to dump, rebuild, and restore the file system. Another possibility, suggested by [Maruyama76], would be to have a process that periodically reorganized the data on the disk to restore locality.

New File System Organization

In the new file system organization (as in the old file system organization), each disk drive contains one or more file systems. A file system is described by its super-block, located at the beginning of the file system's disk partition. Because the super-block contains critical data, it is replicated to protect against catastrophic loss. This is done when the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To ensure that it is possible to create files as large as 2^{32} bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of 2 greater than or equal to 4096. The block size of a file system is recorded in the file system's super-block so it is possible for file systems with different block sizes to be simultaneously accessible on the same system. The block size must be decided when the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization divides a disk partition into one or more areas called *cylinder groups*. A cylinder group is composed of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for i-nodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. The bit map of available blocks in the cylinder group replaces the traditional file system's free list. For each cylinder group, a static number of i-nodes is allocated at file system creation time. The default policy is to allocate one i-node for each 2048 bytes of space in the cylinder group, expecting this to be far more than will ever be needed.

All cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all redundant copies of the super-block. Thus, the cylinder group bookkeeping information begins at a varying offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group than the preceding cylinder group. In this way, the redundant information spirals down to the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-block. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks. (While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with block sizes of 16 kilobytes or greater. This results from a requirement that the first 8 kilobytes of the disk be reserved for a bootstrap program and a separate requirement that the cylinder group information begin on a file system block boundary. To start the cylinder group on a file system block boundary, file systems with block sizes larger than 8 kilobytes would have to leave an empty space between the end of the boot block and the beginning of the cylinder group. Without knowing the size of the file system blocks, the system would not know what roundup function to use to find the beginning of the first cylinder group.)

Optimizing Storage Utilization

Data is laid out so that larger blocks can be transferred in a single disk transaction, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096-byte data blocks. In the old file system, this file would be composed of 1024-byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096-byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before requiring a seek.

The main problem with larger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Figure 1-1 shows the effect of file system block size on the amount of wasted space in the file system. The files measured to obtain these figures reside on one of our time sharing systems that has roughly 1.2 gigabytes of online storage. The measurements are based on the active user file systems containing about 920 megabytes of formatted space.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512-byte boundary
828.7 Mb	6.9	Data + i-nodes, 512-byte block UNIX file system
866.5 Mb	11.8	Data + i-nodes, 1024-byte block UNIX file system
948.5 Mb	22.4	Data + i-nodes, 2048-byte block UNIX file system
1128.3 Mb	45.6	Data + i-nodes, 4096-byte block UNIX file system

Figure 1-1: Amount of Wasted Space as a Function of Block Size

The space wasted is calculated to be the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly to an intolerable 45.6 percent waste with 4096-byte file system blocks.

To be able to use large blocks without undue waste, small files must be stored more efficiently. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified when the file system is created; each file system block can optionally be broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space available in a cylinder group at the fragment level; to determine if a block is available, aligned fragments are examined. Figure 1-1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1-2: Example Layout of Blocks and Fragments in a 4096/1024 File System

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0-5, 10, and 11 are in use, while fragments 6-9 and 12-15 are free. Fragments of adjoining blocks cannot be used as a full block, even if they are large enough. In this example, fragments 6-9 cannot be allocated as a full block; only fragments 12-15 can be coalesced into a full block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096-byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remaining fragments of the block are made available for allocation to other files. As an example, consider an 11,000-byte file stored on a 4096/1024-byte file system. This file would use two full-size blocks and one three-fragment portion of another block. If a block with three aligned fragments is not available at the time the file is created, a full-size block is split, yielding the necessary fragments and a single unused fragment. This remaining fragment can be allocated to another file as needed.

Space is allocated to a file when a program does a *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased. A program may be overwriting data in the middle of an existing file. In this case, space would already have been allocated. If the file needs to be expanded to hold the new data, one of three conditions exists:

1. Enough space is left in an already allocated block or fragment to hold the new data. The new data is written into the available space.
2. The file contains no fragmented blocks (and the last block in the file contains insufficient space to hold the new data). If space exists in a block already allocated, the space is filled with new data. If the remainder of the new data contains more than a full block of data, a full block is allocated and the first full block of new data is written there. This process is repeated until less than a full block of new data remains. If the remaining new data to be written will fit in less than a full block, a block with the necessary fragments is located. Otherwise, a full block is located. The remaining new data is written into the located space.

3. The file contains one or more fragments (and the fragments contain insufficient space to hold the new data). If the size of the new data plus the size of the data already in the fragments exceeds the size of a full block, a new block is allocated. The contents of the fragments are copied to the beginning of the block and the remainder of the block is filled with new data. The process then continues as in (2) above. Otherwise, if the new data to be written will fit in less than a full block, a block with the necessary fragments is located. Otherwise, a full block is located. The contents of the existing fragments appended with the new data are written into the allocated space.

The problem with expanding a file one fragment at a time is that data may be copied many times as a fragmented block expands to a full block. Fragment reallocation can be minimized if the user program writes a full block at a time, except for a partial block at the end of the file. Since file systems with different block sizes may reside on the same system, the file system interface has been extended to provide application programs the optimal size for a read or write. For files, the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The amount of wasted space in the 4096/1024-byte new file system organization is empirically observed to be about the same as in the 1024-byte old file system organization. A file system with 4096-byte blocks and 512-byte fragments has about the same amount of wasted space as the 512-byte block UNIX file system. The new file system uses less space than the 512-byte or 1024-byte file systems for indexing information for large files and the same amount of space for small files. These savings are offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when a new file system's fragment size equals an old file system's block size.

For the layout policies to be effective, a file system cannot be kept completely full. For each file system there is a parameter, termed the free space reserve, that gives the minimum acceptable percentage of file system blocks that should be free. If the number of free blocks drops below this level, only the system administrator can continue to allocate blocks. The value of this parameter may be changed at any time, even when the file system is mounted and active. The transfer rates that appear in section 4 were measured on file systems kept less than 90 percent full (a reserve of 10 percent). If the number of free blocks falls to zero, the file system throughput tends to be cut in half, because of the inability of the file system to localize blocks in a file. If a file system's performance degrades because of overfilling, it may be restored by removing files until the amount of free space once again reaches the minimum acceptable level. Access rates for files created during periods of little free space may be restored by moving their data once enough space is available. The free space reserve must be added to the percentage of waste when comparing the organizations given in Figure 1-1. Thus, the percentage of waste in an old 1024-byte UNIX file system is roughly comparable to a new 4096/512-byte file system with the free space reserve set at 5 percent. (Compare 11.8 percent wasted with the old file system to 6.9 percent waste + 5 percent reserved space in the new file system.)

File System Parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration-dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can be adapted to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be rotationally well positioned. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block, depending on system characteristics. On a processor with an input/output channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks can often be accessed without suffering lost time because of an intervening disk revolution. For processors without input/output channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to service an interrupt and schedule a new disk transfer. Given a block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in the file will come into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the available blocks in a cylinder group at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution-per-minute drive. The super-block contains a vector of lists called *rotational layout tables*. The vector is indexed by rotational position. Each component of the vector lists the index into the block map for every data block contained in its rotational position. When looking for an allocatable block, the system first looks through the summary counts for a rotational position with a nonzero block count. It then uses the index of the rotational position to find the appropriate list to use to index through only the relevant parts of the block map to find a free block.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with a rotational separation of two milliseconds, and the disk pack is then moved to a system that has a processor requiring four milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

Layout Policies

The file system layout policies are divided into two distinct parts. At the top level are global policies that use file system-wide summary information to decide the placement of new i-nodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because insufficient blocks remain in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80] and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space, forcing the data to be scattered to nonlocal cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is i-nodes. I-nodes are used to describe both files and directories. I-nodes of files in the same directory are frequently accessed together. For example, the "list directory" command often accesses the i-node for each file in a directory. The layout policy tries to place all i-nodes of files in a directory in the same cylinder group. To ensure that files are distributed throughout the disk, a different policy is used for directory allocation. A new directory is placed in a cylinder group that has a greater than average number of free i-nodes, and the smallest number of directories already in it. The intent of this policy is to allow the i-node clustering policy to succeed most of the time. The allocation of i-nodes within a cylinder group is done using a next free strategy. Although this allocates the i-nodes randomly within a cylinder group, all the i-nodes for a particular cylinder group can be read with 8 to 16 disk transfers. (At most, 16 disk transfers are required because a cylinder group may have no more than 2048 i-nodes.) This puts a small and constant upper bound on the number of disk transfers required to access the i-nodes for all files in a directory. In contrast, the old file system typically requires one disk transfer to fetch the i-node for each file in a directory.

The other major resource is data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. The problem with allocating all data blocks in the same cylinder group is that large files will quickly use available space in the cylinder group, forcing a spill over to other areas. Further, using all the space in a cylinder group causes future allocations for any file in the cylinder group to also spill to other areas. Ideally, none of the cylinder groups should ever become completely full. The heuristic solution chosen is to redirect block allocation to a different cylinder group when a file exceeds 48 kilobytes, and at every megabyte thereafter. The first spill over point at 48 kilobytes is where a file on a 4096-byte block file system first requires a single indirect block. This appears to be a natural first point at which to redirect block allocation. The other spill-over points are chosen with the intent of forcing block allocation to be redirected when a file has used about 25 percent of the data blocks in a cylinder group. In observing the new file system in day-to-day use, the heuristics appear to work well in minimizing the number of completely filled cylinder groups.

The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. Otherwise, it allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus, the implementation of the global layout policy uses heuristics that employ only partial information.

If a requested block is not available, the local allocator uses a four-level allocation strategy: Use the next available block rotationally closest to the requested block on the same cylinder. It is assumed here that head switching time is zero. On disk controllers where this is not the case, it may be possible to incorporate the time required to switch between disk platters when constructing the rotational layout tables. This, however, has not yet been tried. If blocks are not available on the same cylinder, use a block within the same cylinder group. If that cylinder group is entirely full, quadratically hash the cylinder group number to choose another cylinder group to look for a free block. Finally if the hash fails, apply an exhaustive search to all cylinder groups.

Quadratic hash is used because of its speed in finding unused slots in nearly full hash tables [Knuth75]. File systems that are parameterized to maintain at least 10 percent free space rarely use this strategy. File systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random; the most important characteristic of the strategy used under such conditions is that the strategy be fast.

Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long-term performance of the new file system.

Our empirical studies have shown that the i-node layout policy has been effective. When running the "list directory" command on a large directory that contains many directories (to force the system to access i-nodes in multiple cylinder groups), the number of disk accesses for i-nodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, with disk accesses for i-nodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Figures 1-3 and 1-4 summarize the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate at which user programs can transfer data to or from a file without performing any processing on it. These programs must read and write enough data to ensure that buffering in the operating system does not affect the results. They are also run at least three times in succession; the first to get the system into a known state and the second two to ensure that the experiment has stabilized and is repeatable. The tests used and their results are discussed in detail in [Kridle83]. (A UNIX command that is similar to the reading test that we used is "cp file /dev/null," where "file" is eight megabytes long.) The systems were running multiuser but were otherwise quiescent. There was no contention for either the CPU or the disk arm. The only difference between the UNIBUS™ and MASSBUS™ tests was the controller. All tests used an AMPEX Capricorn 330-megabyte Winchester disk. All file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured. The same number of system calls were performed in all tests; the basic system call overhead was a negligible portion of the total running time of the tests.

Type of File System	Processor and Bus Measured	Speed	Read Bandwidth	% CPU
old 1024	750/UNIBUS	29 K bytes/sec	29/983 3%	11%
new 4096/1024	750/UNIBUS	221 K bytes/sec	221/983 22%	43%
new 8192/1024	750/UNIBUS	233 K bytes/sec	233/983 24%	29%
new 4096/1024	750/MASSBUS	466 K bytes/sec	466/983 47%	73%
new 8192/1024	750/MASSBUS	466 K bytes/sec	466/983 47%	54%

Figure 1-3: Reading Rates of the Old and New UNIX File Systems

Type of File System	Processor and Bus Measured	Speed	Write Bandwidth	% CPU
old 1024	750/UNIBUS	48 K bytes/sec	48/983 5%	29%
new 4096/1024	750/UNIBUS	142 K bytes/sec	142/983 14%	43%
new 8192/1024	750/UNIBUS	215 K bytes/sec	215/983 22%	46%
new 4096/1024	750/MASSBUS	323 K bytes/sec	323/983 33%	94%
new 8192/1024	750/MASSBUS	466 K bytes/sec	466/983 47%	95%

Figure 1-4: Writing Rates of the Old and New UNIX File Systems

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in the tables were based on a file system with a 10% free space reserve. Synthetic workloads suggest that throughput deteriorates to about half the rates given in the figures when the file systems are full.

The percentage of bandwidth given in Figures 1-3 and 1-4 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is calculated by multiplying the number of bytes on a track by the number of revolutions of the disk per second. The bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-5% of the disk bandwidth, while the new file system uses up to 47% of the bandwidth.

Both reads and writes are faster in the new system than in the old system. The biggest factor in this speedup is because of the larger block size used by the new file system. The overhead of allocating blocks in the new system is greater than the overhead of allocating blocks in the old system. However, fewer blocks need to be allocated in the new system because they are bigger. The net effect is that the cost per byte allocated is approximately the same for both systems.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must work more when allocating blocks than when simply reading them. Note that the writing rates are approximately the same as the reading rates in the 8192-byte block file system; the writing rates are slower than the reading rates in the 4096-byte block file system. The slower write rates occur because the kernel has to perform twice as many disk allocations per second, making the processor unable to keep up with the disk transfer rate.

In contrast, the old file system is about 50 percent faster at writing files than reading them. This is because the write system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced. Hence, disk transfers queue up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek distance, the average seek between the scheduled disk writes is much less than it would be if the data blocks were written in the random disk order in which they are generated. However when the file is read, the read system call is processed synchronously so the disk blocks must be retrieved from the disk in the nonoptimal seek order in which they are requested. This forces the disk scheduler to do long seeks, resulting in a lower throughput rate.

In the new system, the blocks of a file are more optimally ordered on the disk. Even though reads are still synchronous, the requests are presented to the disk in a much better order. Even though the writes are still asynchronous, they are already presented to the disk in minimum seek order so reordering them is not beneficial. Hence, the disk seek latencies that limited the old file system have little effect in the new file system. The cost of allocation is the factor in the new system that causes writes to be slower than reads.

The performance of the new file system is currently limited by memory to memory copy operations required to move data from disk buffers in the system's address space to data buffers in the user's address space. These copy operations account for about 40 percent of the time spent performing an input/output operation. If the buffers in both address spaces were properly aligned, this transfer could be performed without copying by using the VAX virtual memory management hardware. This would be especially desirable when transferring large amounts of data. We did not implement this because it would change the user interface to the file system in two major ways: user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow contiguous disk blocks to be read in a single disk transaction. Many disks used with UNIX systems contain either 32 or 48, 512-byte sectors per track. Each track holds exactly two or three 8192-byte file system blocks, or four or six 4096-byte file system blocks. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than 50% of the available bandwidth. If the next block for a file cannot be laid out contiguously, the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. Block chaining has not been implemented because it would require rewriting all disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up allocations, the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79]. This technique was not included because block allocation currently accounts for less than 10 percent of the time spent in a write system call and, once again, the current throughput rates are already limited by the speed of the available processors.

File System Functional Enhancements

The performance enhancements to the UNIX file system did not require any changes to the semantics or data structures visible to application programs. However, several changes had been generally desired for some time but had not been introduced because they would require users to dump and restore all their file systems. Since the new file system already required all existing file systems to be dumped and restored, these functional enhancements were introduced at this time.

File Locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to use a separate "lock" file. A process would try to create a "lock" file. If the creation succeeded, the process could proceed with its update; if the creation failed, the process would wait and try again. This mechanism had three drawbacks. Processes consumed CPU time by looping over attempts to create locks. Locks left lying around because of system crashes had to be manually removed (normally in a system startup command script). Finally, processes running as system administrator are always permitted to create files, so they were forced to use a different mechanism. While it is possible to get around all these problems, the solutions are not straightforward, so a mechanism for locking files has been added.

The most general schemes allow multiple processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to serialize access to a file with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the standard system applications, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the extent of enforcement. A hard lock is always enforced when a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus, advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks, some override policy must be implemented in the kernel. With advisory locks, the policy is left to the user programs. In the UNIX system, programs with system administrator privilege are allowed to override any protection scheme. Because many of the programs that need to use locks must also run as the system administrator, we chose to implement advisory locks rather than create an additional protection scheme that was inconsistent with the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process may have an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock or an exclusive lock is requested when another process holds any lock, the lock request will block until the lock can be obtained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process may access the file.

Locks are applied or removed only on open files. This means that locks can be manipulated without needing to close and reopen a file. This is useful, for example, when a process wishes to apply a shared lock, read information and determine whether an update is required, and then apply an exclusive lock and update the file.

A request for a lock will cause a process to block if the lock cannot be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock cannot be immediately obtained. Being able to conditionally request a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling occurs, later daemon processes can easily check to see if an active daemon exists. Since locks exist only while the locking processes exist, lock files can never be left active after the processes exit or if the system crashes.

Almost no deadlock detection is attempted. The only deadlock detection performed by the system is that the file to which a lock is applied must not already have a lock of the same type. (The second of two successive calls to apply a lock of the same type will fail.)

Symbolic Links

The traditional UNIX file system allows multiple directory entries in the same file system to reference a single file. Each directory entry "links" a file's name to an i-node and its contents. The link concept is fundamental; i-nodes do not reside in directories, but exist separately and are referenced by links. When all links to an i-node are removed, the i-node is deallocated. This style of referencing an i-node does not allow references across physical file systems, nor does it support intermachine linkage. To avoid these limitations, *symbolic links* similar to the scheme used by Multics [Feiertag71] have been added.

A symbolic link is implemented as a file that contains a path name. When the system encounters a symbolic link while interpreting a component of a path name, the contents of the symbolic link is prepended to the rest of the path name, and this name is interpreted to yield the resulting path name. In UNIX, path names are specified relative to the root of the file system hierarchy, or relative to a process's current working directory. Path names specified relative to the root are called absolute path names. Path names specified relative to the current working directory are termed relative path names. If a symbolic link contains an absolute path name, the absolute path name is used. Otherwise, the contents of the symbolic link are evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a path name that they are using. However, certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links; seven system utilities required changes to use these calls.

In future Berkeley software distributions, it may be possible to reference file systems located on remote machines using path names. When this occurs, it will be possible to create symbolic links that span machines.

Rename

Programs that create a new version of an existing file typically create the new version as a temporary file and then rename the temporary file with the name of the target file. In the old UNIX file system, renaming required three calls to the system. If a program were interrupted or the system crashed between these calls, the target file could be left with only its temporary name. To eliminate this possibility, the *rename* system call has been added. The rename call performs the rename operation in a fashion that guarantees the existence of the target name.

Rename works both on data files and directories. When renaming directories, the system must perform special validation checks to ensure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the descendants of the target directory to ensure that it does not include the directory being moved.

References

[Almes78]	Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.
[Bass81]	Bass, J. "Implementation Description for File Locking," Onyx Systems Inc., 73 E. Trimble Rd, San Jose, CA 95131, Jan. 1981.
[Feiertag71]	Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System," Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct. 1971, pp 35-41.
[Ferrin82a]	Ferrin, T.E., "Performance and Robustness Improvements in Version 7 UNIX," Computer Graphics Laboratory Technical Report 2, School of Pharmacy, University of California, San Francisco, January 1982. Presented at the 1982 Winter Usenix

Conference, Santa Monica, California.

[Ferrin82b]	Ferrin, T.E., "Performance Issues of VMUNIX Revisited," ;login: (The Usenix Association Newsletter), Vol 7, #5, November 1982, pp 3-6.
[Kridle83]	Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2 BSD UNIX," Computer Systems Research Group, Dept. of EECS, Berkeley, CA 94720, Technical Report #8.
[Kowalski78]	Kowalski, T. "FSCK - The UNIX System Check Program," Bell Laboratory, Murray Hill, NJ 07974. March 1978.
[Knuth75]	Kunth, D. "The Art of Computer Programming," Volume 3 - Sorting and Searching, Addison-Wesley Publishing Company Inc, Reading, Mass, 1975, pp 506-549.
[Maruyama76]	Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files," CACM, 19, 11. Nov. 1976, pp 634-642.
[Nevalainen77]	Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods," The Computer Journal, 20, 3, Aug. 1977, pp 245-247.
[Pechura83]	Pechura, M., and Schoeffler, J. "Estimating File Access Time of Floppy Disks," CACM, 26, 10. Oct. 1983, pp 754-763.
[Peterson83]	Peterson, G. "Concurrent Reading While Writing," ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan. 1983, pp 46-55.
[Powell79]	Powell, M. "The DEMOS File System," Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov. 1977, pp 33-42.
[Ritchie74]	Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System," CACM 17, 7, July 1974, pp 365-375.

[Smith81a]	Smith, A. "Input/Output Optimization and Disk Architectures: A Survey," Performance and Evaluation 1, Jan. 1981, pp 104-117.
[Smith81b]	Smith, A. "Bibliography on File and I/O System Optimization and Related Topics," Operating Systems Review, 15, 4, Oct. 1981, pp 39-54.
[Symbolics81]	"Symbolics File System," Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311, Aug. 1981.
[Thompson78]	Thompson, K. "UNIX Implementation," Bell System Technical Journal, 57, 6, part 2, pp 1931-1946. July-August 1978.
[Thompson80]	Thompson, M. "Spice File System," Carnegie-Mellon University, Department of Computer Science, Pittsburg, PA 15213 #CMU-CS-80, Sept 1980.
[Trivedi80]	Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments," Journal of the ACM, 27, 3, July 1980, pp 457-473.
[White80]	White, R. M. "Disk Storage Technology," Scientific American, 243(2), August 1980.

2. FFS Check Tutorial

Chapter 2: FFS Check Tutorial

Introduction

This tutorial reflects the use of **ffsfsck**(1M) with the 4.2 Berkeley Software Distribution (BSD), 4.3 BSD, and CLIX Fast File System (FFS) file system organization. This is a revision of the original paper written by T. J. Kowalski.

File System Check Program (**ffsfsck**(1M)) is an interactive file system check and repair program. **ffsfsck**(1M) uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. Unless there has been a hardware failure, **ffsfsck**(1M) is able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by **ffsfsck**(1M). Both the program and the interaction between the program and the operator are described.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to ensure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. **ffsfsck**(1M) runs in two modes. Normally, it is run noninteractively by the system after a normal boot. When running in this mode, it will only make changes to the file systems that are known to always be correct. If an unexpected inconsistency is found, **ffsfsck**(1M) will exit with a nonzero exit status, leaving the system running single-user. Typically, the operator then runs **ffsfsck**(1M) interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether the suggested correction should be made.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by **ffsfsck**(1M) (the Coast Guard to the rescue) is presented.

Overview of the File System

The file system is discussed in detail in [Mckusick84]; this section gives a brief overview.

Super-block

A file system is described by its *super-block*. The super-block is built when the file system is created (**newfs**(1M)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, **newfs**(1M) replicates it to protect against catastrophic loss. The *default super-block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super-blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *i-node*. The i-node contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the i-node structure itself. (The actual number may vary from system to system, but is usually in the range 5-13.) The i-node structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096-byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks. (The triple indirect block is never needed in practice.)

In order to create files with up to 2^{32} bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when **newfs**(1M) creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

Summary Information

Associated with the super-block is nonreplicated *summary information.* The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, i-nodes and directories in the file system.

Cylinder Groups

The file system partitions the disk into one or more areas called *cylinder groups.* A cylinder group is composed of one or more consecutive cylinders on a disk. Each cylinder group includes i-node slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of i-nodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one i-node for each 2048 bytes of disk space; this is expected to be far more i-nodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However, if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus, the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the *i+1*st cylinder group is approximately one track further from the beginning of the cylinder group than it was for the *i*th cylinder group. In this way, the redundant information spirals down into the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096-byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000-byte file stored on a 4096/1024-byte file system. This file uses two full-size blocks and a 3072-byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split, yielding the necessary 3072 byte fragment and an unused 1024-byte fragment. This remaining fragment can be allocated to another file as needed.

Updates to the File System

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program performs an operation to change the file system, such as a **write**(2), the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the i-node information reflecting the change, is eventually written to disk. The real disk write may not happen until long after the **write**(2) system call has returned. Thus, at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when an **update**(2) is performed (at five-second intervals) by **bdflush**, or by manual operator intervention with the **sync**(1M) command. If the system is halted without writing the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two i-nodes. Such an inconsistency can occur when the system is halted before the pointer to the block in the old i-node has been cleared in the copy of the old i-node on the disk, and after the pointer to the block in the new i-node has been written to the copy of the new i-node on the disk. Here, there is no deterministic method for deciding which i-node should really claim the block. A similar problem can arise with a multiply claimed i-node.

The problem with asynchronous i-node updates can be avoided by deallocating all i-nodes synchronously. Consequently, i-nodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly, i-nodes are kept consistent by synchronously deleting, adding, or changing directory entries.

Fixing Corrupted File Systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or when a mounted file system is taken off-line. The most common operator procedural failure is forgetting to **sync** the system before halting the CPU.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a nonfunctional disk-controller.

Detecting and Correcting Corruption

Normally, **ffsfsck**(1M) is run noninteractively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that **ffsfsck**(1M) will take when it is running interactively. Throughout this paper, we assume that **ffsfsck**(1M) is being run interactively and all possible errors can be encountered. When an inconsistency is discovered in this mode, **ffsfsck**(1M) reports the inconsistency for the operator to chose a corrective action.

A quiescent (unmounted and not being written on) file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. The file system **must** be in a quiescent state when **ffsfsck**(1M) is run since **ffsfsck**(1M) is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the i-nodes, the indirect blocks, and the data blocks containing directory entries.

Super-block Checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or i-nodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of i-nodes, free-block count, and the free-i-node count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of i-nodes. The file-system size and layout information are the most critical pieces of information for **ffsfsck**(1M). While there is no way to actually check these sizes since they are statically determined by **newfs**(1M), **ffsfsck**(1M) can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If **ffsfsck**(1M) detects corruption in the static parameters of the default super-block, **ffsfsck**(1M) requests the operator to specify the location of an alternate super-block.

Free-block Checking

ffsfsck(1M) checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, **ffsfsck**(1M) checks that the number of free blocks plus the number of blocks claimed by the i-nodes equals the total number of blocks in the file system.

If anything is wrong with the block allocation maps, **ffsfsck**(1M) will rebuild them based on the list it has computed of allocated blocks.

The summary information associated with the super-block counts the total number of free blocks within the file system. **ffsfsck**(1M) compares this count to the number of free blocks it found within the file system. If the two counts do not agree, **ffsfsck**(1M) replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free i-nodes within the file system. **ffsfsck**(1M) compares this count to the number of free i-nodes it found within the file system. If the two counts do not agree, **ffsfsck**(1M) replaces the incorrect count in the summary information by the actual free-i-node count.

Checking the I-node State

An individual i-node is not as likely to be corrupted as the allocation information. However, because of the great number of active i-nodes, a few of the i-nodes are usually corrupted.

The list of i-nodes in the file system is checked sequentially starting with i-node 2 (i-node 0 marks unused i-nodes; i-node 1 is saved for future generations) and progressing through the last i-node in the file system. The state of each i-node is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and i-node size.

Each i-node contains a mode word. This mode word describes the type and state of the i-node. I-nodes must be one of six types: regular i-node, directory i-node, symbolic link i-node, special block i-node, special character i-node, or socket i-node. I-nodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formatted i-node. An i-node can get in this state if bad data is written into the i-node list. The only possible corrective action is for **ffsfsck**(1M) to clear the i-node.

I-node Links

Each i-node counts the total number of directory entries linked to the i-node. **ffsfsck**(1M) verifies the link count of each i-node by starting at the root of the file system and descending through the directory structure. The actual link count for each i-node is calculated during the descent.

If the stored link count is nonzero and the actual link count is zero, a directory entry does not appear for the i-node. If this happens, **ffsfsck**(1M) will place the disconnected file in the **lost+found** directory. If the stored and actual link counts are nonzero and unequal, a directory entry may have been added or removed without the i-node being updated. If this happens, **ffsfsck**(1M) replaces the incorrect stored link count by the actual link count.

Each i-node contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the i-node. Since indirect blocks are owned by an i-node, inconsistencies in indirect blocks directly affect the i-node that owns it.

ffsfsck(1M) compares each block number claimed by an i-node against a list of already allocated blocks. If another i-node already claims a block number, the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, **ffsfsck**(1M) will perform a partial second pass over the i-node list to find the i-node of the duplicated block. The second pass is needed, since without examining the files associated with these i-nodes for correct content, not enough information is available to determine which i-node is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), the i-node with the earliest modify time is usually incorrect, and should be cleared. If this happens, **ffsfsck**(1M) prompts the operator to clear both i-nodes. The operator must decide which one should be kept and which one should be cleared.

ffsfsck(1M) checks the range of each block number claimed by an i-node. If the block number is lower than the first data block in the file system or greater than the last data block, the block number is a *bad block number*. Many bad blocks in an i-node are usually caused by an indirect block that was not written to the file system, a condition which can only occur if there has been a hardware failure. If an i-node contains bad block numbers, **ffsfsck**(1M) prompts the operator to clear it.

I-node Data Size

Each i-node contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. **ffsfsck**(1M) computes the actual number of data blocks and compares that block count against the actual number of blocks the i-node claims. If an i-node contains an incorrect count, **ffsfsck**(1M) prompts the operator to fix it.

Each i-node contains a 32 bit size field. The size is the number of data bytes in the file associated with the i-node. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the i-node and comparing that expected block count against the actual number of blocks the i-node claims.

Checking the Data Associated with an I-node

An i-node can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks contain the information stored in a file; symbolic link data blocks contain the path name stored in a link. Directory data blocks contain directory entries. **ffsfsck**(1M) can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory i-node numbers pointing to unallocated i-nodes, directory i-node numbers that are greater than the number of i-nodes in the file system, incorrect directory i-node numbers for "." and "..", and directories that are not attached to the file system. If the i-node number in a directory data block references an unallocated i-node, **ffsfsck**(1M) will remove that directory entry. Again, this condition can only arise when there has been a hardware failure.

If a directory entry i-node number references outside the i-node list, **ffsfsck**(1M) will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory i-node number entry for "." must be the first entry in the directory data block. The i-node number for "." must reference itself; for example, it must equal the i-node number for the directory data block. The directory i-node number entry for ".." must be the second entry in the directory data block. Its value must equal the i-node number for the parent of the directory entry (or the i-node number of the directory data block if the directory is the root directory). If the directory i-node numbers are incorrect, **ffsfsck**(1M) will replace them with the correct values. If there are multiple hard links to a directory, the first one encountered is the real parent to which ".." should point; **ffsfsck**(1M) recommends deletion for the subsequently discovered names.

File System Connectivity

ffsfsck(1M) checks the general connectivity of the file system. If directories are not linked into the file system, **ffsfsck**(1M) links the directory back into the file system in the **lost+found** directory. This condition only occurs when there has been a hardware failure.

References

[Dolotta78]	Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1,* January 1978.
[Joy83]	Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. 4.2BSD System Manual, *University of California at Berkeley, Computer Systems Research Group Technical Report* #4, 1982.
[McKusick84]	McKusick, M., Joy, W., Leffler, S., and Fabry, R. A Fast File System for UNIX, *ACM Transactions on Computer Systems* 2, 3. pp. 181-197, August 1984.
[Ritchie78]	Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1905-29.
[Thompson78]	Thompson, K., UNIX Implementation, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1931-46.

Appendix A - Ffsfsck Error Conditions

Conventions

ffsfsck(1M) is a multi-pass file system check program. Each file system pass invokes a different *phase* of the **ffsfsck**(1M) program. After the initial setup, **ffsfsck**(1M) performs successive phases over each file system, checking blocks and sizes, path names, connectivity, reference counts, and the map of free blocks (possibly rebuilding it) and performs some cleanup.

Normally **ffsfsck**(1M) is run noninteractively to *preen* the file systems after an unclean halt. While preening a file system, it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that **ffsfsck**(1M) will take when it is running interactively. Throughout this appendix many errors have several options that the operator can take. When an inconsistency is detected, **ffsfsck**(1M) reports the error condition to the operator. If a response is required, **ffsfsck**(1M) prints a prompt message and waits for a response. When preening, most errors are fatal. For those that are expected, the response taken is noted. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the phase of the **ffsfsck**(1M) program in which they can occur. The error conditions that may occur in more than one phase will be discussed in initialization.

Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section is concerned with the opening of files and the initialization of tables. It lists error conditions resulting from command-line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file. All the initialization errors are fatal when the file system is being preened.

C **option?**
C is not a legal option to **ffsfsck**(1M); legal options are **-b**, **-y**, and **-n**. **ffsfsck**(1M) terminates on this error condition. See **ffsfsck**(1M) in the *CLIX System Administrator's Reference Manual* for further detail.

cannot alloc NNN bytes for blockmap
cannot alloc NNN bytes for freemap
cannot alloc NNN bytes for statemap
cannot alloc NNN bytes for lncntp
ffsfsck(1M)'s request for memory for its virtual memory tables failed. **ffsfsck**(1M) terminates on this error condition.

Can't open checklist file: *F*
The file system checklist file *F* (usually **/etc/fstab**) cannot be opened for reading. **ffsfsck**(1M) terminates on this error condition. Check access modes of *F*.

Can't stat root
ffsfsck(1M)'s request for statistics about the root directory "/" failed. **ffsfsck**(1M) terminates on this error condition.

Can't stat *F*
Can't make sense out of name *F*
ffsfsck(1M)'s request for statistics about the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

Can't open *F*
ffsfsck(1M)'s request attempt to open the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

F: **(NO WRITE)**
Either the -n flag was specified or **ffsfsck**(1M)'s attempt to open the file system *F* for writing failed. When running manually, all the diagnostics are printed, but no modifications are attempted to fix them.

file is not a block or character device; OK
You have given **ffsfsck**(1M) a regular file name by mistake. Check the type of the file specified.

Possible responses to the OK prompt are:

YES Ignore this error condition.

NO Ignore this file system and continues checking the next file system given.

UNDEFINED OPTIMIZATION IN SUPER-BLOCK (SET TO DEFAULT)
The super-block optimization parameter is neither OPT_TIME nor OPT_SPACE.

Possible responses to the SET TO DEFAULT prompt are:

YES The super-block is set to request optimization to minimize running time of the system.

NO Ignore this error condition.

IMPOSSIBLE MINFREE=*D* IN SUPER-BLOCK (SET TO DEFAULT)
The super-block minimum space percentage is greater than 99% or less then 0%.

Possible responses to the SET TO DEFAULT prompt are:

YES The minfree parameter is set to 10%.

NO Ignore this error condition.

One of the following messages will appear:
MAGIC NUMBER WRONG
NCG OUT OF RANGE
CPG OUT OF RANGE
NCYL DOES NOT JIVE WITH NCG*CPG
SIZE PREPOSTEROUSLY LARGE
TRASHED VALUES IN SUPER BLOCK
and will be followed by the message:
F*: BAD SUPER BLOCK: *B
USE -b OPTION TO FFSFSCK TO SPECIFY LOCATION OF AN ALTERNATE
SUPER-BLOCK TO SUPPLY NEEDED INFORMATION; SEE fsck(1M).
The super-block has been corrupted. An alternative super-block must be selected from among those listed by **newfs**(1M) (in the *CLIX System*

Administrator's Reference Manual) when the file system was created. For file systems with a block size less than 32K, specifying **-b** 32 is a good first choice.

INTERNAL INCONSISTENCY: ***M***
ffsfsck(1M) has had an internal panic, whose message is specified as *M*.

CAN NOT SEEK: BLK ***B*** **(CONTINUE)**
ffsfsck(1M)'s request for moving to a specified block number *B* in the file system failed.

Possible responses to the CONTINUE prompt are:

YES	Attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system. If the block was part of the virtual memory buffer cache, **ffsfsck**(1M) will terminate with the message "Fatal I/O error."
NO	Terminate the program.

CAN NOT READ: BLK ***B*** **(CONTINUE)**
ffsfsck(1M)'s request for reading a specified block number *B* in the file system failed.

Possible responses to the CONTINUE prompt are:

YES Attempt to continue to run the file system check. It will retry the read and print the message:
THE FOLLOWING SECTORS COULD NOT BE READ: ***N***
where *N* indicates the sectors that could not be read. If **ffsfsck**(1M) tries to write back one of the blocks on which the read failed, it will print the message:

WRITING ZERO'ED BLOCK ***N*** **TO DISK**

where *N* indicates the sector that was written with zeros. If the disk is experiencing hardware problems, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system. If the block was part of the virtual memory buffer cache, **ffsfsck**(1M) will terminate with the message "Fatal I/O error".

NO Terminate the program.

CAN NOT WRITE: BLK *B* (CONTINUE)
ffsfsck(1M)'s request for writing a specified block number *B* in the file system failed. The disk is write-protected; check the write-protect lock on the drive.

Possible responses to the CONTINUE prompt are:

YES Attempt to continue to run the file system check. The write operation will be retried with the failed blocks indicated by the message: **THE FOLLOWING SECTORS COULD NOT BE WRITTEN:** *N* where *N* indicates the sectors that could not be written. If the disk is experiencing hardware problems, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system. If the block was part of the virtual memory buffer cache, **ffsfsck**(1M) will terminate with the message "Fatal I/O error."

NO Terminate the program.

bad i-node number DDD to gi-node
An internal error has attempted to read nonexistent i-node *DDD*. This error causes **ffsfsck**(1M) to exit.

Phase 1 - Check Blocks and Sizes

This phase is concerned with the i-node list. It lists error conditions resulting from checking i-node types, setting up the zero-link-count table, examining i-node block numbers for bad or duplicate blocks, checking i-node size, and checking i-node format. All errors in this phase except **INCORRECT BLOCK COUNT** and **PARTIALLY TRUNCATED INODE** are fatal if the file system is being preened.

UNKNOWN FILE TYPE I=*I* (CLEAR)
The mode word of the i-node *I* indicates that the i-node is not a special block i-node, special character i-node, socket i-node, regular i-node,

symbolic link, or directory i-node.

Possible responses to the CLEAR prompt are:

YES	Deallocate i-node *I* by zeroing its contents. This will always invoke the UNALLOCATED error condition in phase 2 for each directory entry pointing to this i-node.
NO	Ignore this error condition.

PARTIALLY TRUNCATED INODE I=*I* (SALVAGE)
ffsfsck(1M) has found i-node *I* whose size is shorter than the number of blocks allocated to it. This condition should only occur if the system crashes while in the midst of truncating a file. When preening, the file system, **ffsfsck**(1M) completes the truncation to the specified size.

Possible responses to SALVAGE are:

YES	Complete the truncation to the size specified in the i-node.
NO	Ignore this error condition.

LINK COUNT TABLE OVERFLOW (CONTINUE)
An internal table for **ffsfsck**(1M) containing allocated i-nodes with a link count of zero cannot allocate more memory. Increase the virtual memory for **ffsfsck**(1M).

Possible responses to the CONTINUE prompt are:

YES	Continue with the program. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system. If another allocated i-node with a zero link count is found, this error condition is repeated.
NO	Terminate the program.

B* BAD I=*I
I-node *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the **EXCESSIVE BAD BLKS** error condition in phase 1 (see next paragraph) if i-node

I has too many block numbers outside the file system range. This error condition will always invoke the **BAD/DUP** error condition in phase 2 and phase 4.

EXCESSIVE BAD BLKS I=*I* (CONTINUE)
There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system associated with i-node *I*.

Possible responses to the CONTINUE prompt are:

YES Ignore the rest of the blocks in this i-node and continue checking with the next i-node in the file system. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system.

NO Terminate the program.

BAD STATE DDD TO BLKERR
An internal error has scrambled **ffsfsck**(1M)'s state map to have the impossible value *DDD*. **ffsfsck**(1M) exits immediately.

B* DUP I=*I
I-node *I* contains block number *B* that is already claimed by another i-node. This error condition may invoke the **EXCESSIVE DUP BLKS** error condition in phase 1 if i-node *I* has too many block numbers claimed by other i-nodes. This error condition will always invoke phase 1b and the **BAD/DUP** error condition in phase 2 and phase 4.

EXCESSIVE DUP BLKS I=*I* (CONTINUE)
There is more than a tolerable number (usually 10) of blocks claimed by other i-nodes.

Possible responses to the CONTINUE prompt are:

YES Ignore the rest of the blocks in this i-node and continue checking with the next i-node in the file system. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system.

NO Terminate the program.

DUP TABLE OVERFLOW (CONTINUE)
An internal table in **ffsfsck**(1M) containing duplicate block numbers cannot allocate any more space. Increase the amount of virtual memory available to **ffsfsck**(1M).

Possible responses to the CONTINUE prompt are:

YES	Continue with the program. This error condition will not allow a complete check of the file system. A second run of **ffsfsck**(1M) should be made to recheck this file system. If another duplicate block is found, this error condition will repeat.
NO	Terminate the program.

PARTIALLY ALLOCATED INODE I=*I* (CLEAR)
I-node *I* is neither allocated nor unallocated.

Possible responses to the CLEAR prompt are:

YES	Deallocate i-node *I* by zeroing its contents.
NO	Ignore this error condition.

INCORRECTBLOCK
The block count for i-node *I* is *X* blocks, but should be *Y* blocks. When preening, the count is corrected.

Possible responses to the CORRECT prompt are:

YES	Replace the block count of i-node *I* with *Y*.
NO	Ignore this error condition.

Phase 1b: Rescan for More Dups

When a duplicate block is found in the file system, the file system is rescanned to find the i-node that previously claimed that block. This section lists the error condition when the duplicate block is found.

B **DUP I=*I***
I-node *I* contains block number *B* that is already claimed by another i-node. This error condition will always invoke the **BAD/DUP** error condition in phase 2. You can determine which i-nodes have overlapping blocks by examining this error condition and the DUP error condition in phase 1.

Phase 2 - Check Path Names

This phase is concerned with removing directory entries pointing to error conditioned i-nodes from phase 1 and phase 1b. This section lists error conditions resulting from root i-node mode and status, directory i-node pointers in range, and directory entries pointing to bad i-nodes and directory integrity checks. All errors in this phase are fatal if the file system is being preened, except for directories not being a multiple of the block size and extraneous hard links.

ROOT INODE UNALLOCATED (ALLOCATE)
The root i-node (usually i-node number 2) has no allocate mode bits. This should never happen.

Possible responses to the ALLOCATE prompt are:

YES	Allocate i-node 2 as the root i-node. The files and directories usually found in the root will be recovered in phase 3 and put in **lost+found**. If the attempt to allocate the root fails, **ffsfsck**(1M) will exit with the message: **CANNOT ALLOCATE ROOT INODE.**
NO	**ffsfsck**(1M) will exit.

ROOT INODE NOT DIRECTORY (REALLOCATE)
The root i-node (usually i-node number 2) is not a directory i-node type.

Possible responses to the REALLOCATE prompt are:

YES Clear the existing contents of the root i-node and reallocate it. The files and directories usually found in the root will be recovered in phase 3 and put in **lost+found**. If the attempt to allocate the root fails, **ffsfsck**(1M) will exit with the message:
CANNOT ALLOCATE ROOT INODE.

NO **ffsfsck**(1M) will then prompt with **FIX**.

Possible responses to the FIX prompt are:

YES Replace the root i-node's type to be a directory. If the root i-node's data blocks are not directory blocks, many error conditions will be produced.

NO Terminate the program.

DUPS/BAD IN ROOT INODE (REALLOCATE)
Phase 1 or phase 1b have found duplicate blocks or bad blocks in the root i-node (usually i-node number 2) for the file system.

Possible responses to the REALLOCATE prompt are:

YES Clear the existing contents of the root i-node and reallocate it. The files and directories usually found in the root will be recovered in phase 3 and put in **lost+found**. If the attempt to allocate the root fails, **ffsfsck**(1M) will exit with the message:
CANNOT ALLOCATE ROOT INODE.

NO **ffsfsck**(1M) will then prompt with **CONTINUE**.

Possible responses to the CONTINUE prompt are:

YES Ignore the **DUPS/BAD** error condition in the root i-node and attempt to continue to run the file system check. If the root i-node is not correct, this may result in many other error conditions.

NO Terminate the program.

NAME TOO LONG *F*
An excessively long path name has been found. This usually indicates loops in the file system name space. This can occur if the super-user has

made circular links to directories. The offending links must be removed.

I OUT OF RANGE I=*I* NAME=*F* (REMOVE)
A directory entry *F* has an i-node number *I* that is greater than the end of the i-node list.

Possible responses to the REMOVE prompt are:

YES	The directory entry *F* is removed.
NO	Ignore this error condition.

UNALLOCATED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* *type=F* (REMOVE)
A directory or file entry *F* points to an unallocated i-node *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and name *F* are printed.

Possible responses to the REMOVE prompt are:

YES	The directory entry *F* is removed.
NO	Ignore this error condition.

DUP/BAD I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* *type=F* (REMOVE)
Phase 1 or phase 1b have found duplicate blocks or bad blocks associated with directory or file entry *F*, i-node *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES	The directory entry *F* is removed.
NO	Ignore this error condition.

ZERO LENGTH DIRECTORY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (REMOVE)
A directory entry *F* has a size *S* that is zero. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the REMOVE prompt are:

YES The directory entry *F* is removed; this will always invoke the BAD/DUP error condition in phase 4.

NO Ignore this error condition.

DIRECTORY TOO SHORT I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)

A directory *F* has been found whose size *S* is less than the minimum size directory. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to the FIX prompt are:

YES Increase the size of the directory to the minimum directory size.

NO Ignore this directory.

DIRECTORY *F* LENGTH *S* NOT MULTIPLE OF *B* (ADJUST)

A directory *F* has been found with size *S* that is not a multiple of the directory block size *B*.

Possible responses to the ADJUST prompt are:

YES The length is rounded up to the appropriate block size. Thus, when preening the file system, only a warning is printed and the directory is adjusted.

NO Ignore the error condition.

DIRECTORY CORRUPTED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (SALVAGE)

A directory with an inconsistent internal state has been found.

Possible responses to the FIX prompt are:

YES Throw away all entries up to the next directory boundary (usually 512-byte) boundary. This drastic action can throw away up to 42 entries, and should be taken only after other recovery efforts have failed.

NO Skip up to the next directory boundary and resume reading, but do not modify the directory.

BAD INODE NUMBER FOR '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose i-node number for '.' does not equal *I*.

Possible responses to the FIX prompt are:

YES Change the i-node number for '.' to be equal to *I*.

NO Leave the i-node number for '.' unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose first entry is unallocated.

Possible responses to the FIX prompt are:

YES Build an entry for '.' with i-node number equal to *I*.

NO Leave the directory unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* CANNOT FIX, FIRST ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose first entry is *F*. **ffsfsck**(1M) cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and **ffsfsck**(1M) should be run again.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* CANNOT FIX, INSUFFICIENT SPACE TO ADD '.'
A directory *I* has been found whose first entry is not '.'. **ffsfsck**(1M) cannot resolve this problem as it should never happen.

EXTRA '.' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '.'.

Possible responses to the FIX prompt are:

YES Remove the extra entry for '.'.

NO Leave the directory unchanged.

BAD INODE NUMBER FOR '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose i-node number for '..' does not equal the parent of *I*.

Possible responses to the FIX prompt are:

YES Change the i-node number for '..' to be equal to the parent of *I* ("..") in the root i-node points to itself.)

NO Leave the i-node number for '..' unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose second entry is unallocated.

Possible responses to the FIX prompt are:

YES build an entry for '..' with i-node number equal to the parent of *I* ("..") in the root i-node points to itself).

NO leave the directory unchanged.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* CANNOT FIX, SECOND ENTRY IN DIRECTORY CONTAINS *F*
A directory *I* has been found whose second entry is *F*. **ffsfsck**(1M) cannot resolve this problem. The file system should be mounted and the offending entry *F* moved elsewhere. The file system should then be unmounted and **ffsfsck**(1M) should be run again.

MISSING '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* CANNOT FIX, INSUFFICIENT SPACE TO ADD '..'
A directory *I* has been found whose second entry is not '..'. **ffsfsck**(1M) cannot resolve this problem. The file system should be mounted and the second entry in the directory moved elsewhere. The file system should then be unmounted and **ffsfsck**(1M) should be run again.

EXTRA '..' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '..'.

Possible responses to the FIX prompt are:

YES	Remove the extra entry for '..'.
NO	Leave the directory unchanged.

***N* IS AN EXTRANEOUS HARD LINK TO A DIRECTORY *D* (REMOVE)**
ffsfsck(1M) has found a hard link, *N*, to a directory, *D*. When preening the extraneous links are ignored.

Possible responses to the REMOVE prompt are:

YES	Delete the extraneous entry, *N*.
NO	Ignore the error condition.

BAD INODE *S* TO DESCEND
An internal error has caused an impossible state *S* to be passed to the routine that descends the file system directory structure. **ffsfsck**(1M) exits.

BAD RETURN STATE *S* FROM DESCEND
An internal error has caused an impossible state *S* to be returned from the routine that descends the file system directory structure. **ffsfsck**(1M) exits.

BAD STATE *S* FOR ROOT INODE
An internal error has caused an impossible state *S* to be assigned to the root i-node. **ffsfsck**(1M) exits.

Phase 3 - Check Connectivity

This phase is concerned with the directory connectivity seen in phase 2. This section lists error conditions resulting from unreferenced directories and missing or full **lost+found** directories.

UNREF DIR I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
The directory i-node *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory i-node *I* are printed. When preening, the directory is reconnected if its size is nonzero. Otherwise, it is cleared.

Possible responses to the RECONNECT prompt are:

YES Reconnect directory i-node *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the **lost+found** error condition in phase 3 if there are problems connecting directory i-node *I* to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

NO Ignore this error condition. This will always invoke the UNREF error condition in phase 4.

NO lost+found DIRECTORY (CREATE)
There is no **lost+found** directory in the root directory of the file system; When preening, **ffsfsck**(1M) tries to create a **lost+found** directory.

Possible responses to the CREATE prompt are:

YES Create a **lost+found** directory in the root of the file system. This may raise the message:
NO SPACE LEFT IN / (EXPAND)
See below for the possible responses. Inability to create a *lost+found* directory generates the message:
SORRY. CANNOT CREATE lost+found DIRECTORY
and aborts the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

NO Abort the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

lost+found IS NOT A DIRECTORY (REALLOCATE)
The entry for **lost+found** is not a directory.

Possible responses to the REALLOCATE prompt are:

YES Allocate a directory i-node, and change **lost+found** to reference it. The previous i-node referenced by the **lost+found** name is not cleared. Thus, it will either be reclaimed as an UNREFed i-node or have its link count ADJUSTed later in this phase. Inability to create a **lost+found** directory generates the message:
SORRY. CANNOT CREATE lost+found DIRECTORY
and aborts the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

NO Abort the attempt to linkup the lost i-node. This will always invoke the UNREF error condition in phase 4.

NO SPACE LEFT IN /lost+found (EXPAND)
There is no space to add another entry to the **lost+found** directory in the root directory of the file system. When preening, the **lost+found** directory is expanded.

Possible responses to the EXPAND prompt are:

YES The **lost+found** directory is expanded to make room for the new entry. If the attempted expansion fails **ffsfsck**(1M) prints the message:
SORRY. NO SPACE IN lost+found DIRECTORY
and aborts the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4. Clean out unnecessary entries in **lost+found**. This error is fatal if the file system is being preened.

NO Abort the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

DIR I=*I1* CONNECTED. PARENT WAS I=*I2*
This is an advisory message indicating a directory i-node *I1* was successfully connected to the **lost+found** directory. The parent i-node *I2* of the directory i-node *I1* is replaced by the i-node number of the **lost+found** directory.

DIRECTORY *F* LENGTH *S* NOT MULTIPLE OF *B* (ADJUST)
A directory *F* has been found with size *S* that is not a multiple of the directory block size *B*. (This can recur in phase 3 if it is not adjusted in Phase 2.)

Possible responses to the ADJUST prompt are:

YES	The length is rounded up to the appropriate block size. Thus, when preening the file system, only a warning is printed and the directory is adjusted.
NO	Ignore the error condition.

BAD INODE *S* TO DESCEND
An internal error has caused an impossible state *S* to be passed to the routine that descends the file system directory structure. **ffsfsck**(1M) exits.

Phase 4 - Check Reference Counts

This phase is concerned with the link count information seen in phase 2 and phase 3. This section lists error conditions resulting from unreferenced files, a missing or full **lost+found** directory, incorrect link counts for files, directories, symbolic links or special files, unreferenced files, symbolic links and directories and bad or duplicate blocks in files, symbolic links, and directories. All errors in this phase are correctable if the file system is being preened, except running out of space in the **lost+found** directory.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)
I-node *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of i-node *I* are

printed. When preening, the file is cleared if either its size or its link count is zero. Otherwise, it is reconnected.

Possible responses to the RECONNECT prompt are:

YES	Reconnect i-node *I* to the file system in the directory for lost files (usually **lost+found**). This may invoke the **lost+found** error condition in Phase 4 if there are problems connecting i-node *I* to **lost+found**.
NO	Ignore this error condition. This will always invoke the CLEAR error condition in phase 4.

(CLEAR)
The i-node mentioned in the previous error condition cannot be reconnected. This cannot occur if the file system is being preened, since lack of space to reconnect files is a fatal error.

Possible responses to the CLEAR prompt are:

YES	Deallocate the i-node mentioned in the previous error condition by zeroing its contents.
NO	Ignore this error condition.

NO lost+found DIRECTORY (CREATE)
There is no **lost+found** directory in the root directory of the file system When preening, **ffsfsck**(1M) tries to create a **lost+found** directory.

Possible responses to the CREATE prompt are:

YES	Create a **lost+found** directory in the root of the file system. This may raise the message: **NO SPACE LEFT IN / (EXPAND)** See below for the possible responses. Inability to create a **lost+found** directory generates the message: **SORRY. CANNOT CREATE lost+found DIRECTORY** and aborts the attempt to linkup the lost i-node. This will always invoke the UNREF error condition in phase 4.
NO	Abort the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

lost+found IS NOT A DIRECTORY (REALLOCATE)
The entry for **lost+found** is not a directory.

Possible responses to the REALLOCATE prompt are:

YES Allocate a directory i-node, and change **lost+found** to reference it. The previous i-node reference by the **lost+found** name is not cleared. Thus, it will either be reclaimed as an UNREFed i-node or have its link count ADJUSTed later in this phase. Inability to create a **lost+found** directory generates the message:
SORRY. CANNOT CREATE lost+found DIRECTORY
and aborts the attempt to linkup the lost i-node. This will always invoke the UNREF error condition in phase 4.

NO Abort the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

NO SPACE LEFT IN /lost+found (EXPAND)
There is no space to add another entry to the **lost+found** directory in the root directory of the file system. When preening, the **lost+found** directory is expanded.

Possible responses to the EXPAND prompt are:

YES The **lost+found** directory is expanded to make room for the new entry. If the attempted expansion fails, **ffsfsck**(1M) prints the message:
SORRY. NO SPACE IN lost+found DIRECTORY
and aborts the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4. Clean out unnecessary entries in **lost+found**. This error is fatal if the file system is being preened.

NO Abort the attempt to link the lost i-node. This will always invoke the UNREF error condition in phase 4.

LINK COUNT *type* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)
The link count for i-node *I* is *X*, but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed. When preening the link count is

adjusted unless the number of references is increasing, a condition that should never occur unless precipitated by a hardware failure. When the number of references is increasing under preen mode, **ffsfsck**(1M) exits with the message:
LINK COUNT INCREASING

Possible responses to the ADJUST prompt are:

YES	Replace the link count of file i-node *I* with *Y*.
NO	Ignore this error condition.

UNREF *type* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
I-node *I*, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of i-node *I* are printed. When preening, this is a file that was not connected because its size or link count was zero. Hence, it is cleared.

Possible responses to the CLEAR prompt are:

YES	Deallocate i-node *I* by zeroing its contents.
NO	Ignore this error condition.

BAD/DUP *type* I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (CLEAR)
Phase 1 or phase 1b have found duplicate blocks or bad blocks associated with i-node *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of i-node *I* are printed. This error cannot arise when the file system is being preened, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES	Deallocate i-node *I* by zeroing its contents.
NO	Ignore this error condition.

Phase 5 - Check Cyl Groups

This phase is concerned with the free-block and used i-node maps. This section lists error conditions resulting from allocated blocks in the free-block maps, free blocks missing from free-block maps, and the total free-block count incorrect. It also lists error conditions resulting from free i-nodes in the used-i-node maps, allocated i-nodes missing from used i-node maps, and the total used i-node count incorrect.

CG *C*: BAD MAGIC NUMBER
The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group maps have been destroyed. When running manually, the cylinder group is marked as needing to be reconstructed. This error is fatal if the file system is being preened.

BLK(S) MISSING IN BIT MAPS (SALVAGE)
A cylinder group block map is missing some free blocks. During preening, the maps are reconstructed.

Possible responses to the SALVAGE prompt are:

YES	Reconstruct the free block map.
NO	Ignore this error condition.

SUMMARY INFORMATION BAD (SALVAGE)
The summary information was found to be incorrect. When preening, the summary information is recomputed.

Possible responses to the SALVAGE prompt are:

YES	Reconstruct the summary information.
NO	Ignore this error condition.

FREE BLK COUNT(S) WRONG IN SUPER-BLOCK (SALVAGE)
The super-block free block information was found to be incorrect. When preening, the super-block free block information is recomputed.

Possible responses to the SALVAGE prompt are:

YES	reconstruct the super-block free block information.
NO	ignore this error condition.

Cleanup

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

***V* files, *W* used, *X* free (*Y* frags, *Z* blocks)**
This is an advisory message indicating that the file system checked contained *V* files using *W* fragment sized blocks leaving *X* fragment sized blocks free in the file system. The numbers in parentheses break the free count down into *Y* free fragments and *Z* free full sized blocks.

******* REBOOT UNIX *******
This is an advisory message indicating that the root file system has been modified by **ffsfsck**(1M). If UNIX is not rebooted immediately, the work done by **ffsfsck**(1M) may be undone by the in-core copies of tables UNIX keeps. When preening, **ffsfsck**(1M) will exit with a code of 4. The standard auto-reboot script distributed with 4.3 BSD interprets an exit code of 4 by issuing a reboot system call.

******* FILE SYSTEM WAS MODIFIED *******
This is an advisory message indicating that the current file system was modified by **ffsfsck**(1M). If this file system is mounted or is the current root file system, **ffsfsck**(1M) should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by **ffsfsck**(1M) may be undone by the in-core copies of tables UNIX keeps.

3. BSD LP Spooler Tutorial

Chapter 3: BSD LP Spooler Tutorial

Introduction

This document describes the structure and installation procedure for the line printer spooling system developed for the 4.3 BSD version of the UNIX operating system.

Overview

The line printer system supports:

- Multiple printers
- Multiple spooling queues
- Local and remote printers
- Printers attached via serial lines that require line initialization such as the baud rate

The line printer system consists mainly of the following files and commands:

/etc/printcap	Printer configuration and capability database
/usr/lib/lpd	Line printer daemon, does all the real work
/usr/spool/lpr	Program to enter a job in a printer queue
/usr/spool/lpq	Spooling queue examination program
/usr/spool/lprm	Program to delete jobs from a queue
/etc/lpc	Program to administer printers and spooling queues
/dev/printer	Socket on which **lpd**(1M) listens

The file **/etc/printcap** is a master database describing line printers directly attached to a machine and printers accessible across a network. The manual page entry **printcap**(4) provides the authoritative definition of the format of this database, and specifies default values for important items such as the directory in which spooling is performed. This document introduces information that may be placed in **printcap**(4).

Commands

lpd - Line Printer Daemon

The program **lpd**(1M), usually invoked at boot time from the **/etc/rc2.d/lpr** file, acts as a master server for coordinating and controlling the spooling queues configured in the **printcap**(4) file. When **lpd**(1M) is started, it makes a single pass through the **printcap**(4) database, restarting any printers that have jobs. In normal operation, **lpd**(1M) listens for service requests on multiple sockets, one in the UNIX domain (named **/dev/printer**) for local requests, and one in the Internet domain (under the "printer" service specification) for requests for printer access from off-machine; see **socket**(2B) and **services**(4) for more information on sockets and service specifications, respectively. **lpd**(1M) spawns a copy of itself to process the request; the master daemon continues to listen for new requests.

Clients communicate with **lpd**(1M) using a simple transaction-oriented protocol. Remote clients are authenticated based on the "privilege port" scheme employed by **rcmd**(3B). The following table shows the requests understood by **lpd**(1M). In each request, the first byte indicates the "meaning" of the request, followed by the name of the printer to which it should be applied. Additional qualifiers may follow, depending on the request.

Request	Interpretation
^Aprinter\n	check the queue for jobs and print any found
^Bprinter\n	receive and queue a job from another machine
^Cprinter [users ...] [jobs ...]\n	return short list of current queue state
^Dprinter [users ...] [jobs ...]\n	return long list of current queue state
^Eprinter person [users ...] [jobs ...]\n	remove jobs from a queue

The **lpr**(1) command is used to enter a print job in a local queue and to notify the local **lpd**(1M) that new jobs are in the spooling area. **lpd**(1M) either schedules the job to be printed locally, or if printing remotely, attempts to forward the job to the appropriate machine. If the printer cannot be opened or the destination machine is unreachable, the job will remain queued until it is possible to complete the work.

lpq - Show Line Printer Queue

The **lpq**(1) program works recursively backward displaying the queue of the machine with the printer and then the queue(s) of the machine(s) that lead to it. **lpq**(1) has two forms of output: in the default, short, format it gives a single line of output per queued job; in the long format it shows the list of files and their sizes that compose a job.

lprm - Remove Jobs from a Queue

The **lprm**(1) command deletes jobs from a spooling queue. If necessary, **lprm**(1) will first kill a running daemon servicing the queue and restart it after the required files are removed. When removing jobs destined for a remote printer, **lprm**(1) acts similarly to **lpq**(1) except it first checks locally for jobs to remove and then tries to remove files in queues off-machine.

lpc - Line Printer Control Program

The system administrator uses the **lpc**(1M) program to control the operation of the line printer system. For each line printer configured in **/etc/printcap**, **lpc**(1M) may be used for the following:

- Disable or enable a printer
- Disable or enable a printer's spooling queue
- Rearrange the order of jobs in a spooling queue
- Find the status of printers and their associated spooling queues and printer daemons

Access Control

The printer system maintains protected spooling areas so that users cannot circumvent printer accounting or remove files other than their own. The strategy used to maintain protected spooling areas is as follows:

- The spooling area is writable only by a *daemon* user and daemon group.
- lpr(1) runs set-user-id to *root* and set-group-id to group daemon. The *root* access permits reading of any file required. Accessibility is verified with an **access**(2) call. The group ID is used to set up proper ownership of files in the spooling area for **lprm**(1).
- Control files in a spooling area are made with daemon ownership and group ownership daemon. Their mode is 0660. This ensures that a user does not modify control files and that no user can remove files except through **lprm**(1).
- The spooling programs, **lpd**(1M), **lpq**(1), and **lprm**(1) run set-user-id to *root* and set-group-id to group daemon to access spool files and printers.
- The printer server, **lpd**(1M), uses verification procedures to authenticate remote clients. The host on which a client resides must be in the file **/etc/hosts.equiv** or **/etc/hosts**. **lpd**(1M) and the request message must come from a reserved port number.

  In practice, **lpd**(1M), **lpq**(1), or **lprm**(1) would not have to run as user *root* if remote spooling were not supported.

Setting Up

The real work in setting up is creating the **printcap**(4) file and any printer filters for printers not supported in the distribution system.

Creating a Printcap File

The **printcap**(4) database contains one or more entries per printer. A printer should have a separate spooling directory; otherwise, jobs will be printed on different printers depending on which printer daemon starts first. This section describes how to create entries for printers that do not conform to the default printer description (an LP-11 style interface to a standard, band printer).

Printers on Serial Lines

When a printer is connected via a serial communication line, it must have the proper baud rate and terminal modes set. The following example is for a DecWriter™ III printer connected locally via a 1200-baud serial line.

```
lp|LA-180 DecWriter III:\
     :lp=/dev/lp:br#1200:\
     :tr=\f:of=/usr/lib/lpf:lf=/usr/adm/lpd-errs:
```

The **lp** entry specifies the file name to open for output. Here it could be omitted since **/dev/lp** is the default. The **br** entry sets the baud rate for the tty line (see **termio**(7S)). The **tr** entry indicates that a form feed should be printed when the queue empties so the paper can be torn off without turning the printer offline and pressing form feed. The **of** entry specifies that the filter program **lpf** should be used for printing the files; more will be said about filters later. The last entry causes errors to be written to the file **/usr/adm/lpd-errs** instead of to the console.

Remote Printers

Printers that reside on remote hosts should have an empty lp entry. For example, the following **printcap**(4) entry would send output to the printer named **lp** on the machine **ucbvax.**

```
lp|default line printer:\
    :lp=:rm=ucbvax:rp=lp:sd=/usr/spool/vaxlpd:
```

The **rm** entry is the name of the remote machine to connect to; this name must be a known host name for a machine on the network. The **rp** capability indicates that the name of the printer on the remote machine is **lp**; here it could be omitted since this is the default value. The **sd** entry specifies **/usr/spool/vaxlpd** as the spooling directory instead of the default value **/usr/spool/lpd.**

Output Filters

Filters are used to handle device dependencies and for accounting functions. The output filtering of **of** is used when accounting is not being performed or when all text data must be passed through a filter. It is not intended for accounting since it is started only once, all text files are filtered through it, and no provision is made for passing owners' login names, identifying the beginning and ending of jobs, etc. The other filters (if specified) are started for each file printed and for accounting if an **af** entry exists. If entries for both of and other filters are specified, the output filter is used only to print the banner page; it is then stopped to allow other filters access to the printer. An example of a printer that requires output filters is the Benson™-Varian.

```
va|varian|Benson-Varian:\
        :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
        :tf=/usr/lib/rvcat:mx#2000:pl#58:px=2112:py=1700:tr=\f:
```

The **tf** entry specifies **/usr/lib/rvcat** as the filter to be used to print **troff**(1) output. This filter is needed to set the device to print mode for text, and plot mode for printing *troff* files and raster images. The page length is set to 58 lines by the **pl** entry for 8.5" x 11" fan-fold paper. To enable accounting, the varian entry would be augmented with an **af** filter as shown below.

```
va|varian|Benson-Varian:\
        :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
        :if=/usr/lib/vpf:tf=/usr/lib/rvcat:af=/usr/adm/vaacct:\
        :mx#2000:pl#58:px=2112:py=1700:tr=\f:
```

NOTE **/usr/lib/vpf** and **/usr/lib/rvcot** are not supplied with **lpr**(1).

Access Control

Local access to printer queues is controlled with the **rg printcap**(4) entry.

```
    :rg=lprgroup:
```

Users must be in the group **lprgroup** to submit jobs to the specified printer. The default is to allow all users access. Once the files are in the local queue, they can be printed locally or forwarded to another host, depending on the configuration.

Remote access is controlled by listing the hosts in either the file **/etc/hosts.equiv** or **/etc/hosts.lpd**, with one host per line. **rcmd**(1) and **rcp**(1) use **/etc/hosts.equiv** to determine which hosts are equivalent for allowing logins without passwords. The file **/etc/hosts.lpd** is only used to control which hosts have line printer access. Remote access can be further restricted to only allow remote users with accounts on the local host to print jobs by using the **rs printcap**(4) entry.

```
        :rs:
```

Output Filter Specifications

Depending on the device and accounting methods, you may need to create a filter.

Filters are spawned by **lpd**(1M) with the data to be printed as their standard input, and the printer as their standard output. The standard error is attached to the **lf** file for logging errors. A filter must return a **0** exit code if no errors occurred, **1** if the job should be reprinted, and **2** if the job should be discarded. When **lprm**(1) sends a kill signal to the **lpd**(1M) process controlling printing, it sends a SIGINT signal to all filters and descendants of filters. This signal can be trapped by filters that need to do cleanup operations such as deleting temporary files.

Arguments passed to a filter depend on its type. The **of** filter is called with the following arguments.

filter **-w***width* **-l***length*

The ***width*** and ***length*** values come from the **pw** and **pl** entries in the **printcap**(4) database. The following parameters are passed to the **if** filter.

filter [**-c**] **-w***width* **-l***length* **-i***indent* **-n** *login* **-h** *host accounting_file*

The **-c** flag is optional and only supplied when control characters are to be passed uninterpreted to the printer (when using the **-l** option of **lpr**(1) to print the file). The **-w** and **-l** parameters are the same as they are for the **of** filter. The **-n** and **-h** parameters specify the login name and host name of the job owner. The last argument is the name of the accounting file from **printcap**(4).

All other filters are called with the following arguments:

filter **-x***width* **-y***length* **-n** *login* **-h** *host accounting_file*

The **-x** and **-y** options specify the horizontal and vertical page size in pixels (from the **px** and **py** entries in the **printcap**(4) file). The remaining arguments are the same as they are for the **if** filter.

NOTE

Filters are user-supplied executables. lpr(1) only provides pr and /usr/lib/lpf.

Initializing for the First Time

lpr(1) is delivered in the SYSTEMV product in a dormant state. To initialize **lpr**(1), enter the following as super-user:

```
/etc/init.d/lpr init
```

This enters **lpr**(1) in the initial startup for the system and starts the line printer daemon, **lpd**(1M).

If no Internet address exists for the machine, the initialization script will prompt the user for one. See the *BSD Network Configuration* procedure for further information about the format of an Internet address. If an Internet address is entered, the system must be rebooted to initialize the address.

Hereafter, the lpr printer daemon is automatically started after each system boot to listen for print requests.

Disabling the lpr Service Completely

After **lpr**(1) has been initialized, it can be removed from default startup routines by entering the following as super-user.

```
/etc/init.d/lpr disable
```

This removes **lpr**(1) from the startup script and
prevents the **lpr**(1) service from starting after a boot.
However, the user can still manually start the **lpd**(1M) print daemon
by entering the following
as the super-user.

```
/usr/lib/lpd
```

After the **lpr**(1) service is disabled, it must be initialized again as described above before the service will automatically start at boot time.

WARNING The concurrent use of more than one print service (lp, lpr(1), or NQS) can cause problems if more than one print service is using the same device.

Line Printer Control

The **lpc**(1M) program provides local control over line printer activity. The major commands and their intended uses will be described. The command format and remaining commands are described in **lpc**(1M).

abort and start

Abort terminates an active spooling daemon on the local host immediately and then disables printing (preventing new daemons from being started by **lpr**(1)). This is normally used to forcibly restart a hung line printer daemon. (**lpq**(1) reports that a daemon is present, but nothing is happening.) It does not remove jobs from the queue. (Use the **lprm**(1) command instead.) **Start** enables printing and requests **lpd**(1M) to start printing jobs.

enable and disable

Enable and **disable** allow spooling in the local queue to be turned on/off. This will allow/prevent **lpr**(1) from putting new jobs in the spool queue. It is frequently convenient to turn spooling off while testing new line printer filters since the *root* user can still use **lpr**(1) to put jobs in the queue, but no other user can. The other main use is to prevent users from putting jobs in the queue when the printer is expected to be unavailable for a long time.

restart

Restart allows ordinary users to restart printer daemons when **lpq**(1) reports that no daemon is present.

stop

Stop halts a spooling daemon after the current job completes; this also disables printing. This is a clean way to shut down a printer for maintenance, etc. Users can still enter jobs in a spool queue while a printer is **stopped**.

topq

Topq places jobs at the top of a printer queue. This can be used to reorder high-priority jobs since **lpr**(1) only provides first-come-first-serve ordering of jobs.

Troubleshooting

Several messages may be generated by the line printer system. This section categorizes the most common and explains the causes for their generation. When the message implies a failure, directions are given to remedy the problem.

In the examples below, the name *printer* is the name of the printer from the **printcap**(4) database.

LPR Error Messages

lpr: *printer* **: unknown printer**

The *printer* was not found in the **printcap**(4) database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the **/etc/printcap** file.

lpr: *printer* **: Jobs queued, but cannot start daemon.**

The connection to **lpd**(1M) on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket **/dev/printer** to ensure that it still exists. (If it does not exist, no **lpd**(1M) process is running). Usually a super-user typing the following will restart **lpd**(1M):

```
/usr/lib/lpd
```

You can also check the state of the master printer daemon with the following:

```
ps -p `cat /usr/spool/lpd.lock`
```

Another possibility is that the **lpr**(1) program is not set-user-id to *root*, set-group-id to group daemon. This can be checked by entering the following:

```
ls -lg /usr/spool/lpr
```

lpr: ***printer*****: printer queue is disabled**

This means the queue was turned off with the following to prevent **lpr**(1) from putting files in the queue.

```
lpc disable printer
```

This is normally done by the system manager when a printer will be down for a long time. The printer can be turned back on by a super-user with **lpc**(1M).

LPQ Error Messages

waiting for ***printer*** **to become ready (offline ?)**

The printer device could not be opened by the daemon. This can happen for several reasons; the most common is that the printer is turned offline. This message can also be generated if the printer is out of paper or the paper is jammed, etc. The actual reason depends on the meaning of error codes returned by the system device driver. Not all printers supply enough information to distinguish when a printer is offline or having trouble (such as a printer connected through a serial line). Another possible cause of this message is that some other process, such as an output filter, has an exclusive lock on the device. Your only recourse here is to kill the offending program(s) and restart the printer with **lpc**(1M).

printer **is ready and printing**

The **lpq**(1) program checks to see if a daemon process exists for *printer* and prints the file **status** located in the spooling directory. If the daemon is hung, a super-user can use **lpc**(1M) to abort the current daemon and start a new one.

waiting for ***host*** **to come up**

This implies that a daemon is trying to connect to the remote machine named *host* to send the files in the local queue. If the remote machine is up, **lpd**(1M) on the remote machine is probably dead or hung and should be restarted as mentioned for **lpr**(1).

sending to *host*

The files should be in the process of being transferred to the remote *host*. If not, the local daemon should be aborted and started with **lpc**(1M).

Warning: *printer* is down

The printer has been marked as being unavailable with **lpc**(1M).

Warning: no daemon present

The **lpd**(1M) process overseeing the spooling queue, as specified in the **lock** file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer should be checked for a diagnostic from the deceased process. To restart an **lpd**(1M), use the following:

```
lpc restart printer
```

no space on remote; waiting for queue to drain

This implies that insufficient disk space is on the remote. If the file is large enough, there will never be enough space on the remote (even after the queue on the remote is empty). The solution is to move the spooling queue or make more free space on the remote.

LPRM Error Messages

lprm: *printer*: cannot restart printer daemon

This case is the same as when **lpr**(1) prints that the daemon cannot be started.

LPD Error Messages

The **lpd**(1M) program can log many different messages on the console. Most of these messages are about files that cannot be opened and usually imply that the **printcap**(4) file or the protection modes of the files are incorrect. Files may also be inaccessible if people manually manipulate the line printer system. (They bypass the **lpr**(1) program.)

In addition to messages generated by **lpd**(1M), any of the filters that **lpd**(1M) spawns may log messages using the error log file (the file specified in the **lf** entry in **printcap**(4)).

LPC Error Messages

couldn't start printer

This case is the same as when **lpr**(1) reports that the daemon cannot be started.

cannot examine spool directory

Error messages beginning with "cannot" usually result from incorrect ownership or protection mode of the lock file, spooling directory, or **lpc**(1M) program.

4. NQS Tutorial

Chapter 4: NQS Tutorial

NQS Concepts

NQS is a network-based, multipurpose printing, plotting, and batch- queuing resource for Intergraph CLIPPER workstations and InterServe processors.

Network Queuing System (NQS) Features

NQS is a batch- and device-queuing facility supporting job requests in a networked environment composed of CLIPPER workstations and/or InterServe processors. The nodes in an NQS network must be running the CLIX 3.1 operating system (or later version).

Convenience and Security

NQS provides a convenient and secure solution to network printing, plotting, and batch queuing. Using NQS, you can submit print requests, plots, or batch jobs to your local node or to a remote node. You gain access to other NQS nodes when the remote node's owner *map*s your node name and user name to a user name on the remote node. You are mapped to another node when you are added to the list of users that can submit requests to the remote node.

Clients and Servers

An NQS network is composed of clients (nodes that submit requests) and servers (nodes that receive and service requests). A client must have access to a server before it can submit jobs. This access restriction allows you to determine the users who can access your node. (A node can be both a client and server.)

Understanding Queues

A queue is a waiting line in which job requests reside until they are processed. NQS supports these three queue types.

- Device queues
- Batch queues
- Pipe queues

Each queue type is discussed briefly below.

Device Queues

A device queue routes requests to a printer or plotter. Device queues exist only on NQS server nodes because only server nodes are connected to output devices. A device queue may service any number of devices and several queues may service the same device. Each request in a device queue awaits its turn for the next available device serviced by that queue.

Pipe Queues

A pipe queue routes requests to another queue the way a device queue routes requests to an output device. Pipe queues can accept any type of request: a batch request, a device request, or a request from another pipe queue.

A pipe queue interprets the request and pipes it to the first available queue that is in its destination list and that will accept the request. The destination list may contain any queue type. You must use pipe queues to submit jobs to remote nodes.

Batch Queues

A batch queue accepts requests to execute shell scripts or commands from the keyboard. The batch request executes as if the requester had logged in. The resulting output returns to the original requester's current directory when the job was submitted. The output may remain on the executing node or return to a remote node at your request.

Understanding Devices

Devices in NQS are the interfaces to the printers and plotters from which hardcopy output is produced. The queues discussed in the previous section service these devices. You must create a device interface for every physical printer or plotter. (The "Manipulating Devices From the Command Line" section describes how to create devices.) However, when you create a device queue with the **pconfig** utility, a device interface is created automatically. (See the *Network Queuing System (NQS) User's Guide* for more information on **pconfig**)

Device Capabilities

The capabilities for each NQS device are stored in a *devicecap* file. A devicecap file contains the default behavior, environment, and permitted input types for the device that it serves. A devicecap file must be defined for each device you create. However, more than one device can use the same devicecap file.

NOTE When you create a device queue with the **pconfig** utility, an appropriate devicecap file is created automatically.

Understanding Forms

A form is a tag associating a name with a specific device configuration. The most common use for forms is for specifying the type of paper that your print request will print on. A form might also specify a switch set on a printer.

Forms are specified in two places:

- Through the Queue Manager **qmgr**(1M)
- In the command line that submits your request

NQS does not understand form types and will not configure your devices differently for specific forms. Therefore, you can assign any form name to any type characteristic. NQS uses form names only by ensuring that the form name in your request matches the form type allowed for that device. Using forms is optional; if no form is specified, NQS will print on whatever form exists for the device.

Because NQS does not designate form names, it is helpful to assign logical names to your forms. Below are several suggestions for naming forms:

- Sysform is generally the default form (the form most frequently used at your site).
- The form land10 might represent a landscape form with a type style of 10 characters-per-inch.

The Queue Manager Utility (qmgr)

The NQS Queue Manager (**qmgr**(1M)) utility allows the system manager or operator to control NQS configuration and operation. Using **qmgr**(1M), you can (if you have system privileges) manipulate NQS requests, queues, and devices. (System privileges allow you to perform NQS maintenance tasks that other users cannot perform.)

Because **qmgr**(1M) performs a wide variety of functions, these functions are explained in context in the appropriate sections of this tutorial. The following list describes a subset of the functions that can be performed in **qmgr**(1M).

- Assign NQS privileges. (See the "NQS Privileges" section.)
- Display status information using the show commands. (See the "Displaying Status Information" section of the Programmer's & User's "NQS Tutorial".)
- Create and manage queues from the command line. (See the "Configuring Queues from the Command Line" section.)
- Create and manage devices from the command line. (See the "Manipulating Devices from the Command Line" section.)

When you first install NQS, **qmgr**(1M) access is limited to the super-user, who assigns initial NQS privileges. Other users can later change NQS configuration or operations only if a user with manager privileges assigns them manager or operator privileges. Any user can obtain status information using **qmgr**(1M).

Accessing the Queue Manager

When NQS is first installed, gain access to **qmgr**(1M) by keying it in at the super-user prompt. The Mgr: prompt displays afterwards as follows:

```
# qmgr
Mgr:
```

After NQS privileges have been assigned, privileged users can key in **qmgr** at the system prompt to display the Mgr: prompt:

```
$ qmgr
Mgr:
```

NOTE

Unprivileged users may enter qmgr(1M) but will receive an "insufficient privileges" message if they attempt to perform operations requiring privileges.

Keying in Commands in qmgr

Queue manager commands must be executed through **qmgr**(1M). Long command strings that do not fit on one line can be continued on a second line by keying in a backslash (\) at the end of the first line. (Key in a space before the backslash only if a space is needed in the command line.) The system will return an Mgr:_ prompt to let you know that you may finish keying in the command.

For example, the following command is valid in **qmgr**(1M):

```
Mgr: show long \
Mgr:_ queue
```

Getting Help in qmgr

qmgr(1M) has a help facility that provides information about **qmgr**(1M) commands. To use the help facility, key in **h** or **help** at the Mgr: prompt and a help screen displays.

```
Mgr: help
The available commands are:

ABort     ADd       Create  DElete    DIsable  ENable
EXit      Help      HOld    Lock      MODify   MOVe
Purge     RELease   REMove  SEt       SHOw     SHUtdown
STArt     STOp      UNHold  UNLock

To obtain more information about a command, type:

Help command-name
```

If you key in **help** alone as shown in the example above the list of available commands displays. The capital letters in each command indicate the minimum number of letters that must be keyed in to execute that command. All **qmgr(1M)** commands can be abbreviated to the minimum number of characters that uniquely identifies them. If you key in **help** followed by a command name, help for that command displays as follows:

```
Mgr: help show queue

  The comand:
    SHow Queue [ <queue-name> [ <user-name> ] ]
  is used to display status information about NQS queues in a
  short format.

  If no queue name is specified, then status information is
  displayed for all NQS queues.

  If a queue name is specified, then specific status information
  for the named queue is displayed, including the ordering of
  requests within the queue.  If a user name is specified, then
  the requests shown will be limited to those belonging to that
  user.
```

Exiting the Queue Manager

To exit **qmgr**(1M), key in exit at the Mgr: prompt:

```
Mgr: exit
$
```

Command Overview

Six NQS commands are available outside the **qmgr**(1M) environment. Any NQS user can key in these commands from the system prompt:

- **qsub**(1) is used to submit batch requests. The "Submitting and Deleting NQS Requests" section in the Programmer's & User's "NQS Tutorial" details the **qsub**(1) command.
- **qpr**(1) is used to submit print requests. The "Submitting and Deleting NQS Requests" section in the Programmer's & User's "NQS Tutorial" details the **qpr**(1) command.
- **qdel**(1) is used to delete and signal requests. The "Submitting and Deleting NQS Requests" section in the Programmer's & User's "NQS Tutorial" details the **qdel**(1) command.
- **qstat**(1) is used to display queue status without needing to enter **qmgr**(1M). The "Displaying Status Information" section in the Programmer's & User's "NQS Tutorial" details the **qstat**(1) command.
- **qdev**(1) is used to display device status. The "Submitting and Deleting NQS Requests" section in the Programmer's & User's "NQS Tutorial" details the **qdev**(1) command.
- **qlimit**(1) is used to display NQS limits. The "Displaying Status Information" section in the Programmer's & User's "NQS Tutorial" details the **qlimit**(1) command.

NQS Privileges

Every NQS user has a certain privilege level. These privilege levels are *manager*, *operator*, and nonprivileged *user*. This section discusses these privilege levels and how to assign them using **qmgr**(1M).

Privilege Classifications

NQS recognizes the following privilege classes:

- Nonprivileged users
- Operators
- Managers

Nonprivileged users can submit requests to unrestricted queues and display status information about requests. They cannot affect queue control or change the NQS configuration. Most users are in this category.

Operators can access commands that directly affect queue control and request management (such as starting and stopping queues and deleting requests). They cannot change the NQS configuration.

Managers have all available NQS privileges. Only managers can change the NQS configuration (such as creating queues and defining default parameters).

Assigning and Removing Privileges

Only a person with manager privileges or the super-user can use **qmgr**(1M) to assign privileges to other NQS users. The following sections describe the commands that a manager can use to assign and remove user privileges.

The add managers Command

The **add managers** command is used to assign manager or operator privileges to one or more users. You must invoke **qmgr**(1M) to use the **add managers** command.

```
$ qmgr
Mgr:
```

To give manager privileges to one or more users, key in the **add managers** command as follows.

Mgr: **add managers** *username*:**m** [*username*:**m** ...]

To give operator privileges to one or more users, key in the **add managers** command as follows:

Mgr: **add managers** *username*:o [*username*:o...]

- You can assign managers and operators on the same line.
- Root is always listed as a manager and cannot lose privileges.
- A user must have an account on your workstation or server to be given manager or operator privileges.
- Managers have all privileges that operators have; however, operators have only a subset of the privileges that managers have.

The delete managers Command

The **delete managers** command removes manager and operator privileges from the specified users. You must invoke **qmgr**(1M) to use the **delete managers** command. To remove manager privileges from one or more users, key in the following:

Mgr: **delete managers** *username*:**m** [*username*:**m**...]

To remove operator privileges from one or more users, key in the following:

Mgr: **delete managers** *username*:o [*username*:o...]

NOTE Root is always listed as a manager and cannot lose privileges.

The set managers Command

The **set managers** command removes manager and operator privileges from all existing managers and operators (except root) and reassigns these privileges to the specified users.

To define a new set of managers and operators, key in the **set managers** command as follows:

Mgr: **set managers** *username*:**m** [*username*:o *username*:**m**...]

Obtaining Access to Other Nodes

To submit jobs to or display status information on NQS nodes (other than your local node), you must have access to these nodes. To gain access, the system manager of the node you want to access must add your account to the **/etc/hosts.nqs** file on the node. (The Berkeley method of mapping users through **.rhosts** and **/etc/hosts.equiv** can also be used.)

The **hosts.nqs** file is explained below.

The hosts.nqs File

If during the NQS delivery procedure you specified that you wanted to map all remote users to the rje account, a default **hosts.nqs** file was placed in your **/etc** directory. If you did not accept the default, a model **hosts.nqs** file was placed in the product directory, **/usr/ip32/nqs**. You can copy this model to your **/etc** directory and modify it using any editor supported on your workstation.

NOTE The default and model hosts.nqs files contain helpful information about how to map users to your node. This information is commented out and will not interfere with your mappings.

Remember that the **hosts.nqs** file on your node determines only who can access your node. The information in the **hosts.nqs** files on other nodes determines whether you can access them.

Format for the hosts.nqs File

The **hosts.nqs** file contains a list of remote nodes that have NQS access on the local node. Users on remote nodes map to users on the local node. Here is the basic format for the **hosts.nqs** file:

```
remote_node     remote_user
remote_node     remote_user=local_user
# comments
```

These terms are defined as follows:

remote_node	name of the remote node that has access or * (all remote nodes have access)
remote_user	user name on remote node or * (all users)
# comments	Any lines preceded by the # sign are comments that NQS does not read.

NOTE

You may use the wildcard character (*) in place of any remote node name or remote user name in the hosts.nqs file.

Modifying the hosts.nqs File

If you need to modify the **hosts.nqs** file, use the following examples as a guideline:

- The following line is the default entry in the **hosts.nqs** file. This line maps any user to the **rje** account on this node.

  ```
  *  *=rje
  ```

- The next line gives user **fred** on node **red** access to the local node. With this mapping, when **fred** submits a job to this node, he will be logged in as **joe**.

  ```
  red fred=joe
  ```

- The next line gives any user on node **yellow** access to the local node. When users from node **yellow** submit jobs to this node, they will map in as **guest**.

  ```
  yellow *=guest
  ```

- The following line in a **hosts.nqs** file maps all users by name to the local node. Use this mapping with extreme caution, as it provides no security for your local accounts.

  ```
  *
  ```

Creating and Managing Queues from the Command Line

This chapter provides instructions on how to create and manipulate queues directly from the command line using NQS's queue manager utility, **qmgr**(1M).

NOTE

You must have NQS manager privileges to create and make changes to queues. See the "NQS Privileges" section.

NOTE

The commands in this chapter must be executed from the Mgr: prompt. The Mgr: prompt indicates that you are in the **qmgr**(1M) environment. To access the Mgr: prompt, key in **qmgr** at the system prompt as follows:

```
$ qmgr
Mgr:
```

Creating and Manipulating Queues

The following sections describe the commands used to create and manipulate queues from the **qmgr**(1M) command line.

NOTE

You can continue long command strings that do not fit on one line on a second line by keying in a backslash (\) at the end of the first line. (Key in a space before the backslash only if a space is needed in the command line.) The system will return an Mgr:_ prompt to notify you that you may finish keying in the command.

Creating Batch Queues

Batch queues are used to submit batch requests. Batch-processing requests are shell-scripts or other program files that will execute as batch jobs on the local node or on a remote server.

To create a batch queue, key in the **create batch_queue** command at the Mgr: prompt as follows:

```
Mgr: create batch_queue queue_name priority=n [pipeonly ]\
Mgr:_ [run_limit=n ]
```

- The required queue name contains any printable character except for a space, @, comma (,), and = and may not begin with a number or a dash (-).
- The required priority determines which queues will be checked first for requests. (Queues with higher priority values will be checked first.) The *n* value is a number between 0 and 63.
- The optional pipeonly parameter restricts queue requests to those routed from a pipe queue.
- The optional run_limit is a limit on the number of jobs that may execute simultaneously in the queue. The *n* value is the job-number limit.

Batch Queue Examples

The examples below illustrate several batch queue configurations:

- The queue named sysbatch has the simplest queue configuration. Requests in this queue will be processed one-at-a-time by default.

  ```
  Mgr: create batch_queue sysbatch priority=16
  ```

- The queue named batchpipe has the same configuration as sysbatch, except that it will only accept requests coming from a pipe queue (running either locally or on another server node.)

  ```
  Mgr: create batch_queue batchpipe priority=16 pipeonly
  ```

- The queue named batchquick has a higher priority than sysbatch, so it will be checked first for requests. Batchquick has a run limit of five; only five requests may run simultaneously in the queue.

  ```
  Mgr: create batch_queue batchquick priority=20 run_limit=5
  ```

Creating Device Queues

Device queues schedule requests to NQS devices. NQS devices are the printer and plotter interfaces that produce the hardcopy output requested by NQS.

To create a device queue, key in the **create device_queue** command as follows:

> Mgr: **create device_queue** *queue_name* **priority=***n*\
> Mgr:_ [**device=(***device,device* . . .] \
> Mgr:_ [**pipeonly**]

- The required queue name contains any printable character except for a space, @, comma (,), and = and may not begin with a number or a dash (-).
- The required priority determines which queues will be checked first for requests. (Queues with higher priority values will be checked first.) The *n* value is a number between 0 and 63.
- The optional device parameter specifies the devices serviced by the queue. Devices must be created before NQS recognizes them as valid; see the "Manipulating Devices from the Command Line" section.) Enclose the device list in parentheses and separate the devices by commas. (If only one device is in the list, you may omit the parentheses.) NQS will search for the devices when requests are submitted from the queue to the printer. Devices are checked in the order in which they appear in the list. Therefore, the first device in the list will be used most frequently.

NOTE Do not confuse the create device_queue command with the create device command discussed in the "Manipulating Devices from the Command Line" section.

- The optional **pipeonly** parameter restricts queue requests to those routed from a pipe queue.

Device Queue Examples

The examples below illustrate several device queue configurations:

NOTE In the examples below, the devices called printronix, lp11, and dotmat must have been previously defined to NQS before their servicing queues can be created. The "Manipulating Devices from the Command Line" section details devices.

- The device queue **sysprint** has a priority of 16 and services the device called **printronix**. Since **sysprint** services only one device, requests will execute sequentially.

  ```
  Mgr: create device_queue sysprint priority=16\
  Mgr:_ device=printronix
  ```

- The device queue **laser** has the same priority as **sysprint** but services a device called **lp811**.

  ```
  Mgr: create device_queue laser priority=16 device=lp811
  ```

- The device queue **dotmat** also has a priority of 16 but services three devices: **epson, fujitsu**, and **printronix**. When requests are queued to the printer, the NQS scheduler will check for the first available device beginning with the first device in the device list. Therefore, the epson printer will be used more frequently than the fujitsu printer will because the epson will always be checked first. If both devices are busy, the request will be printed on the printronix device so long as another queue to that device has not already requested it. When all three devices are busy, the request will be scheduled for the first device in the list that becomes available.

  ```
  Mgr: create device_queue dotmat priority=16 device=(epson,\
  Mgr:_ fujitsu,printronix)
  ```

Creating Pipe Queues

Pipe queues are the most versatile queues. Pipe queues route requests to batch or device queues or to other pipe queues. Any request sent to a remote server must be routed through a pipe queue.

To create a pipe queue, key in the **create pipe_queue** command as follows:

```
Mgr: create pipe_queue queue_name priority=n\
Mgr:_ server=(/usr/lib/nqs/pipeclient)\
Mgr: [destination=(queue,queue@host,...)] [pipeonly] [run_limit=n]
```

- The required queue name contains any printable character except for a space, @, comma (,), and = and may not begin with a number or a dash (-).
- The required priority determines which queues will be checked first for requests. (Queues with high priority values will be checked first.) The *n* value is a number between 0 and 63.
- The server is always the queue server process, **/usr/lib/nqs/pipeclient**.
- The optional destination allows you to specify the local or remote destinations that requests can be routed to. The *queue@host* list names destination queues separated by commas and enclosed in parentheses. If only one destination queue is in the list, you may omit the parentheses.
- The optional **pipeonly** parameter restricts queue requests to those routed from another pipe queue.
- The optional **run_limit** parameter allows you to specify a limit on the number of jobs that may be routed simultaneously through that queue.

Pipe Queue Examples

The examples below illustrate several pipe queue configurations:

- The queue **syspipe** has a basic pipe queue configuration. The two destinations are local queues **sysbatch** and **sysprint**. Jobs submitted to this queue will execute on the local node.

  ```
  Mgr: create pipe_queue syspipe priority=16\
  Mgr:_ destination=(sysbatch,sysprint)\
  Mgr:_ server=(/usr/lib/nqs/pipeclient)
  ```

- The queue **to_jim** executes requests on the remote server named **jim**. This is the most common configuration for a queue that executes requests on a remote host.

```
Mgr: create pipe_queue to_jim priority=16\
Mgr:_ destination=(sysbatch@jim, sysprint@jim)\
Mgr:_server=(/usr/lib/nqs/pipeclient)
```

- The queue **to_print** will route requests to the first available destination queue that will accept the request, specifically, the device queues **sysprint** at the local host, **sysprint** at the node **jim**, or **sysprint** at the node **kathy**.

  ```
  Mgr: create pipe_queue to_print priority=16\
  Mgr:_ destination=(sysprint,sysprint@jim,\
  Mgr:_ sysprint@kathy) server=(/usr/lib/nqs/pipeclient)
  ```

  The queue **routepipe** would be useful in an environment where all other queues on the local workstation or server are restricted. (See the "Restricting Queue Access" section.) In this case, all requests coming from remote clients would be forced through **routepipe** so that the local system can route them to another local queue.

  ```
  Mgr: create pipe_queue routepipe priority=16\
  Mgr: pipeonly destination=(sysbatch,\
  Mgr:_ sysprint) server=(/usr/lib/nqs/pipeclient)
  ```

Creating a Queue Complex

Queue complexes allow you to group queues for setting parameters that affect the entire complex. Queue complexes are primarily used to limit the number of jobs that a batch or device queue complex will service at a time. This is useful for ensuring that jobs to different devices on a mux do not collide. Key in the **create complex** command as follows:

Mgr: **create complex =(** *queue_name* [*,queue_name,...*]) *complex_name*

Adding Queues to a Queue Complex

Add queues to a queue complex using the **add queue** command. Key in the command as follows:

Mgr: **add queue =** (*queue_name* [*,queue_name,...*]) *complex_name*

Removing Queues from a Queue Complex

Remove queues from a queue complex using the **remove queue** command. Key in the command as follows:

Mgr: **remove queue** = (*queue_name* [,*queue_name*,..]) *complex_name*

Enabling and Disabling Queues

A queue is disabled when it is created. Queues can be enabled or disabled in **qmgr**(1M).

Disabling a queue prevents requests from being submitted to the queue; it does not prevent jobs in the queue from running. If you disable a queue while it is running, jobs already in the queue will execute. Therefore, it may be necessary to stop the queue before disabling it.

- An enabled queue is ready to accept requests. Enable a queue by keying in the following:

  Mgr: **enable queue** *queue_name*

- A disabled queue will not accept requests. Disable a queue by keying in the following:

  Mgr: **disable queue** *queue_name*

Saving and Restoring Queue Configuration

The **qsave** and **qrestore** commands save and restore the current NQS configuration.

The **qsave** command saves the current NQS queue configuration in a file and can be performed after any queue is created. The **save** command is normally used after the system manager has configured all queues. Key in the **qsave** command at the super-user prompt as follows:

qsave *file_name*

The **qrestore** command restores any previously saved queue configurations, but will not restore requests held in the queue.
Key in the **qrestore** command at the super-user prompt as follows:

qrestore *file_name*

The queue configuration can only be restored when NQS is shut down. The following command sequence can be used to shut down NQS, restore a queue, and restart NQS:

```
# qmgr
Mgr: shutdown
Mgr: exit
# qrestore file_name
# /etc/init.d/nqs start
```

Starting, Stopping, and Restarting Queues

A queue is stopped when it is first created. Queues can be started and stopped in **qmgr**(1M). A started queue may be running or inactive.

- A started queue is running if it is servicing requests. To start a queue, key in the following:

  Mgr: **start queue** *queue_name*

- A started queue is inactive if it is ready to service requests.
- A stopped queue will accept but not execute requests. You may want to stop a queue to fix a device or change paper.

NOTE Requests in a stopped queue will remain there indefinitely (or for the time limit specified in the **set lifetime** command discussed in the "set lifetime Command" section.

To stop a queue, key in the following:

Mgr: **stop queue** *queue_name*

Restarting a stopped queue causes running jobs to be killed and rescheduled. Restarting is useful when you want to reprint the job that is currently running. To restart a queue that has been stopped, key in the following:

Mgr: **restart queue** *queue_name*

Changing the Queue Priority (All Queue Types)

Queue priorities determine the order in which NQS checks queues for requests. When you create a pipe or device queue with **pconfig**, the queue priority defaults to 31.

Change the priority value for any existing queue by keying in the following, where *n* is the priority value from 0-63:

Mgr: **set priority=***n queue_name*

Changing the Queue Run Limit (Batch and Pipe Queues)

The run limit value is the number of jobs that can execute simultaneously in the named batch queue or the number of requests that can be routed simultaneously through the named pipe queue.

Change the run limit for an existing queue by keying in the following, where *run_limit* is the maximum number of requests allowed in that queue.

Mgr: **set run_limit=***run_limit queue_name*

Moving Requests and Queues

Moving requests and/or queues can be useful in rerouting jobs from a printer/plotter to another device.

Use the **qstat**(1) command to determine the request ID of the job to be moved and then use the **move** command to move one request from one queue to another. The format of the **move** command is as follows:

Mgr: move *request_id destination_queue_name*

Moving a queue moves all requests in the queue. Key in the following to move a queue:

Mgr: move queue *source_queue_name destination_queue_name*

Modifying Queue Destination Lists (Pipe Queues Only)

You may add, delete, or set new destinations for an existing pipe queue. The following sections provide instructions for modifying the queue destination list.

NOTE Pipe queues created in **pconfig** have only one destination, which is always a device queue.

Adding a Destination to a Pipe Queue

To add a destination queue to the end of a pipe queue's destination list, key in the **add destination** command as follows. Destination queues on remote nodes must have the format *queue_name@host*.

Mgr: **add destination=(***dest_queue*,[*dest_queue*,...]) *queue_name*

Deleting a Destination from a Pipe Queue

To delete a destination from a pipe queue's destination list, key in the **delete destination** command as follows. Destination queues at remote locations must have the format *queue_name@host*.

Mgr: **delete destination=(***dest_queue*,[*dest_queue*,...]) *queue_name*

Creating a New Destination List for a Pipe Queue

Since destinations are placed in the destination list in priority order, you may sometimes need to replace an existing destination list with a new one. To create a new destination list, key in the **set destination** command as follows. Destination queues at remote locations must have the format *queue_name@host*.

Mgr: **set destination=(***dest_queue*,[*dest_queue*,...]) *queue_name*

Restricting Queue Access

When you create any queue, it defaults to unrestricted access. This means that any user with access to NQS can submit requests to the queue. You can, however (if you have NQS manager privileges), restrict queue access to certain groups or users.

To restrict access to a queue, you must first restrict all access to the queue and then add users as desired. Groups and users may be deleted later if necessary. Follow steps 1 and 2 below to restrict queue access to specific users and groups.

Step 1: Restrict All Access to the Queue

Restrict all access to a queue by keying in the **set no_access** command as follows:

Mgr: **set no_access** *queue_name*

Step 2: Add Specific Users and Groups

- Add users as desired by keying in the **add users** command as follows:

  Mgr: **add users=(***username*,[*username* . . .]) *queue_name*

  The *username*(s) are the names of users who will have access to the queue. Multiple *usernames* are separated by commas and enclosed in parentheses. If there is only one *username*, you may omit the parentheses.

- Add groups as desired by keying in the **add groups** command as follows:

  Mgr: **add groups** (*groupname*,[*groupname* . . .]) *queue_name*

  The *groupname*(s) are the names of groups that will have access to the queue. The *groupnames* specified are the default log-in groups defined in the **/etc/passwd** file. Multiple *groupnames* are enclosed in parentheses and separated by commas. If there is only one *groupname*, you may omit the parentheses.

NOTE If a user is a member of a group that you have added, you do not have to add that user with the **add users** command.

Deleting Users and Groups from a Queue

After you have established users and groups for a queue, you may need to restrict one or more of those users and groups from that queue. To delete users from a queue, key in the **delete users** command as follows:

Mgr: **delete users=(***username*,[*username* . . .]) *queue_name*

The *username*(s) are the names of users who will be restricted from using the queue. Multiple *usernames* are separated by commas and enclosed in parentheses. If there is only one *username*, you may omit the parentheses. To delete groups from a queue, key in the **delete groups** command as follows:

Mgr: **delete groups=(***groupname*,[*groupname* . . .]) *queue_name*

The *groupname*(s) are the names of groups (specified in your **/etc/group** file) that will be restricted from the queue. Multiple *groupnames* are enclosed in parentheses and separated by commas. If there is only one *groupname*, you may omit the parentheses.

Restoring Unrestricted Access to Queues

If you want to restore unrestricted access to a restricted queue, key in the **set unrestricted_access** command as follows:

Mgr: **set unrestricted_access** *queue_name*

Deleting Queues

Before a queue can be deleted, it must be disabled and inactive. Use the **show queue** command to ensure that the queue is disabled and inactive before deleting it. These are the steps to delete a queue:

1. Ensure that the queue you want to delete is inactive. Check the queue using the **show queue** command.

   Mgr: **show queue**

Information similar to this should display. Note the queue's inactive status.

```
sysbatch@olga; type=BATCH; [ENABLED, INACTIVE]; pri=16
 0 exit; 0 run; 0 stage; 0 queued; 0 wait; 0 hold; 0 arrive;
```

2. Disable the queue as follows:

   Mgr: **disable queue** *queue_name*

   You may want to use the **show queue** command again to ensure that the queue is disabled.

   Mgr: **show queue**

```
sysbatch@olga; type=BATCH; [DISABLED, INACTIVE]; pri=16
 0 exit; 0 run; 0 stage; 0 queued; 0 wait; 0 hold; 0 arrive;
```

3. Key in the **delete queue** command as follows:

   Mgr: **delete queue** *queue_name*

Creating and Managing Devices from the Command Line

This section provides instructions on how to create and manipulate devices directly from the command line using NQS's queue manager utility, qmgr(1M).

NOTE The commands in this chapter must be executed from the qmgr(1M) utility. The Mgr: prompt indicates that you are in the qmgr(1M) environment. To access the Mgr: prompt, key in **qmgr** at the system ($) prompt as follows:

```
$ qmgr
Mgr:
```

Creating Devices

Devices in NQS are the interfaces to the printers and plotters on which hardcopy output is produced. The queues discussed in the previous section service these devices. An NQS device must be created for every physical printer or plotter.

To create a device, key in the following:

```
Mgr: create device device_name forms=forms fullname=pathname\
Mgr: server=(/usr/lib/nqs/devserver [-n comment ]\
Mgr:_ /usr/lib/nqs/config_files/devicecap_file)
```

You must specify all parameters, except for the **-n** comment parameter, when creating a device. These parameters are described as follows:

- The *device_name* can be any name that describes the device.
- The *form* mounted on the device can be any form that you have defined to NQS.
- The full *pathname* for the device may be one of the following:

For a serial device connected to...	Key in...
The RS-232 port	/dev/tty00
RS-232 auxiliary port 0	/dev/tty00
RS-232 auxiliary port 1	/dev/tty01
RS-232 auxiliary port2	/dev/tty02

For a parallel device connected to...	Key in...
Plotter port (versatec-type device)	/dev/vop
Mux port 1 (versatec-type device)	/dev/vop1
Mux port 2 (versatec-type device)	/dev/vop2
Mux port 3 (versatec-type device)	/dev/vop3
Mux port 4 (versatec-type device)	/dev/vop4
Parallel port (centronix-type device)	/dev/cop
Mux port 1 (centronix-type device)	/dev/cop1
Mux port 2 (centronix-type device)	/dev/cop2
Mux port 3 (centronix-type device)	/dev/cop3
Mux port 4 (centronix-type device)	/dev/cop4

For an offline device...	Key in...
Key in any valid directory path name such as...	/usr/plotwrk/offline

- The device server process is **/usr/lib/nqs/devserver**. The **-n** option following the server specifies a comment that will print on a banner page before your job begins (if a banner is enabled in your devicecap file.)
- The devicecap file that you specify will pass information about the device to the devserver process. Every device must have its own devicecap file. (See the "Using Devicecap Files" section.)

  A device is disabled and inactive when it is first created. (See the "Enabling and Disabling Devices" section.)

Using Devicecap Files

Devicecap files contain the capabilities, including the default behavior, environment, and permitted input types, for the device that it serves. One devicecap file must be defined for each device.

NOTE When you create device queues using **pconfig**, an appropriate devicecap file is generated for you.

Devicecap File Format

A devicecap file contains a list of device characteristics, separated by colons, that define how jobs submitted to that device will print. An example of a devicecap file follows.

NOTE Information can be continued to the next line by placing a "\" at the end of the first line. Lines beginning with "#" are comments that NQS ignores.

```
# devicecap for lp811 (lp811.cap)
# define stty settings and environment characteristics
CONFIGURATION:af=/usr/tmp/sysprint.acc:\
      :io=9600 -raw:stty settings:\
      :en=MYDRIVERS=/usr/drivers,NAME=lp811:environ vars:
# set data types, filters, ports, and page parameters
text|default|ascii:of=/usr/bin/textplot -p p0:\
      if=pr -l66:pl#66:pw#132:
igds:of=/usrbin/igdsplot -p p0:
script:of=/usr/bin/ps -p p0:
```

The values that can be included in the devicecap file are discussed in the following table.

Value	**Purpose**
CONFIGURATION	Defines the default values for the entire file.
default	Defines the input type if none is specified in the **qpr**(1) command line.
text, ascii, igds, script	Specifies valid input types. (See your *InterPlot User's Guide* for a complete list of input types.)
af=*string*	Defines the name of the file that accounting information will be written to.
ba (boolean)	Generates an ASCII banner for the specified input type.
io=*string*	Defines the **stty** settings.
en=*string*	Defines the default environment variables.
of=*string*	Defines the output filter. Output filters are invoked only once in a job's lifetime.
if=*string*	Defines the input filter. Input filters are invoked once for each file in a job's lifetime.
ff (boolean)	Understands formfeeds.
lf=*string*	Defines the name of the file that errors will be written to.
pl#number	Defines the page length.
pw#number	Defines the page width.

Changing the Device List for a Device Queue

You may (if you have NQS manager privileges) add, delete, or set (redefine) devices for an existing device queue. (Device queues are discussed in the "Configuring Queues from the Command Line" section.)

Adding a Device to a Device List

- To add a device to the end of a device list for a queue, key in the following:

  Mgr: **add device=***device queue_name*

- To delete a device from a device list for a queue, key in the following:

  Mgr: **delete device** *device queue_name*

- To redefine an existing device list (delete it and create a new one), key in the following:

  Mgr: **set device** (*device, device,...*) *queue_name*

Example Device Configuration

The following is an example of a simple device being created:

```
Mgr: create device epson fullname=/dev/tty00 form=sysform\
Mgr:_ server=(/usr/lib/nqs/devserver -n EPSON /usr/lib/nqs/config_files\
Mgr:_ /epson.cap)
```

The command lines above specify a **printronix** device that exists on serial port 0 as **/dev/tty00**. The print request will be printed on the form **sysform**. The server that will service the device is **devserver**, and the devicecap file that will pass device information to **devserver** is called **epson.cap**. The word EPSON will print on the banner page.

Device Defaults

The default device configuration has the following characteristics:

- The printer is an ASCII device that will accept output at 9600 baud.
- The NQS banner has 66-lines-per-page and 132 columns.
- The printer does not understand form feeds for banner pages. NQS will use carriage returns to advance to the top of the next page.

Enabling and Disabling Devices

Like queues, devices can be enabled or disabled through **qmgr**(1M). Disabling a device prevents a broken device from being used.

- An enabled device is ready to accept requests.
- A disabled device will not accept requests.

A device is disabled when it is first created. When a device is enabled, it will handle one request at a time from its device queue(s).

The enable device Command

Enable a device by keying in the following:

Mgr: **enable device** *device*

The disable device Command

Disable a device by keying in the following:

Mgr: **disable device** *device*

Setting Defaults

NOTE The commands in this chapter must be executed from the Mgr: prompt. The Mgr: prompt indicates that you are in the qmgr(1M) environment. To access the Mgr: prompt, key in qmgr at the system prompt as follows:

```
$ qmgr
Mgr:
```

You must have NQS manager privileges to set system defaults.

Setting Default Request Parameters

If you have NQS manager privileges, you can set several default request parameters in **qmgr**(1M). These parameters will be used in the absence of specific parameters on an NQS command line. Setting defaults permits you to submit requests without needing to key in each argument. For example, if a default queue parameter is specified to NQS, you will not need to key in the queue name when you want to submit a print or batch job to that queue. If no default queue is specified to NQS, the job will fail in the absence of a queue name. The following sections discuss the defaults that you can set.

Setting a Default Batch Request Priority

The **set default batch_request priority** command sets the intraqueue priority (the relative ordering of requests in a queue) for batch requests. If you do not specify a priority using the **-p** parameter on the **qsub**(1) command line, NQS uses the default priority.

NOTE The default batch request priority does not determine the request's execution priority; it determines only the relative ordering of requests in a queue.

Key in the command as follows, where *n* is the default priority value:

Mgr: **set default batch_request priority=***n*

Setting the Default Batch Request Queue

There are two ways to set a default batch request queue. These are the following:

- Place the default batch request queue in your system environment.
- Use the **qmgr(1M) set default batch_request queue** command.

Placing a Default Batch Request Queue in Your Environment

If you want to set a default batch request queue in your system environment, place the following line in your **.env** or **.profile** file, where *default_queue* is the name of your default batch queue:

```
QSUB_QUEUE=default_queue
export QPR_QUEUE
```

Using the qmgr(1M) set default batch_request queue Command

The **set default batch_request queue** command sets the default queue for batch requests. If you do not specify a queue using the **-q** parameter on the **qsub(1)** command line and you have not set up an environment variable to define your default batch queue, NQS uses the default queue set with this command.

Key in the command as follows, where *queue_name* is the name of the default batch queue.

```
Mgr: set default batch_request queue=queue_name
```

Removing a Default Batch Queue

The **set no_default batch_request queue** removes a default batch queue. Key in the command as follows:

```
Mgr: set no_default batch_request queue
```

Setting the Default Print Request Queue

There are two ways to set a default print request queue. These are the following:

- Place the default print request queue in your system environment.
- Use the **qmgr(1M) set default print_request queue** command.

Placing a Default Print Request Queue in Your Environment

If you want to set a default print request queue in your system environment, place the following lines in your **.env** or **.profile** file, where *default_queue* is the name of your default print queue:

```
QPR_QUEUE=default_queue
export QPR_QUEUE
```

Using the qmgr set default print_request queue Command

The **set default print_request queue** command sets the default queue for print requests. If you do not specify a queue using the **-q** parameter on the **qpr(1)** command line, and you have not set up an environment variable to define your default print queue, NQS uses the default queue set with this command.

Key in the command as follows, where *queue_name* is the name of the default print queue.

```
Mgr: set default print_request queue=queue_name
```

Removing a Default Print Queue

The **set no_default print_request queue** removes a default print queue. Key in the command as follows:

```
Mgr: set no_default print_request queue
```

Setting the Debug Level

The **set debug** command sets the level at which NQS records debug informtion. Currently, debugging may be either on or off; the system defaults to off.

Key in the **set debug** command as follows, where level is either 0 (off) or 1 (on):

```
Mgr: set debug level
```

Setting a Default Destination Retry Time

The **set default destination_retry time** command sets the default maximum number of hours that may elapse while NQS attempts to reach a pipe queue's destination. After this time, the request will fail.

Key in the command as follows, where *retry_time* is the number of hours that NQS will attempt to reach the destination queue:

Mgr: **set default destination_retry time** *retry_time*

Setting a Default Destination Retry Wait Time

The **set default destination_retry wait** command sets the default number of minutes that NQS will wait before retrying to reach a pipe queue's destination queue that was unreachable during the last attempt. NQS will retry to reach the destination queue until the maximum retry time has elapsed.

Key in the command as follows, where *interval* is the number of minutes that NQS will wait before retrying to reach a destination queue:

Mgr: **set default destination_retry wait** *interval*

Setting a Default Device Request Priority

The **set default device_request priority** command sets the intraqueue priority (the relative ordering of requests in a queue) for device requests. If you do not specify a priority using the **-p** parameter on the **qpr**(1) command line, NQS uses the default priority.

NOTE

The default device request priority does not determine the request's execution priority; it determines only the relative ordering of requests in a queue.

Key in the command as follows, where *n* is the default priority value:

Mgr: **set default device_request priority=***n*

Setting a Default Print Form

The **set default print_request forms** command sets the default form to be used for print requests. If no form is specified on the **qpr**(1) command line using the **-f** parameter, NQS will assume that the request is printing on the default form. If no default is specified, NQS assumes that any form is acceptable.

Key in the **set default print_request forms** command, where *form_name* is the name of the default form.

Mgr: **set default print forms** *form_name*

Removing a Default Form

The **set no_default print forms** command removes the default print form. Key in the command as follows:

Mgr: **set no_default print forms**

Setting a Default Shell Strategy

The **set shell_strategy** sets the default shell strategy for the execution of batch requests. These are the shell strategy options:

- fixed
- free
- login

Fixed Shell Strategy

A fixed shell strategy defines the specified shell path name as the shell to be used to execute all batch requests. Use a fixed shell strategy for NQS installations that use only one shell type.

Set the shell strategy to fixed by keying in the following, where *shell_pathname* is the full path name of the specified shell (Korn, C-shell, or Bourne):

Mgr: **set shell_strategy fixed=(***shell_pathname***)**

Free Shell Strategy

A free shell strategy instructs NQS to spawn the user's login shell when that user's batch request is run. The user's login shell will examine the user's batch request shell script to determine the shell that will execute it. Therefore, the free shell strategy results in the spawning of an extra shell for all batch request executions, just as if you had executed the script interactively.

Set the shell strategy to free by keying in the following:

Mgr: **set shell_strategy free**

Login Shell Strategy

A login shell strategy instructs NQS to spawn the originating user's login shell to execute all batch requests. The login shell will not examine the batch shell script to determine its script type.

Set the shell strategy to login by keying in the following:

Mgr: **set shell_strategy login**

Setting Default Maximum Values

If you have NQS manager privileges, you can specify several default maximum values for queues in **qmgr**(1M). These values are the limits that prevent requests from waiting indefinitely in the queue or devices from becoming overloaded. The commands used to set default maximum values are discussed below.

The set lifetime Command

The **set lifetime** command sets the maximum lifetime for a request waiting in the named pipe queue. After this time expires, the request will fail and NQS will send a mail message informing the request owner that the request has been deleted.

The **set lifetime** command prevents queues from filling up due to system or network failures. The lifetime value defaults to 72 hours for queues created in **qmgr**(1M). If you want to set an indefinite lifetime, set the lifetime value to zero.

Key in the command as follows, where *hours* is the maximum lifetime in hours for a request:

Mgr: **set lifetime** *hours*

The set maximum copies Command

The **set maximum copies** command sets the maximum number of copies that can be printed for a request submitted to the named device queue. NQS defaults to two copies.

Key in the command as follows, where *n* is the maximum number of copies:

Mgr: **set maximum copies** *n queue_name*

The set maximum open_retries Command

The **set maximum open_retries** command sets the maximum number of times that NQS will attempt to open a device for writing.

Key in the command as follows, where *n* is the maximum number of retries.

Mgr: **set maximum open_retries** *n queue_name*

The set maximum print_size Command

The **set maximum print_size** command sets the maximum size in bytes for a print request submitted to the named device queue.

Key in the command as follows, where *size* is the print file's size in bytes.

Mgr: **set maximum print_size** *size queue_name*

The set nice_value_limit Command

The **set nice_value_limit** command is used to set the per-process CLIX nice value limit for any batch request placed in the named batch queue without a user-specified nice value.

Each batch request is examined when it is queued to ensure that any user-specified nice value does not exceed to nice value for the batch queue. The user-specified nice value cannot be numerically less than the queue's nice value limit. If you attempt to queue a batch job that requests a nice value limit that is numerically too low, the job is rejected. (Your NQS online help and the CLIX documentation contains details about nice values.)

NOTE **Some implementations of UNIX do not support configurable limits.**

Key in the **set nice_value_limit** command as follows:

Mgr: **set nice_value_limit=***nice_value batch_queue*

The set per_process permfile_limit Command

The **set per_process permfile_limit** command sets the per-process maximum permanent file size for all batch requests placed in the specified batch queues. When each batch request is placed in a queue, it is examined to ensure that the maximum permanent file size does not exceed the maximum file size for the queue (provided that a maximum file size has been set for the queue). If you attempt to queue a batch job that requests a larger file size than the queue allows, the request is rejected.

Key in the **set per_process permfile_limit** command as follows:

Mgr: **set per_process permfile_limit** *limit batch_queue*

The set open_wait Command

The **set open_wait** command sets the number of seconds that NQS will wait before retrying to send a request from a device queue to a failed device.

Key in the command as follows, where *seconds* is the interval in seconds between device retries:

Mgr: **set open_wait** *seconds*

Network Services

NQS contains three network service utilities. These are the following:

- The **netdaemon** (network daemon) listens for messages from remote clients and manages network requests. The full path name for the default **netdaemon** process is **/usr/lib/nqs/netdaemon**.
- The **netclient** (network client) stages out batch request output files and empties network queues. The full path name for the default **netclient** process is **/usr/lib/nqs/netclient**.
- The **netserver** (network server) processes submission requests from remote pipe clients. The full path name for the default **netserver** is **/usr/lib/nqs/netserver**.

NOTE Do not attempt to modify the network service utilities.

Locking the Local Daemon

Locking the network daemon in memory permits faster access to the daemon. However, locking the daemon requires additional system memory and should not be done if the daemon is used infrequently.

To lock the daemon in memory, key in the following:

Mgr: **lock local_daemon**

Unlocking the Local Daemon

To unlock a daemon that you have locked in memory, key in the following:

Mgr: **unlock local_daemon**

An Internet address is often referred to as a TCP/IP address. The terms Internet address and TCP/IP address are used interchangeably. If the TCP/IP software is running on the NQS host, the host will have an Internet address. Note that the TCP/IP software does not need to run on the host to run NQS. For more information, see the "BSD Network Configuration Tutorial."

Classes of Addresses

The Internet address identifies the network and individual hosts within the network.

Internet addresses are divided into three ranges, each representing a network with a different size than the others.

- *Class A* addresses are reserved for large networks. Class A contains only 126 networks composed mainly of government and large commercial organizations.
- *Class B* addresses are generally assigned to all other organizations with a large network. Class B addresses allow 64,516 nodes.
- *Class C* addresses allow only 254 hosts on each network; however, Class C can have several networks.

Valid Formats

An Internet address has the following form:

Class A

Internet Address:	*nnn.hhh.hhh.hhh*
Restrictions:	*nnn* < 126

Class B

Internet Address:	*nnn.nnn.hhh.hhh*
Restrictions:	128.001 ⩽ *nnn.nnn* < 191.254

Class C

Internet Address:	*nnn.nnn.nnn.hhh*
Restrictions:	192.001.001 ⩽ *nnn.nnn.nnn* < 233.254.254

N is the network number and *h* is the host number. The numbers 0 and 255 should not be used in any field.

Assigning an Internet Address

The system administrator must assign an Internet address to all NQS nodes when the NQS software is delivered.

- If the NQS host is not connected to an official Department of Defense (DoD) network, any valid Internet address can be assigned to the NQS nodes.
- If the NQS host will be connected to a DoD network, you must get a registered Internet address from the DoD.

Account Mapping

NQS account mapping can be tedious. Because the queuing is networked, a more complex mapping is needed to secure NQS from unauthorized clients. And, knowing the local user responsible for the request is necessary in case a disaster strikes, the job fails, or the original requester's remote system is unreachable.

Two main methods are used to define NQS access. The first method is the default system access defined in **/etc/hosts.equiv** or **.rhosts**. Using **/etc/hosts.equiv** is dangerous because it maps any user from a specified remote system so long as the user name matches.

The **.rhosts** file is better because the user controls the access. However, the user name must still match on both local and remote systems for NQS to recognize the mapping. This method is more secure than the above method and easier to use than using the **nmapmgr** method discussed next.

The last type of mapping is performed through **nmapmgr**. Although this method is more cumbersome to use, it enables you to control NQS access on the local system. The sections below discuss each method in more detail.

When submitting a request to a remote host, both local and remote hosts must allow access to each other. For example, if **red** can access machine **yellow**, but **yellow** cannot access **red**, a strange problem will occur.

If **red** submits a request to **yellow**, **yellow** will accept and service the request. When **yellow** tries to return the output to **red**, it fails because **red** does not allow access to **yellow**. In the opposite case when **yellow** submits a job to **red**, the job will immediately fail because **red** will not allow access.

/etc/hosts.equiv and .rhosts

The **/etc/hosts.equiv** file grants account access by machine and user name. Edit this file to include all names of machines allowed to have direct remote-user-name to local-user-name (**bill** on machine **red** to **bill** on machine **blue**) access to your machine. Each line should have one node name.

When a remote client succeeds in mapping a machine-id from the **nmapmgr**, it checks the **/etc/hosts.equiv** file to see if machine-wide access is allowed from a machine **x**. So, if **bill** at **red** tries to use **blue**, **/etc/hosts.equiv** at **blue** is checked to see if **red** is listed. If **red** is listed, it will allow **bill** to use **bill** at **blue** if **bill** at **blue** exists. If **bill** at **blue** does not exist, access is denied.

However, if **bill** at **blue** exists, but **red** is not listed in **/etc/hosts.equiv**, **bill** at **red** can still access **bill** at **blue** if **bill** at **blue** has a file called **.rhosts** in its home directory on **blue** that contains the line "red bill." This means that **bill** at **blue** explicitly gives access when **bill** at **red** tries to access the account.

Consequently, **bill** cannot implicitly use **tom** at the remote machine because NQS will not know to check tom's **.rhosts** file. This type of mapping must now be achieved with **nmapmgr**.

Nmapmgr: Machine Mappings

Nmapmgr controls NQS access for remote users wanting to use local resources. It also performs remote machine validation and account-to-account mappings. The super-user must run **nmapmgr** to add, delete, or change machine mappings. Otherwise, any user may query the database.

While in **nmapmgr**, you can access **help** to list valid commands. Here are some of the common commands.

add mid *mid principal-name*
change name *mid new-name*
create
delete mid *mid*
exit
get mid *name*
get name *mid*
help
quit

If installation was successful, the internal **nmap** file should have been created and the local machine-id (mid) initialized. The screen below shows a **nmapmgr** session where the **nmap** file has not been created.

```
# nmapmgr
NMAPMGR> : create  (usually done during installation)
NMAPMGR> : add mid 1 red
NMAPMGR> : add mid 2 blue
NMAPMGR> : get mid blue
NMAP_SUCCESS:  Successful completion
Mid = 2.
NMAPMGR: get name 1
NMAP_SUCCESS:  Successful completion
Name = red.
NMAPMGR: change name 2 bluejay
nMAP_SUCCESS:  Successful completion
NMAPMGR: get name 2
NMAP_SUCCESS:  Successful completion
Name = bluejay.
NMAPMGR: delete mid 1
NMAP_SUCCESS:  Successful completion
NMAPMGR: get name red
NMAP_ENOMAP:  No such mapping.
NMAPMGR: exit
#
```

This example shows that the **nmap** file is created and **red** and **blue** are added to the list of machine-ids as 1 and 2, respectively. When the machine-id **blue** is requested, **nmapmgr** returns 2, just as it returns **red** as the name of the machine-id 1.

Then, machine 2's principal name **blue** changes to **bluejay**. When you query for machine 2's name, the program verifies the name **bluejay**.

Next, machine 1 is removed from the list. The program indicates that machine 1 no longer exists when it is queried for. Finally, **nmapmgr** is exited.

The commands, **add name** *name to-machine-id* and **delete name** *name*, add and delete alias names for machines. Additional aliases can be added to machine-ids. However, if the primary machine name like **red** is deleted, the machine-id will still map to **red**, but **red** will no longer map to the machine-id. We suggest that you avoid these commands.

All machines that you wish to access must be listed in **nmapmgr**. Machine-ids must map correctly with the remote machine before access is permitted.

Nmapmgr: User Mappings

User mappings can be achieved with the **/etc/hosts** file or **.rhosts** file as discussed previously. However, **nmapmgr** is the designated mechanism that NQS uses to control user access to the local NQS.

More complex mapping can be performed in **nmapmgr** than is available in the other methods described before. However, this mapping is now done by numeric UID and can be awkward to use. Uids are mapped with the following commands:

> **add uid** *from-mid from-uid to-uid*
> **delete uid** *from-mid from-uid to-uid*

For example, uid=100 calling from mid=1 can be given explicit access to uid=127 on the local system even if the user names differ. The mapping is specified as follows:

> **add uid 1 100 127**

NOTE

Groups can also be specified using a similar command add gid. However, the command does not have meaning in this implementation.

When many users want to use a particular system, account mapping can become time consuming. In this vein, **nmapmgr** has provided default mappings. The associated commands are

> **set defuid** *from-mid defuid*
> **delete defuid** *from-mid defuid*

Suppose that anybody calling from mid=10 should use uid=500. This is specified using the following command:

```
add defuid 10 500
```

Again, default group-ids (gids) are mapped similarly, but do not have meaning.

Displaying the mappings is primitive at this point. Only the following command is provided:

get uid *from-mid from-gid*

The following command will specify where uid=107 calling from mid=7 is mapped to:

```
get uid 7 107
```

If a specific mapping is listed, it will be shown. Otherwise, a default mapping is used if possible. If not, the mapping will fail. It will also fail if the machine-id is not defined in **nmapmgr**.

An Internal Overview

Several processes are important to NQS. They are as follows:

logdaemon
nqsdaemon
netdaemon

These three processes are the core of NQS. The **nqsdaemon** is the master process of NQS. It maintains all configurations, handles all queues, schedules all requests, and owns all child processes through interprocess messaging using named pipes.

The **netdaemon** handles networking for NQS. It receives and processes all network requests. It communicates directly with the main local **nqsdaemon.**

The third daemon is the **logdaemon.** It records all error messages and activity for NQS. The amount of information in records is controlled by the **set debug** command.

As requests are processed, server processes are created to service the requests. NQS server processes include the following:

nqsdaemon (a child process of the main local nqsdaemon)
netdaemon (a child process of the main netdaemon)
netserver
netclient
pipeclient
lpserver

Sometimes the above servers will appear as *machine* **server** or *user* **server**, where *machine* and *user* name a particular machine or user.

5. YP Tutorial

Chapter 5: YP Tutorial

Introduction and Terminology

The Yellow Pages (YP) is a network service providing read access to a replicated database. The lookup service is provided by a set of YP database servers. The client interface to this service uses the Remote Procedure Call (RPC) mechanism.

Translating or mapping a name to its value is one of the most common operations performed in computer systems. Common examples are the translation of a variable name to a virtual memory address, the translation of a user name to a system ID or list of capabilities, and the translation of a network node name to an Internet address. There are two fundamental read-only operations that can be performed on a map: match and enumerate. Match means to look up a name (which we call a **key**) and return its current value. Enumerate means to return each key-value pair, in turn.

The YP supplies match and enumerate operations in a network environment, where high availability and reliability are required. It provides that availability and reliability by replicating both databases and database servers on multiple nodes within a single local network, and within the Internet. The database is replicated, but not distributed: all changes are made at a single server and eventually propagate to the remaining servers without locking. The YP is appropriate for an environment in which changes to the mapping databases occur on the order of tens per day.

The YP operates on an arbitrary number of map databases. Map names provide the lower of two levels of a naming hierarchy. Maps are grouped into named sets called *domains*. Domain names provide a second, higher level of naming. Map names must be unique within a domain, but may be duplicated in different domains. The YP client interface requires that both a map name and a domain name be supplied to perform match and enumeration operations.

The YP achieves high availability by replication. One area not addressed by the protocol which has to be addressed by the implementors is global consistency among the replicated copies of the database. Every implementation should be designed so that, at steady state, a request yields the same result when it is made of any YP database server. Update and update-propagation mechanisms must be implemented to supply the required degree of consistency.

RPC — Remote Procedure Call

The Remote Procedure Call (RPC) mechanism defines a paradigm for interprocess communication modeled on function calls. *Clients* call functions that optionally return values. All input and output to the functions are in the client's address space. The function is executed by a *server* program.

Using RPC, clients address servers by a program number, that identifies the application-level protocol that the server speaks, and a version number. Additionally, each server procedure has a procedure number assigned to it.

In an Internet environment, clients must also know the server's host Internet address and the server's rendezvous port. The server listens for service requests at ports associated with a particular transport protocol — Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol/Internet Protocol (UDP/IP).

The format of the data structures used as input to and output from the remotelyexecuted procedures are typically defined by header files that are included when the client interface functions are compiled. Levels above the client interface package need not know any particulars of the RPC interface to the server.

XDR — External Data Representation

The External Data Representation (XDR) specification establishes standard representations for basic data types (such as strings, signed and unsigned integers, and structures and unions) in a way that allows them to be transferred among machines with varying architectures. XDR provides primitives to encode (translate from the local host's representation to the standard representation) and decode (translate from the standard representation to the local host's representation) basic data types. Constructor primitives allow arbitrarily complex data types to be made from the basic types.

The YP's RPC input and output data structures are described using XDR's data description language. In general, the data description language looks like the C language, with a few extra constructs. One such extra construct is the *discriminated union*. This is like a C language union in that it can hold various objects, but differs from it in that a discriminant indicates which object it currently holds. The discriminant is first across the wire. Consider a simple example:

```
union switch (long int) {
        1:
                string expl_name16
        0:
                unsigned int expl_error_code
        default:
                struct {}
}
```

The example should be interpreted as follows: the first object to be encoded/decoded (that is, the discriminant) is a long integer. If it has the value one, the next object is a string. If the discriminant has the value zero, the next object is an unsigned integer. If the discriminant takes any other value, do not encode or decode any more data.

A *string* data type in the XDR data definition language adds the ability to specify the maximum number of elements in a byte array or string of potentially variable size. For instance:

```
        string domainYPMAXDOMAIN;
```

states that the byte sequence *domain* may be less than or equal to *YPMAX-DOMAIN* bytes long.

An additional primitive data type is a boolean, which takes the value one to mean TRUE and zero to mean FALSE.

YP Database Servers

Maps and Map Operations

Map Structure

Maps are named sets of key-value pairs. Keys and their values are counted binary objects, and may be ASCII information, but need not be. The data composing a map is determined by the client applications that are the final customers for the data, not by the YP. The YP has no syntactic or semantic knowledge of the map contents. Neither does the YP determine or know any map's name. Map names are managed by the YP's clients. Conflict in the map name space must be resolved by human administrators outside the YP system.

Typical implementations for YP maps are files or database management systems. The design of the YP's map database is an implementation detail and is unspecified by the protocol.

Match Operation

The YP supports an exact match operation in the YPPROC_MATCH procedure. That is, if a match string and some key in the map are exactly the same, the value of the key is returned. No pattern matching, case conversion, or wildcarding is supported.

Map Entry Enumeration

It is possible to get the first key-value pair in a map with YPPROC_FIRST and the next key-value pair with YPPROC_NEXT. Calling "get first" once and "get next" until the return value indicates there are no more entries in the map will retrieve each entry once. Making the same calls on the same map at the same YP database server will enumerate all entries in the same order. The actual order, however, is unspecified. Enumerating a map at a different YP database server will not necessarily return entries in the same order.

Entire Map Retrieval

The YPPROC_ALL operation retrieves all key-value pairs in a map with a single RPC request. This is faster than map entry enumeration, and more reliable since it uses TCP. Ordering is the same as when enumeration is applied.

Map Update

The update of YP maps is an implementation detail which is outside the specification of the YP service.

Master and Slave YP Database Servers

For each map, there is one YP database server called the map's *master*. Map updates take place only on the master. An updated map should be transferred from the master to the rest of the YP database servers, which are *slave* servers for this map.

It is possible for each map to have a different YP database server as its master, for all maps to have the same master, or any other combination. The choice of how to set up map masters is one of implementation and administrative policy.

Map Propagation and Consistency

Getting map updates from the master to the slaves is called map propagation. Neither technology nor algorithms for map propagation are specified by the protocol. Map propagation may be entirely manual: for instance, a person could copy the maps from the master to the slaves at a regular interval or when a change is made on the master. This is unnecessarily labor intensive.

In order to escape the idiosyncrasies of any particular implementation, all maps should be uniformly timestamped.

Functions to Aid in Map Propagation

The way a map is transferred from one server to another is not specified by the YP protocol. One possibility would be for the system administrator to do it manually. Another would be for the YP database server to activate another process to perform the map transfer. A third would be for a server to enumerate a recent version of the map, using the normal client map enumeration functions.

The YPPROC_XFR procedure requests the YP server to update a map and permits the actual transfer agent (some server process) to call back the requester with a summary status.

Domains

Domains provide a second level for naming within the YP subsystem. They are names for sets of maps. Therefore, create separate map name spaces. Domains provide an opportunity to break large organizations into administerable chunks and the ability to create parallel, noninterfering test and production environments.

Ideally, the domain of interest to a client ought to be associated with the invoking user, but in practice it is useful for client machines to be in a default domain. Implementations of the YP client interface should supply some mechanism for telling processes the domain name they should use. This is needed not only because the concept of domain is a useless one as far as most programs are concerned, but, more importantly, so that programs that are insensitive to both location and the invoking user can be written.

Nonfeatures

The following capabilities are not included in the current YP protocols:

All write (and delete) access to the YP's map database is assumed to be outside of the YP subsystem. It is probable that write access to the map database will be included in later versions of the YP protocols.

Version Commitment Across Multiple Requests

The YP protocol was designed to keep the YP database server stateless with regard to its clients. Therefore, there is no facility for contracting with a server to preallocate any resource beyond that required to service any single request. In particular, there is no way to get a server to commit to use a single version of a map while trying to enumerate that map's entries. Use YPPROC_ALL to avoid these problems.

Guaranteed Global Consistency

There is no facility for locking maps during the update or propagation phases. Therefore, it is virtually guaranteed that the map database be globally inconsistent during those phases. The set of client applications for which the YP is an appropriate lookup service is one that (by definition) must be tolerant of transient inconsistencies.

Access Control

The YP database servers make no attempt to restrict access to the map data by any means. All syntactically correct requests are serviced.

YP Database Server Protocol Definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions as the service matures.

RPC Constants

All numbers are in decimal.
The YP database server protocol program number.

YPVERS 2	The current YP protocol version.

Other Manifest Constants

All numbers are in decimal.
The total maximum size of key and value for any pair. The absolute sizes of the key and value may divide this maximum arbitrarily.

YPMAXDOMAIN 64	The maximum number of characters in a domain name.
YPMAXMAP 64	The maximum number of characters in a map name.
YPMAXPEER 64	The maximum number of characters in a YP host name.

Remote Procedure Return Values

This section presents the status values returned by several of the YP remote procedures. All numbers are in decimal.

ypstat

```
typedef enum {
        YP_TRUE    =  1,  /* General purpose success code. */
        YP_NOMORE  =  2,  /* No more entries in map. */
        YP_FALSE   =  0,  /* General purpose failure code.*/
        YP_NOMAP   = -1,  /* No such map in domain. */
        YP_NODOM   = -2,  /* Domain not supported. */
        YP_NOKEY   = -3,  /* No such key in map. */
        YP_BADOP   = -4,  /* Invalid operation. */
        YP_BADDB   = -5,  /* Server database is bad. */
        YP_YPERR   = -6,  /* YP server error. */
        YP_BADARGS = -7,  /* Request arguments bad. */
        YP_VERS    = -8    /* YP server version mismatch. */
} ypstat
```

ypxfrstat

```
typedef enum {
        YPXFR_SUCC    =   1,  /* Success */
        YPXFR_AGE     =   2,  /* Master's version not newer */
        YPXFR_NOMAP   =  -1,  /* Can't find server for map */
        YPXFR_NODOM   =  -2,  /* Domain not supported */
        YPXFR_RSRC    =  -3,  /* Local resource alloc failure */
        YPXFR_RPC     =  -4,  /* RPC failure talking to server */
        YPXFR_MADDR   =  -5,  /* Can't get master address */
        YPXFR_YPERR   =  -6,  /* YP server/map db error */
        YPXFR_BADARGS=   -7,  /* Request arguments bad */
        YPXFR_DBM     =  -8,  /* Local database failure */
        YPXFR_FILE    =  -9,  /* Local file I/O failure */
        YPXFR_SKEW    = -10,  /* Map version skew in transfer */
        YPXFR_CLEAR   = -11,  /* Can't clear local ypserv */
        YPXFR_FORCE   = -12,  /* Must override defaults */
        YPXFR_XFRERR  = -13,  /* ypxfr error */
        YPXFR_REFUSED= -14    /* ypserv refused transfer */
} ypxfrstat
```

Basic Data Structures

This section defines the data structures used as input to and output from the YP remote procedures.

domainname

```
typedef string domainname<YPMAXDOMAIN>
```

mapname

```
typedef string mapname<YPMAXMAP>
```

peername

```
        typedef string peernameYPMAXPEER
```

keydat

```
        typedef string keydatYPMAXRECORD
```

valdat

```
        typedef string valdatYPMAXRECORD
```

ypmap_parms

```
        typedef struct {
                domainname
                mapname
                unsigned long ordernum
                peername
        } ypmap_parms
```

This contains parameters giving information about map *mapname* within domain *domainname*; *peername* is the name of the map's master YP database server. If any of the three strings is null, it indicates information is unknown or unavailable. The *ordernum* element contains a binary value representing the value of the map's order number; if unavailable, this is 0.

ypreq_xfr

```
        typedef struct {
                struct ypmap_parms map_parms
                unsigned long transid
                unsigned long prog
                unsigned short port
        } ypreq_xfr
```

ypresp_val

```
typedef struct {
        ypstat
        valdat
} ypresp_val
```

ypresp_key_val

```
typedef struct {
        ypstat
        keydat
        valdat
} ypresp_key_val
```

ypresp_master

```
typedef struct {
        ypstat
        peername
} ypresp_master
```

ypresp_order

```
typedef struct {
        ypstat
        unsigned long ordernum
} ypresp_order
```

ypresp_all

```
typedef union switch (boolean more) {
        TRUE:
                ypresp_key_val
        FALSE:
                struct { }
} ypresp_all
```

ypresp_xfr

```
        typedef struct {
                unsigned long transid
                ypxfrstat xfrstat
        } ypresp_xfr
```

ypmaplist

```
        typedef struct {
                mapname
                ypmaplist *
        } ypmaplist
```

ypresp_maplist

```
        typedef struct {
                ypstat
                ypmaplist *
        } ypresp_maplist
```

YP Database Server Remote Procedures

This section contains a specification for each function that can be called as a remote procedure. The input and output parameters are described using the XDR data definition language.

Do Nothing

Procedure 0, Version 2.

```
        0. YPPROC_NULL ( ) returns ( )
```

This takes no arguments, does no work, and returns nothing. It is made available in all RPC services to allow server response testing and timing.

Do You Serve This Domain?

Procedure 1, Version 2.

```
1. YPPROC_DOMAIN (domain) returns (serves)
        domainname domain;
        boolean serves;
```

This returns TRUE if the server serves domain and FALSE otherwise. This procedure allows a potential client to determine if a given server supports a certain domain.

Answer Only If You Serve This Domain

Procedure 2, Version 2.

```
2. YPPROC_DOMAIN_NONACK (domain) returns (serves)
        domainname domain;
        boolean serves;
```

This procedure returns TRUE if the server serves *domain*; otherwise, it does not return. The intent of the function is that it be called in a broadcast environment in which it is useful to restrict the number of useless messages. If this function is called, the client interface implementation must be written to regain control in the negative case, such as by means of a timeout on the response.

The current implementation currently returns in the FALSE case by forcing an RPC decode error.

Return Value of a Key

Procedure 3, Version 2.

```
3. YPPROC_MATCH (req) returns (resp)
        ypreq_key req;
        ypresp_val resp;
```

This returns the value associated with the datum *keydat* in *req*. If the *status* element in *resp* has the value YP_TRUE, the value data is returned in the datum *valdat*.

Get First Key-Value Pair in Map

Procedure 4, Version 2.

```
4. YPPROC_FIRST (req) returns (resp)
        ypreq_key req;
        ypresp_key_val resp;
```

If *status* has the value YP_TRUE, this returns the first key-value pair from the map named in *req* to the *keydat* and *valdat* elements within *resp*. When *status* contains the value YP_NOMORE, the map is empty.

Get Next Key-Value Pair in Map

Procedure 5, Version 2.

```
5. YPPROC_NEXT (req) returns (resp)
        ypreq_key req;
        ypresp_key_val resp;
```

If *status* has the value YP_TRUE, this returns the key-value pair following the key-value named *req* to the *keydat* and *valdat* elements within *resp*. If the passed key is the last key in the map, the value of *status* is YP_NOMORE.

Transfer Map

Procedure 6, Version 2.

```
6. YPPROC_XFR (req) returns (resp)
        ypreq_xfr req;
        ypresp_xfr resp;
```

The action taken in response to this request is unspecified and is implementation-dependent. The intention is to indicate to the server that a map should be updated, and to allow the actual transfer agent (whether it be the YP server process or some other process) to call back the requester with a summary status.

The transfer agent should call back the program running on the requesting host with program number *req.prog*, program version 1, and listening at port *req.port*. The procedure number is 1 and the callback data is of type *ypresp_xfr*. The *transid* field should turn around *req.transid*, and the *xfrstat* field should be set appropriately.

Reinitialize Internal State

Procedure 7, Version 2.

```
        7. YPPROC_CLEAR ( ) returns ( )
```

The action taken in response to this request is unspecified and is implementation-dependent. Different server implementations may have different amounts of internal state (such as open files or the current map). This request signals that all such state should be expunged.

Get All Key-Value Pairs In Map

Procedure 8, Version 2.

```
        8. YPPROC_ALL (req) returns (resp)
                ypreq_nokey req;
                ypresp_all resp;
```

This allows all key-value pairs from a map to be transferred with a single RPC request. When the union's discriminant is FALSE, no more key-value pairs will be returned. The status field of the last *rpresp_key_val* structure should be consulted to determine why the flow of returned key-value pairs has stopped.

Get Map Master Name

Procedure 9, Version 2.

```
        9. YPPROC_MASTER (req) returns (resp)
                ypreq_nokey req;
                ypresp_master resp;
```

This returns the map's master YP server inside the *resp* structure.

Get Map Order Number

Procedure 10, Version 2.

```
        10. YPPROC_ORDER (req) returns (resp)
                ypreq_nokey req;
                ypresp_order resp;
```

This returns a map's order number as an unsigned long integer, which indicates when the map was built. This quantity represents the number of seconds since the midnight before 1 January 1970 GMT.

Get All Maps in Domain

Procedure 11, Version 2.

```
11. YPPROC_MAPLIST (req) returns (resp)
        domainname req;
        ypresp_maplist resp;
```

This returns a list of all the maps in a domain.

YP Binders

Introduction

So that any network service is usable, there must be some way for potential clients to find the servers. This section describes the YP binder, an optional element in the YP subsystem that supplies YP database server addressing information to potential YP clients.

In order to address a YP server in the Internet environment, a client must know the server's Internet address and the port at which the server is listening for service requests. No contract is negotiated between a YP server and a potential client. Therefore, the addressing information is sufficient to bind the client to the server.

Of the many possible ways for a client to get the addressing information, one alternative is to supply an entity to cache the bindings and to serve that binding database to potential YP clients. The theory is that, if finding the service takes a lot of work, allocate a specialist to do it rather than burden every client with a job that is irrelevant to its real function. A YP binder only makes sense if it is easier for a client to find the YP binder than to find a YP database server and if the YP binder can find a YP database server.

We assume that a YP binder is present at every network node, and because of this, addressing the YP binder is easier than addressing a YP database server. The scheme for finding a local resource is implementation-specific, but given that a resource is guaranteed to be local, there may be some efficient way of finding it. We further assume that the YP binder can find a YP database server somehow, but that the way is either complicated, time-consuming, or resource-consuming. If either of these assumptions is untrue, probably your implementation is not a good choice for a YP binder.

If a YP binder is implemented, it can provide added value beyond the binding: it can verify that the binding is correct and that the YP database server is alive and well, for instance. The degree of sureness in a binding that the YP binder gives to a client is a parameter that can be tuned appropriately in the implementation.

YP Binder Protocol Definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions as the service matures.

RPC Constants

All numbers are in decimal.

YPBINDPROG 100007	The YP binder protocol program number.
YPBINDVERS 2	The current YP binder protocol version.

Other Manifest Constants

All numbers are decimal.

YPMAXDOMAIN 64	The maximum number of characters in a domain name. This is identical to the constant defined above within the YP database server protocol section.

ypbind_resptype

```
    enum ypbind_resptype {
                                    YPBIND_SUCC_VAL = 1,
                                    YPBIND_FAIL_VAL = 2
    }
```

This discriminates between success responses and failure responses to a YPBINDPROC_DOMAIN request.

ypbinderr

```
    typedef enum {
                                    YPBIND_ERR_ERR 1/* Internal error */
                                    YPBIND_ERR_NOSERV 2/* No bound server for domain */
                                    YPBIND_ERR_RESC 3/* Can't allocate system resource */
    } ypbinderr
```

The error case of most interest to a YP binder client is YPBIND_ERR_NOSERV; this means that the binding request cannot be satisfied because the YP binder does not know how to address any YP database server in the named domain.

Basic Data Structures

This section defines the data structures used as input to and output from the YP binder remote procedures.

domainname

```
typedef string domainnameYPMAXDOMAIN
```

This is identical to the domainname string defined above within the YP database server protocol section.

ypbind_binding

```
typedef struct {
                        unsigned long ypbind_binding_addr
                        unsigned short ypbind_binding_port
} ypbind_binding
```

This contains the information necessary to bind a client to a YP database server in the Internet environment: *ypbind_binding_addr* holds the host IP address (four bytes), and *ypbind_binding_port* holds the port address (two bytes). Both IP address and port address must be in ARPA network byte order (most significant byte first or big endian), regardless of the host machine's native architecture.

ypbind_resp

```
typedef struct {
                        union switch (enum ypbind_resptype status) {
                        YPBIND_SUCC_VAL:
                        ypbind_binding
                        YPBIND_FAIL_VAL:
                        ypbinderr
                        default:
                        { }
                        }
} ypbind_resp
```

This is the response to a YPBINDPROC_DOMAIN request.

ypbind_setdom

```
typedef struct {
                                domainname
                                ypbind_binding
                                version
} ypbind_setdom
```

This is the input data structure for the YPBINDPROC_SETDOM procedure.

YP Binder Remote Procedures

Like the YP procedures earlier, these procedures are described using the XDR data definition language.

Do Nothing

Procedure 0, Version 2.

```
0. YPBINDPROC_NULL ( ) returns ( )
```

This does no work. It is made available in all RPC services to allow server response testing and timing.

Get Current Binding for a Domain

Procedure 1, Version 2.

```
1. YPBINDPROC_DOMAIN (domain) returns (resp)
                              domainname domain;
                              ypbind_resp resp;
```

This returns the binding information necessary to address a YP database server within the Internet environment.

Set Domain Binding

Procedure 2, Version 2.

```
2. YPBINDPROC_SETDOM (setdom) returns ( )
                              ypbind_setdom setdom;
```

This instructs a YP binder to use the passed information as its current binding information for the passed domain.

Part 2: System Administrator's Procedures

Procedure 1: System Rebuild

Overview of System Rebuild

Purpose	To replace damaged file systems on a hard disk
When Performed	When the hard disk is corrupted so that the workstation/server will not boot or when bad block messages are received
Bootable Programs	Rebuild Boot Floppy
Media	Rebuild Boot Floppy (#1) Rebuild Boot Floppy (#2) Rebuild Root Floppy (#3) Rebuild Root Floppy (#4) Rebuild Root Floppy (#5) Rebuild Root Floppy (#6)
Time	1 hour
Caution	Formatting the hard disk destroys all data on the hard disk. Rebuilding a partition destroys all data on the partition. Therefore, you should back up all data.

Rebuilding the hard disk involves (at the minimum) rewriting the file system(s). Existing data on a file system is overwritten; thus, rebuilding can be a destructive process. Rebuild the hard disk under any of the following circumstances. Unless one of these circumstances is true, do not rebuild the hard disk.

- The hard disk is corrupted and you cannot boot your workstation.
- You cannot boot your workstation to single-user mode. (You never reach the blue introductory screen.)
- Bad block messages such as the following appear on the screen:

```
Disk failed: s0u0p7.1 read error at block 2933
```

- You want to change file system types. (For example, you want to convert **/usr** from a standard file system to a Fast File System.)
- You want to alter the partition sizes on the hard disk.

This chapter describes how to rebuild the hard disk. Following these steps will ensure that you lose as little data as possible when you rebuild. Each step is described in detail throughout the rest of this procedure.

- Back up all personal files.
- Create Rebuild floppy disks with the current software.
- Boot from the Rebuild Boot floppy (#1).
- Enter the Utility Pages.
- Verify hard disk flaws.
- Format the hard disk (optional).
- Select the **Rebuild** Utility Page icon.
- Repartition the hard disk (optional).
- Load the Rebuild Root media.
- Enter the Rebuild environment.
- Restore the file systems.
- Load nucleus software with the **newprod**(1M) utility.
- Prepare nonstandard partitions for use.
- Load application software with the **newprod**(1M) utility.
- Restore files from backups.

Backing Up Files

Before you rebuild the hard disk you should first back up all personal files, including system files (such as **/etc/passwd** and **/etc/group**) that are unique to the machine you are rebuilding. The rebuild process recreates the file systems on the hard disk and, in the process, overwrites all data on the hard disk. After the rebuild, you will not be able to access any files in the file systems that were recreated.

Creating Current Rebuild Floppy Disks

After you have backed up your personal files, you should create or locate a current set of Rebuild floppy disks. The Rebuild floppy disk set is used to rebuild a corrupted hard disk to a bootable state.

Customers receive a set of Rebuild floppy disks with each shipment of Intergraph software. If you have Rebuild floppy disks from the most recent software shipment, you may use them to rebuild.

CLIPPER Rebuild floppy sets contain these diskettes:

- Rebuild Boot for 100/32C/300-series (floppy #1)
- Rebuild Boot for 300/400/3000/4000-series (floppy #1)
- Rebuild Boot for 6000-series (floppy #1)
- Rebuild Root (floppy #2)
- Rebuild Root (floppy #3)
- Rebuild Root (floppy #4)
- Rebuild Root (floppy #5)
- Rebuild Root (floppy #6)

To rebuild, you will use the Rebuild Boot floppy disk appropriate for the machine you are rebuilding to boot and the Rebuild Root floppies (2-6) to rewrite the file systems. You will not use the extra Rebuild Boot floppies.

If you cannot locate the most recent Rebuild floppy disks, you will need to create a current set of Rebuild floppy disks. This section provides procedures for creating Rebuild floppy disks. You need six formatted, high-density floppy disks to create a Rebuild floppy disk set.

NOTE **Do not attempt to make Rebuild floppy disks on the machine that you are rebuilding. Instead, use another machine that is completely operational.**

Follow these steps to create a CLIPPER Rebuild floppy disk set:

1. You must be in super-user mode to perform this procedure. Log in and key in **su** at the system prompt as follows:

   login: *username*

```
$ su
#
```

2. Invoke the **newprod**(1M) utility to load the rebuild product (sss0003) by keying in the following:

   newprod sss0003

3. Key in the correct entry at the following prompt:

   ```
   Enter source of installation n)etwork, f)loppy, t)ape,
   r)emote cdrom, or local c)drom ;
   ```

   If you key in **n** for network or **r** for remote CDROM, the following prompt appears:

   ```
   Enter Ethernet connect string (08-00-36-XX-XX-XX) ;
   ```

   Key in the Ethernet address or the node name of the delivery node where the new products reside and a valid user name and password.

4. Messages similar to the following appear. Follow the instructions as prompted on the screen. To create a complete CLIPPER Rebuild floppy set, select "both" at the first prompt.

```
Installing: REBUILD (sss0003) - Rebuild Floppies
Installing.....

Do you want to make a "root" set, a "boot" floppy, both or
none? [both]:

Do you want to make a "boot" floppy for a 100/32C/200 series
system, a 300/400/3000/4000 system or a 6000 series system?
[100/32C/200]:

Downloading the Clipper Rebuild #1 (100/32C/200) floppy.
Insert a formatted floppy into the floppy drive and press
<RETURN> -->

You have a good "100/32C/200" floppy!
Label this floppy "Clipper Rebuild #1 (100/32C/200)"

Downloading the Clipper Rebuild #2 (root) floppy.
Insert a formatted floppy into the floppy drive and press
<RETURN> -->
```

continued

```
You have a good "root" floppy!
Label this floppy "Clipper Rebuild #2 (root) (100/32C/200)"

Downloading the Clipper Rebuild #3 (root) floppy.
Insert a formatted floppy into the floppy drive and press
<RETURN> -->

You have a good "root" floppy!
Label this floppy "Clipper Rebuild #3 (root) (100/32C/200)"

Downloading the Clipper Rebuild #4 (root) floppy.
Insert a formatted floppy into the floppy drive and press
<RETURN> -->

You have a good "root" floppy!
Label this floppy "Clipper Rebuild #4 (root) (100/32C/200)"

Downloading the Clipper Rebuild #5 (root) floppy.
Insert a formatted floppy into the floppy drive and press
<RETURN> -->

You have a good "root" floppy!
Label this floppy "Clipper Rebuild #5 (root) (100/32C/200)"

Downloading the Clipper Rebuild #6 (root) floppy.
Insert a formatted floppy into the floppy drive and press
<RETURN> -->

You have a good "root" floppy!
Label this floppy "Clipper Rebuild #6 (root) (100/32C/200)"
Cleaning up...
Successful installation: REBUILD (sss0003)
Product installed in the /usr/ip32/rebuild directory
```

5. After these messages appear, press <RETURN> to return to the **newprod**(1M) menu. Key in **q** to exit the **newprod**(1M) utility.

Booting from the Rebuild Boot Floppy Disk

After you have created a set of Rebuild floppy disks with the current software, you must boot the workstation from the Rebuild Boot floppy disk.

If you use the Rebuild set delivered with the software, notice that you received the following Rebuild Boot (#1) floppy disks:

- Rebuild Boot for 100/32C/300-series (floppy #1)
- Rebuild Boot for 300/400/3000/4000-series (floppy #1)
- Rebuild Boot for 6000-series (floppy #1)

To boot from the Rebuild Boot floppy, select the Boot floppy appropriate for the machine that you will rebuild. Insert this disk in the floppy disk drive and boot the workstation/server. The machine will boot from the Rebuild floppy rather than from the hard disk.

Entering the Utility Pages

After you have booted from the Rebuild Boot floppy disk, you must enter the Utility Pages. You will use the Utility Pages to verify hard disk flaws, format the hard disk (if desired), repartition the hard disk (if desired), and load the rebuild media. Because the interface is different for workstations and servers, this section separates the procedures for workstations and servers.

Entering the Utility Pages on a Workstation

When an Intergraph workstation boots from the Rebuild floppy disk, a blue introductory screen appears. You must move the mouse or cursor within five seconds if you wish to enter the Utility Pages. From the introductory screen, select the **Utility** icon to enter the Utility Pages. If the Utility Pages have been assigned a password, you will be prompted to enter the Utility Page password.

NOTE If the Utility Pages do not have a password, you may wish to assign one for security purposes. (Only the system manager should be allowed to access the Utility Pages because this environment allows you to format and repartition the hard disk.) To assign a password to the Utility Pages, select the **Workstation Password** icon on the Main Utility Page and key in the desired password when prompted.

The Utility Pages are a series of screens that allow you to configure the system. At the bottom of each Utility Page screen, notice the **Help** icon. To use the help function, place the cursor on the item for which you want information and tap the reset button.

Entering the Utility Pages on a Server

When an Intergraph server boots from the Rebuild floppy disk, an introductory screen appears. You must press any key within five seconds to enter the Startup Utility Pages. From this screen, key in **UT** to enter the Utility Pages. If the Utility Pages have been assigned a password, you will be prompted to enter the Utility Page password.

NOTE If the Utility Pages have not been assigned a password, you may wish to assign one for security purposes. (Only the system manager should be allowed to access the Utility Pages because this environment allows you to format and repartition the hard disk.) To assign a password to the Utility Pages, key in **PW** at the Main Utility Page and key in the desired password when prompted.

The Utility Pages are a series of screens that allow you to configure the system. At the bottom of each Utility Page screen, notice the **Help** option. To use the help function, key in **h** followed by the string required to select the option. For example, to read help information on the Disk Partitioning screen, key in **h dp**. Notice that the help function is not case-sensitive.

Verifying Hard Disk Flaws

From the Utility Pages, you will be able to verify the locations of hard disk flaws by running the Verify procedure. This procedure reads the hard disk and records all flaws to ensure that data is not written over flawed areas. It does not erase or change data on the disk.

If you are rebuilding because you wish to repartition the hard disk or convert to a different file system type, you do not need to run the Verify procedure. However, if you are rebuilding the hard disk to correct bad block messages or to correct a corrupted disk, you must run the Verify procedure.

The following is an example of a bad block message:

```
Disk failed: s0u0p07.1 medium error:
read error at block 2933
```

Bad blocks are flawed areas on the hard disk. As the disk drive writes data on the disk, it calculates an Error Correction Code (ECC) for each block and stores the code at the end of each block. When the drive reads data from the disk, it calculates another ECC and compares it to the original stored value. If the codes match, the data stored in that block is intact and uncorrupted. If the codes do not match, the drive generates an ECC error and the host displays a medium error or a bad block message.

The Intergraph 156-MB CDC and 180-MB Imprimis disks have automatic read and write reallocation. This means that if the system finds an error that it cannot recover from (save the data, move it to a new location, and experience no data loss), the block will be automatically reallocated to another sector of the disk and the original data will be rewritten there (with no data loss).

However, if the system finds an unrecoverable data error while executing a read or write operation, it will report a medium error. If you receive one or more medium errors, run the Verify procedure on the disk to register and reallocate any other bad blocks that may have developed.

Some Maxtor disks have Automatic Read and Write Reallocation. Regardless of your Maxtor disk version, you will still receive a bad block message if unrecoverable data errors are encountered on the disk. When a bad block message displays, you must run the Verify procedure to reassign the bad blocks.

The Verify procedure attempts to read every location on the disk. It checks the data by comparing a computed value of ECC with a value that was computed when the data was written and stored on the disk at the end of each track. If the ECC code does not match, the procedure reads the disk again. One retry is categorized as a soft error. Two or more retries indicate a hard error. Hard errors are flagged as bad blocks. If the procedure finds a bad block, the host reassigns the bad block to a spare location on the disk.

After all bad blocks are found, the disk is clean and ready to go. A bad block indicates that a block of data has an error. As a result, the data in this block may not be recoverable.

Procedures differ for workstations and servers and are described in separate sections.

Verifying a Workstation Hard Disk

Follow these steps to Verify a workstation hard disk.

1. From the Main Utility Page, select the **Disk Maintenance Utility** screen button.
2. Select the **Next Page** screen button to access the Hard Disk Flaw Data Utility Page.
3. Select the **Verify** screen button to access the Verify page.
4. From the Verify page, check to see that the SCSI ID, Disk Capacity, and Logical Unit Number fields contain the correct settings for the disk you want to verify.

   The SCSI ID and LUN for internal hard disks are 0. The disk capacity depends on the internal hard disk size.
5. Select the Verify screen button to begin the Verify process.
6. If the hard disk is corrupted with numerous bad blocks, the Verify process may not complete. If the process is not successful, you must format the hard disk to mark all bad blocks. Proceed to "Formatting the Hard Disk." If the verification process completes successfully, you will not be required to complete step 6 in the rebuild process, "Formatting the Hard Disk."

Verifying a Server Hard Disk

Follow these steps to Verify a server hard disk:

1. From the Main Utility Page, key in the following to display the Disk Maintenance Utilities menu:

   Option: **DM**

2. Check to see that the SCSI_id, Logical_Unit_Number, and Disk Capacity fields contain the correct settings for the disk you want to verify.
3. Key in the following to select the **VeriFY** option, which runs the verification procedure.

   Option: **VFY**

4. If the hard disk is corrupted with numerous bad blocks, the Verify process may not complete. If the process is not successful, you must format the hard disk to mark all bad blocks. Proceed to "Formatting the Hard Disk."
5. If the verification process completes successfully, you will not be required to complete step 6 in the rebuild process, "Formatting the Hard Disk."

Formatting the Hard Disk

If the Verify procedure fails, you must format the hard disk. Otherwise, formatting is optional. Formatting overwrites all data on the hard disk; therefore, you should not format unless the Verify procedure failed.

Formatting involves structuring the disk so that hardware and software can communicate with the disk. This process writes a test pattern to the disk and reads it to verify the pattern, checks for any errors, and marks locations on the disk (flaws) that cause errors. These flaws are recorded to prevent any future writing to or reading from that location.

Rebuilding without formatting is possible. In many cases, only part of the hard disk is corrupted. You may be able to restore the corrupted file system(s) without formatting the hard disk. If you format, you will lose all data on the disk; if you do not format, you will lose only the data in file systems that you restore. If you do not wish to format the hard disk, proceed to "Selecting the Rebuild Utility Page Icon."

Formatting takes approximately 45 minutes for 80-MB hard disks, 90 minutes for 156-MB hard disks, 4 hours for 355-MB hard disks, and 8 hours for 670-MB hard disks.

CAUTION Formatting destroys all data on the disk. Therefore, back up all personal files before formatting.

Formatting procedures for workstations and servers differ and are described separately.

Formatting Workstation Hard Disks

Follow these steps to format workstation hard disks:

1. From the Main Utility Page, select the Disk Maintenance Utility Page.
2. When the Disk Maintenance Utility Page appears, select the following settings for an internal hard disk:

```
SCSI ID = 0
```

```
Logical Unit Number= 0
```

For an external hard disk, you would set the SCSI ID to 1, 2, or 3, and the LUN to 0.

3. Match the Intergraph Part Number on the Hard Disk Profile Sheet with the part number in the Disk Type roll-through box.
4. Key in the serial number from the Hard Disk Profile Sheet if it does not display in the Serial Number data entry field.
5. Select the **Next Page** screen button to access the Hard Disk Flaw Data Utility Page. Select the following settings:

   ```
   Format With Flaw Data    = Yes
   ```

   Selecting "yes" for the Format With Flaw data option prevents flaws on the disk from being ignored during the format process.

   ```
   Verify on Format         = Yes
   ```

   Selecting "yes" for the Verify on Format option runs the Verify procedure to record any new flaws found on the disk during the format.
6. Select the **Format** screen button to access the red Format Utility Page.
7. Select the **Format** screen button to start the procedure. While the format procedure is running, messages revealing the part of the disk being verified and identifying the bad blocks appear on the screen. You do not need to respond to any of these messages. The message "Format Complete" appears when formatting is finished. Select the Previous Page screen button to return to the Hard Disk Flaw Data Utility Page.

Formatting Server Hard Disks

Follow these steps to format an InterServe hard disk:

1. From the Main Utility Page, key in **DM** to display the Disk Maintenance Utilities menu.

2. Key in the following to set the SCSI ID to 0:

   Option: **SCSI 0**

3. Key in the following to set the logical unit number to 0:

   Option: **LUN 0**

4. Key in the following, where *serial-number* is the correct serial number for the disk as the Hard Disk Profile Sheet indicates:

   Option: **SN** *serial-number*

5. Key in the following so that flaw data will be considered during the format:

   Option: **FWF Y**

6. Key in the following for the flaw data list to be verified during the format and for any additional bad blocks to be added to the list and registered:

   Option: **FWV Y**

7. Key in the following to start the format procedure:

   Option: **FORMAT**

   While the format procedure is running, messages revealing the part of the disk being verified and identifying the bad blocks appear on the screen. The message "Format Complete" appears when formatting is finished.

Selecting the Rebuild Utility Page Icon

After verifying and formatting the hard disk, the next step in the rebuild process is selecting the Rebuild Utility Page from the Main Utility Page. Do so by selecting the **Rebuild** icon on the Main Utility Page (for workstations) or keying in **REBUILD** at the Main Utility Page (for servers).

The Rebuild Utility Page allows you to repartition, load Rebuild media, and enter the Rebuild environment. Before you can begin restoring the file systems, you must have specified an acceptable partition table, successfully loaded the Rebuild media, and selected the icon to enter the Rebuild environment. If any of these steps has not been completed, you will not be allowed to continue the rebuild process.

Notice that the left portion of the Rebuild Utility Page displays messages concerning your current status and your next step. While you are using the Rebuild Utility Page, refer to this message area for brief instructions.

Repartitioning the Hard Disk

From the Rebuild Utility Page, you may repartition the hard disk. This step is not required unless the current partitions are unacceptable for the rebuild procedure.

Partitioning logically divides the hard disk into separate sections or devices for specific areas of user and operating system functions. You must repartition the hard disk only if the current partitions are not acceptable for the Rebuild software. You may also repartition to increase or decrease the size of specific partitions or to add nonstandard partitions. For example, you may need to add a stash partition to produce large plots on raster output devices. See the *InterPlot User's Guide* for more information on creating a stash partition.

NOTE The Rebuild Utility Page refers to the partitions on the disk as the partition table. This documentation will use the terms partitions and partition table interchangeably.

During the rebuild process, you must choose one of the following options concerning hard disk partitions:

- Use the current partitions.
- Repartition using Intergraph-defined default partitions.
- Repartition using custom partitions.

Using the Current Partition Table

If the current partition table is acceptable for the Rebuild software and you do not wish to alter it, you will not need to repartition the hard disk. Partitions must meet the following requirements before the Rebuild software will accept them:

- The root partition must have at least 25,000 blocks.
- The swap partition must have at least 15,000 blocks.

- The usr partition must exist.

NOTE Other software products may have additional partition requirements. The system manager should determine the partition sizes appropriate for the system and the software the system uses. Refer to "Partitioning Overview" for information on factors that you should consider when you establish partition sizes.

Follow these steps if you wish to use the current disk partitions:

1. Access the Rebuild Utility Page.
2. If the messages on the left portion of the screen indicate that the current partitions are not acceptable, you must repartition the hard disk. Proceed to "Repartitioning Using Default Partitions" or to "Repartitioning Using Custom Partitions."
3. If the messages on the left portion of the screen indicate that the partitions are acceptable, proceed to "Loading the Rebuild Media."

Repartitioning Using Default Partitions

If you are not currently using the (Intergraph-defined) default partitions but wish to, you must repartition the hard disk using the Default option supplied on the Rebuild Utility Page. Be aware that you will lose all data on any partition that you alter.

Intergraph has established the following default partition sizes:

	80 MB	156 MB	355 MB	584 MB	670 MB
boot:	3988	7988	7988	7988	7988
root:	25000	25000	25000	25000	25000
swap:	27360	71000	71000	71000	71000
PC-DOS:	5000	none	none	none	none
usr:	100000	200600	590298	1037988	1204900
usr2:	none	none	none	none	none

NOTE The total number of blocks in the partition table and the disk capacity differ, because a one block header is associated with each partition. The header blocks are included in the disk capacity count and not in the partition table count.

Follow these steps to repartition using default partitions:

1. Access the Rebuild Utility Page.
2. Select the **Default** icon under the Create Partition Table heading. This menu option will automatically redefine all hard disk partitions according to the defaults. You will lose all data in any partition that is altered.
3. If the messages on the left portion of the screen indicate that the partitions are acceptable, proceed to "Loading the Rebuild Media."

Repartitioning Using Custom Partitions

This section describes repartitioning the hard disk using custom partitions rather than using the Intergraph-defined default partitions described in the previous section. For example, you may want to create a usr2 or usr3 partition by taking space from usr. Be aware that you will lose all data on any partition that you alter.

- If you are not familiar with the concept of repartitioning the hard disk, read "Partitioning Overview" before you begin actually repartitioning the hard disk.
- If you are familiar with the concept of repartitioning the hard disk, proceed to "Creating Nonstandard Partitions."

Partitioning Overview

This section provides information concerning the following topics:

- Determining partition sizes

- Reallocating space between partitions
- Understanding partition names

Determining Partition Sizes

Intergraph divides workstation/server hard disks into the following partitions:

- boot
- root
- swap
- DOS (32C workstations only)
- usr

NOTE **Intergraph does not automatically place a stash partition (for plotting) on hard disks. However, some plotters require stash on the plot server. A stash may be either a partition on the hard disk or physical memory. Refer to the *InterPlot User's Guide* for information on creating a stash partition.**

The boot partition contains boot images, including the hardware diagnostics test, Utility Pages, and CLIX kernel software. This partition contains 3,988 blocks for workstations with 80-MB hard disks and 7,988 blocks for all other hard disk sizes. You cannot access this partition on the Disk Partitioning Utility Page because you cannot alter its size.

The root partition contains a major portion of the CLIX operating system. By default, this partition contains 25,000 blocks. You should not take space from this partition, and if you receive messages saying that the root (7.0) partition is out of space, you must add space to it.

The swap partition is used for swapping portions of memory to the hard disk. By default, it contains 27,360 blocks for 80-MB hard disks and 71,000 blocks for all other hard disk sizes. Space can be removed from the swap partition to add to another partition. However, some applications will not run without a minimum amount of swap space. This minimum amount varies among applications.

The DOS partition is included on 32C workstations only. If you will not use the PC-DOS partition on your workstation or if you are using SoftPC on your workstation, you may remove the DOS partition and add the space to another partition such as usr. By default, this partition contains 5,000 blocks.

The usr partition contains user directories, most products, and any other directories and files that users access. By default, the usr partition contains 100,000 blocks for 80-MB hard disks; 200,600 blocks for 156-MB hard disks; 590,298 blocks for 355-MB hard disks; 1,037,988 blocks for 584-MB hard disks; and 1,164,276 blocks for 670-MB hard disks. Because this partition is the user's work partition, it contains the most free space. You may wish to remove space from usr and form a usr2 partition.

You must have a stash on your plot server to plot on most plotting devices. A stash may be either a partition on your hard disk or physical memory. (Physical memory may be used only if sufficient physical memory is available on your server. See section 1.13 in the *InterPlot User's Guide*.)

Reallocating Space Between Partitions

To take space from one partition and add it to another, those two partitions must be adjacent on the disk, and the partition receiving the additional space must be positioned to the left of (behind) the partition giving up space.

By default, the standard partition order on hard disks is as follows:

For 32/32C workstations only:

root	swap	PC-DOS	usr

For all other workstations and servers:

root	swap	usr

For example, on a 32C workstation you could remove the DOS partition and add it directly to the swap partition (since these partitions are adjacent). However, you could not remove the DOS partition and add it directly to root. You could add the DOS space to root only by adding it to the swap partition and then removing it from swap and adding it to root.

In addition, when you add a partition, the system will only search forward on the disk for space to claim. That is, to remove the DOS partition (on a 32C workstation) and add that space to the usr partition, you could not remove the DOS partition and add a larger usr partition. Doing so would cause the system to search for and allocate space at the beginning and beyond the usr partition. Instead, you would need to remove the DOS and the usr partition and then add a larger usr partition. In this case, the system would start allocating space from the beginning of the (former) DOS partition.

CAUTION Repartitioning the hard disk loses any information in a moved, removed, or resized partition. Therefore, back up all files before repartitioning.

Understanding Partition Names

Partition names appear in the following form on the Disk Partitioning Utility Pages:

sAuBpC.D

The following table defines the components of this partition name:

Labels	Variables
s = SCSI ID	A: SCSI ID number
u = unit	B: unit (drive) number
p = partition	C: general-purpose division number (partition)
	D: modifier number (subpartition)

Intergraph specifies the following general-purpose division numbers on the hard disk:

2 =	flaw information partition
4 =	error log
5 =	configuration information partition
7 =	CLIX partition
8 =	boot code partition
9 =	DOS partition
A =	diagnostic software partition
F =	stash partition

For example, a CLIX partition begins with a 7 (as in s0u0p7.3).

Intergraph defined the following general-purpose partition and modifier numbers on the hard disk:

7.0 = root partition
7.1 = swap partition
7.3 = usr partition
7.4 = usr2 partition

Combining the partition and modifier allows each of the possible 16 general-purpose partitions to be divided into 16 subpartitions for a total of 256 virtual disks.

Creating Nonstandard Partitions

This section supplies the general steps for creating new partitions. The system manager should decide how much space must be allocated for each partition. Follow steps in this section for adding partitions and altering partition sizes. For more information on creating a stash partition, refer to the *InterPlot User's Guide.*

CAUTION **Repartitioning erases all data on any altered partition. Therefore, back up all files before repartitioning.**

Repartitioning Workstation Hard Disks

Follow these steps to repartition workstation hard disks:

1. Select the **Custom** icon from the Rebuild Utility Page. This selection will transfer control to the Disk Partitioning Utility Page.

2. Notice the chart displaying the hard disk partitions at the top of the Disk Partitioning Utility Page. Study this chart to determine how you wish to alter the partitions. Keep in mind the following restrictions enforced by the Rebuild software:

   - The root partition must have at least 25,000 blocks.

- The swap partition must have at least 15,000 blocks.
- The usr partition must exist.

NOTE Other software products may have additional partition requirements. The system manager should determine the partition sizes appropriate for the system and the software the system uses. Refer to "Determining Partition Sizes" for information on factors you should consider when you establish partition sizes.

3. From the Disk Partitioning Utility Page, you can add and remove partitions. Thus, to change partition sizes, you must remove all partitions being altered and then add them back with new sizes.

   To remove a partition, scroll through the Partition Names roll-through box until the name of the partition to remove appears. If the desired name is not in the roll-through box, key in the Partition Number and the Modifier Number (in their respective fields) for the partition to be removed. Select the Remove Partition button to remove the partition currently displaying in the Partition Names roll-through box.

   Enter the name of the new partition in the Partition Names box. Then, enter the Size in Blocks of the new partition. To add the specified partition, select the Add Partition button. The partition will be added in the first available slot.

NOTE If you plan to make the size of the last partition equal to the amount of available free space listed in the partition table, you must subtract two from the amount of free space. (The partition header and the "End of Disk" header require one block each.)

   Notice that the bar chart at the top of the screen reflects adjustments to the disk partitions.

4. After you have defined all hard disk partitions, select the **Utility** icon to return to the Main Utility Page and then the **Rebuild** icon to return to the Rebuild Utility Page.

   Keep in mind that you must create a device file and then create and mount a file system on any nonstandard partition (such as usr2 or usr3) that you add. These steps are described in "Preparing

Nonstandard Partitions for Use."

Repartitioning an InterServe Hard Disk

Follow these steps to repartition an InterServe hard disk:

1. Select the **Custom** icon from the Rebuild Utility Page. This selection will transfer control to the Disk Partitioning Utility Page.

2. Key in the following to list the partitions currently on the hard disk:

   Option: **LP**

   Keep in mind the following restrictions enforced by the Rebuild software:

   - The root partition must have at least 25,000 blocks.
   - The swap partition must have at least 15,000 blocks.
   - The usr partition must exist.

NOTE Other software products may have additional partition requirements. The system manager should determine the partition sizes appropriate for the system and the software the system uses. Refer to "Determining Partition Sizes" for information on factors that you should consider when you establish partition sizes.

3. From the Disk Partitioning Utility Page, you can add and remove partitions. Thus, to change partition sizes, you must remove all partitions being altered and then add them back with new sizes.

   To remove a partition from an InterServe hard disk, first key in the following to list the existing partitions:

   Option: **LP**

   Next, write down the partition and modifier numbers (such as 7 and 3 for usr) for the partitions you will remove.

   Then, key in the following command line to remove a partition, substituting the appropriate values in each field:

   Option: **RMP** *partition-number modifier-number*

   Components of this command line are defined as follows:

- **RMP** is the ReMove_Partition option, which removes the partition designated on the same line from the disk.
- *partition-number* is the number of the partition to be removed.
- *modifier-number* is the modifier number of the partition to be removed.

To add a partition to a server hard disk, first key in the following to select the Add_SYStem option. This option reserves the first 7988 blocks of disk space for the bootable partitions and images that you must create later when rebuilding the hard disk with the Rebuild floppy disk set (described in "Restoring the File Systems").

Option: **ASYS**

Key in the following to add each desired partition to the disk:

Option: **ADDP** *partition-number modifier-number size*

NOTE

If you plan to make the size of the last partition equal to the amount of available free space listed in the partition table, you must subtract two from the amount of free space. (The partition header and the "End of Disk" header require one block each.)

4. Key in the following to select the List_Partitions option, which lists existing partitions and verifies that the desired partitions have been added:

   Option: **LP**

5. After you have defined all hard disk partitions, key in **UT** to return to the Main Utility Page and then **REBUILD** to return to the Rebuild Utility Page.

   Keep in mind that you must create a device file and then create and mount a file system on any nonstandard partition that you add (such as usr2 or usr3).

Loading the Rebuild Root Media

The next step in the rebuild process is to load the Rebuild Root media. This step loads minimal file systems to the hard disk. The minimal file systems are stored temporarily in swap space. When you begin actually restoring the file systems, these file systems will be copied from swap to the appropriate partitions.

Follow these steps to load the Rebuild Root media:

NOTE You can rebuild your workstation/server with different media types, including floppy disk, microfloppy, tape, or CDROM. This section will provide instructions for loading Rebuild floppy disks; however, the interface is similar for all media types.

1. Locate (or create) the Rebuild Root media. If your requested media type is floppy disk, locate or create the five Rebuild Root floppy disks.
2. Select the appropriate media type (for example, floppy) from the roll-through box in the Load Rebuild Root Media portion of the Rebuild Utility Page. If you are rebuilding with 3½ -inch floppy disks, select the microfloppy media type.
3. Insert one of the Rebuild Root floppy disks in the drive and select the **Load** icon. The Rebuild software does not require you to load the Rebuild Root floppies in any specific order. You may load the media in any order, as long as you load each floppy.
4. As the media is being loaded on the hard disk, notice that the blocks remaining to be read display in the Blocks Remaining menu box. This menu box is purely informational; you do not need to respond to it.
5. When the floppy has been loaded, remove it from the drive and insert the next floppy. Continue this process until all Rebuild Root floppies have been loaded on the hard disk.

NOTE

If the system reports a "CRC error" while you are loading the media, try again to load the media. If the media still does not load successfully, recreate the Rebuild Root disk.

Entering the Rebuild Environment

After loading the Rebuild media, you may enter the Rebuild environment. After you have loaded the Rebuild media on the hard disk, select the **Boot** icon on the Rebuild Utility Page. This icon will boot the workstation into single-user mode and initiate the Rebuild menu. You do not need to boot from the Rebuild floppy disks because the Rebuild media has already been loaded on the hard disk (in swap space). Thus, you may remove Rebuild media from the disk drive before you boot into the Rebuild environment.

Restoring the File Systems

After you have entered the Rebuild environment, you may restore minimal file systems to the hard disk. Restoring minimal **root** and **/usr** file systems prepares the hard disk so that software can be loaded through the **newprod**(1M) utility.

The steps in this section differ depending on the extent you need to rebuild. Depending on the problem with your hard disk, you may need to restore only **root**, or you may need to restore **root** and **/usr**. This section provides guidelines for a partial rebuild. However, this section focuses on performing a complete rebuild (restoring the **root** and **/usr** file systems).

Follow these steps to restore the file systems:

1. After the workstation/server boots to single-user mode, the following menu appears:

```
Rebuild options:

1. Rebuild the ROOT (7.0) and USR (7.3) file systems.
2. Rebuild the hard disk ROOT file system.
3. Rebuild the hard disk USR file system.
4. Make a hard disk fast file system (other than ROOT or USR).
5. Boot the system.
q. Quit the Rebuild utility.

Enter selection(s) separated by spaces --->
```

   The appropriate response to this prompt depends on the extent you need to rebuild. Before you decide which menu option to choose, read the description for each option carefully.

   Option 1. Rebuild the ROOT (7.0) and USR (7.3) file systems.

   The first option restores the **root** and **/usr** file systems. This option allows you to create **/usr** as a Fast File System. (See the description of option 4.) If you select this option, you will lose all data on the disk, including personal files.

   Option 2. Rebuild the hard disk ROOT file system.

   This option creates a minimal **root** file system. Unless you know that the problem is in a file system besides **root** (such as **/usr** or **/usr2**), this menu option should be your initial choice. After choosing option 2, reboot, load software from **newprod**(1M), and then reboot again. If this menu choice does not correct the problem on the hard disk, return to the Rebuild options menu and select option 3.

Option 3. Rebuild the hard disk USR file system.

This option creates a minimal **/usr** file system; thus, it overwrites all files (including any personal files) in the current **/usr** file system. It allows you to create **/usr** as a Fast File System. You should choose this option if option 2 did not solve the problem or if you know that the **/usr** file system is corrupted. After the **/usr** file system has been restored, reboot, load software from **newprod**(1M), and reboot again. If this menu option does not solve the hard disk problem, return to the Rebuild options menu and select option 1.

Option 4. Make a hard disk fast file system (other than ROOT or USR)

This option creates **/usr2**, **/usr3**, or **/usr4** as a Fast File System. Fast File Systems contain 8K-byte units, whereas standard file systems (S51K) contain 1K-byte units. For this reason, Fast File Systems read large files more efficiently than standard S51K file systems do. Any file system can be created as a Fast File System. Because this option creates a new file system on the partition, any data on the partition will be lost.

NOTE

The RFS product does not support FFS. Thus, if you use RFS, do not create any file system as a FFS.

Option 5. Boot the workstation

This option boots the workstation. You will need to select this option after completing option 2, 3, or 4. You do not need to select this option after completing option 1 because option 1 automatically reboots after file systems are restored.

Option q. Quit the Rebuild utility

This option exits the Rebuild options menu and leaves you in single-user mode. In single-user mode, you can key in **Rebuild** to return to the Rebuild options menu or **boot** to reboot the system. In addition, you can use a limited number of commands such as **mount**(1M), **tar**(1), and **vi**(1).

2. Select the option(s) of your choice at the Rebuild options menu.

```
Rebuild options:

1. Rebuild the ROOT (7.0) and USR(7.3) file systems.
2. Rebuild the hard disk ROOT file system.
3. Rebuild the hard disk USR file system.
4. Make a hard disk fast file system (other than ROOT or USR).
5. Boot the workstation.
q. Quit the Rebuild utility.

Enter selection(s) separated by spaces --->
```

This documentation assumes that you needed to rebuild your hard disk entirely, including all file systems. The following prompt appears:

```
Do you want to make root a Fast File System (y/n)? [n]
Do you want to make usr a Fast File System (y/n)? [y]:
```

3. Key in **y** at each prompt if you want to create root and **/usr** as Fast File Systems (unless you use RFS or Informix).

   Once the rebuild procedure is complete, the following prompt appears on the screen:

```
Rebooting the workstation.
WARNING: Insert the Rebuild #1 Boot floppy.
Press <RETURN> when ready to boot --->
```

Before you press <RETURN>, insert the Rebuild Boot floppy (#1) in the disk drive.

4. Press <RETURN>. Messages similar to the following appear briefly:

   ```
   Syncing disks . . .
   INIT: Single User Mode

   System Reboot in progress . . .
   ```

   With the Rebuild Boot floppy disk inserted, the system boots.

CAUTION Do not remove the Rebuild Boot floppy until after you initiate the **newprod(1M)** utility.

Loading Nucleus Software with newprod

After file systems are restored and the hard disk is booted, load nucleus software using the **newprod**(1M) utility. At this point in the rebuild procedure, minimal file systems reside on the hard disk.

After loading nucleus software, you must reboot the system to initiate the new software and check to ensure that the hard disk is functional. You do not need to load application software until the system is completely functional.

If you do not load software from delivery CDROM, proceed to "Loading Software."

Preparing the Hard Disk to Load Software from CDROM

Before you can load software from CDROM after a rebuild, you must complete the following steps:

1. Insert the CDROM in the caddy and the caddy in the CDROM drive.
2. If your CDROM drive is not connected to SCSI ID 4, you must create the CDROM drive device file using the **mknod**(1M) command. Refer to the *CDROM Drive User's Guide* for the appropriate command line.

   If your CDROM drive is connected to SCSI ID 4, the device file has already been created. Check the DIP switch setting on the rear of the CDROM drive to determine the SCSI ID setting. Proceed to step 3 if the drive is connected to SCSI ID 4.
3. Link the device file (**/dev/dsk/s4u0p0.0**) to the CDROM drive (**/dev/dsk/cdrom**) by keying in the following command line at the super-user prompt. This example assumes that the CDROM drive is connected to SCSI ID 4. If it is not, substitute the proper device file for **/dev/dsk/s4u0p0.0.**

   ```
   ln /dev/dsk/s4u0p0.0 /dev/dsk/cdrom
   ```

4. Mount the device by keying in the following at the super-user prompt:

   mount -rf FFS /dev/dsk/s4u0p0.0 /mnt

5. Copy the **runcd**(1M) utility by keying in the following:

   cp /mnt/runcd /usr/bin

6. Unmount the CDROM by keying in the following:

   umount /mnt

7. Invoke the **runcd**(1M) utility by keying in the following:

   runcd

8. At the CDROM Menu, key in **2** to invoke the **newprod**(1M) utility.

Loading Software

Follow these steps to load software:

1. After the system boots, the following prompt appears:

   ```
   CLIPPER Rebuild
   Do you wish to invoke "newprod", the product installation util-
   ity? [y]:
   ```

   Key in **y** to invoke **newprod**(1M).

2. The following prompt appears:

   ```
   Enter source of installation: n)etwork, f)loppy, t)ape or
   r)emote cdrom or local c)drom --->
   ```

   Respond to the prompt by keying in the first letter of your installation source. If you key in **c** for local CDROM, a message similar to the following may appear:

   ```
   Cannot mount /dev/dsk/cdrom on /del
   ```

   If this message appears, complete the steps in "Preparing the Hard

Disk for CDROM Software Delivery" before you begin the rebuild process.

If you keyed in **n** for network or **r** for remote CDROM, you will be prompted to enter the network connect string. After rebuilding, you cannot key in the node name of the delivery node; instead, you must key in the Ethernet address.

3. If the Rebuild Boot floppy disk (#1) is still in the floppy disk drive, remove it from the drive.
4. Load nucleus software with the **newprod**(1M) utility. Refer to the "New Product Delivery" procedure for instructions for using the **newprod**(1M) utility to deliver software.
5. After products have been loaded, reboot from the hard disk (not from the Rebuild Boot floppy). If the system boots without displaying disk errors, continue to the next step in the rebuild process ("Preparing Nonstandard Partitions for Use"). If the system does not boot correctly, return to "Restoring the File Systems" and attempt to rebuild again. This time select a different option from the Rebuild options menu.

Preparing Nonstandard Partitions for Use

After you have successfully loaded all baseline nucleus products, you must prepare all nonstandard partitions for use. If you created nonstandard partitions (such as usr2 or usr3), you may need to create a device file, create a file system on the partition, and then mount the file system before you can access the partition.

If you did not create any nonstandard partitions, proceed to "Loading Application Software with newprod."

Creating a Device File

The first step in preparing a nonstandard partition for use is creating the device file for the partition. The device file must be created in the **/dev/dsk** directory. This section explains the procedure for creating a device file.

The example in this section shows how to create a device file for the **s0u0p7.4** (usr2) internal hard disk partition. The procedure for creating a device file for an external hard disk is no different than the procedure for creating a device file for an internal hard disk except that the device file reflects a different SCSI ID for the external hard disk. The SCSI ID for an internal hard disk is usually 0; the SCSI ID for an external hard disk is usually 1, 2, or 3.

Follow these steps to create a device file:

1. Secure the following information about the partition:

   - Partition name (for example, usr2)
   - Size in blocks
   - Block major (b-maj) number
   - Block minor (b-min) number
   - SCSI ID number

   You can find this information in the Partition Table on the Disk Partitioning Utility Page. In addition, the charts on the following pages provide the partition name, size, block major number, and block minor number, if you know the SCSI ID.

Name	Device	Device File Type **	maj #	min #n
root	/dev/dsk/s0u0p7.0	b	64	112
	/dev/rdsk/s0u0p7.0	c	64	112
swap*	/dev/dsk/s0u0p7.1	b	64	113
	/dev/rdsk/s0u0p7.1	c	64	113
tmp	/dev/dsk/s0u0p7.2	b	64	114
	/dev/rdsk/s0u0p7.2	c	64	114
usr	/dev/dsk/s0u0p7.3	b	64	115
	/dev/rdsk/s0u0p7.3	c	64	115
usr2	/dev/dsk/s0u0p7.4	b	64	116
	/dev/rdsk/s0u0p7.4	c	64	116
usr3	/dev/dsk/s0u0p7.5	b	64	117
	/dev/rdsk/s0u0p7.5	c	64	117
usr4	/dev/dsk/s0u0p7.6	b	64	118
	/dev/rdsk/s0u0p7.6	c	64	118

* Not used for file systems; reserved for swap space.
** The device file can be block (b) or character (c) type.

Figure P1-1: Device File Information for Creating Partitions for SCSI ID 0, LUN 0

Name	Device	Device File Type **	maj #	min #
root	/dev/dsk/s1u0p7.0	b	66	112
	/dev/rdsk/s1u0p7.0	c	66	112
swap*	/dev/dsk/s1u0p7.1	b	66	113
	/dev/rdsk/s1u0p7.1	c	66	113
tmp	/dev/dsk/s1u0p7.2	b	66	114
	/dev/rdsk/s1u0p7.2	c	66	114
usr	/dev/dsk/s1u0p7.3	b	66	115
	/dev/rdsk/s1u0p7.3	c	66	115
usr2	/dev/dsk/s1u0p7.4	b	66	116
	/dev/rdsk/s1u0p7.4	c	66	116
usr3	/dev/dsk/s1u0p7.5	b	66	117
	/dev/rdsk/s1u0p7.5	c	66	117
usr4	/dev/dsk/s1u0p7.6	b	66	118
	/dev/rdsk/s1u0p7.6	c	66	118

* Not used for file systems; reserved for swap space.
** The device file can be block (b) or character (c) type.

Figure P1-2: Device File Information for Creating Partitions for SCSI ID 1, LUN 0

Name	Device	Device File Type **	maj #	min #
root	/dev/dsk/s2u0p7.0	b	68	112
	/dev/rdsk/s2u0p7.0	c	68	112
swap*	/dev/dsk/s2u0p7.1	b	68	113
	/dev/rdsk/s2u0p7.1	c	68	113
tmp	/dev/dsk/s2u0p7.2	b	68	114
	/dev/rdsk/s2u0p7.2	c	68	114
usr	/dev/dsk/s2u0p7.3	b	68	115
	/dev/rdsk/s2u0p7.3	c	68	115
usr2	/dev/dsk/s2u0p7.4	b	68	116
	/dev/rdsk/s2u0p7.4	c	68	116
usr3	/dev/dsk/s2u0p7.5	b	68	117
	/dev/rdsk/s2u0p7.5	c	68	117
usr4	/dev/dsk/s2u0p7.6	b	68	118
	/dev/rdsk/s2u0p7.6	c	68	118

* Not used for file systems; reserved for swap space.
** The device file can be block (b) or character (c) type.

Figure P1-3: Device File Information for Creating Partitions for SCSI ID 2, LUN 0

Name	Device	Device File Type **	maj #	min #
root	/dev/dsk/s3u0p7.0	b	70	112
	/dev/rdsk/s3u0p7.0	c	70	112
swap*	/dev/dsk/s3u0p7.1	b	70	113
	/dev/rdsk/s3u0p7.1	c	70	113
tmp	/dev/dsk/s3u0p7.2	b	70	114
	/dev/rdsk/s3u0p7.2	c	70	114
usr	/dev/dsk/s3u0p7.3	b	70	115
	/dev/rdsk/s3u0p7.3	c	70	115
usr2	/dev/dsk/s3u0p7.4	b	70	116
	/dev/rdsk/s3u0p7.4	c	70	116
usr3	/dev/dsk/s3u0p7.5	b	70	117
	/dev/rdsk/s3u0p7.5	c	70	117
usr4	/dev/dsk/s3u0p7.6	b	70	118
	/dev/rdsk/s3u0p7.6	c	70	118

* Not used for file systems; reserved for swap space.
** The device file can be block (b) or character (c) type.

Figure P1-4: Device File Information for Creating Partitions for SCSI ID 3, LUN 0

2. Boot, log in, and access the super-user account as follows:

```
login: username
$ su
#
```

NOTE If you restored the /usr file system(s), user accounts will no longer exist. Log in using the sys account. You will need to restore user accounts by recovering the /etc/passwd and /etc/group files from the previous backup or recreate user accounts.

3. Change to the **/dev/dsk** directory as follows:

```
cd /dev/dsk
```

4. List the contents of this directory to verify that a file with the same name as the new partition (**s0u0p7.4** in this example) does not already exist.

5. If this device file does not exist, create it by using the **mknod**(1M) command. This command creates a device file, or special file.

   The syntax for this command is as follows, where *name* is the device name, *type* is the device file type represented by the character **b** (for block) or **c** (for character), *b-maj* is the block major number, and *b-min* is the block minor number:

   ```
   /etc/mknod /dev/dsk/name type b-maj b-min
   ```

   For example, to create a CLIX device file for partition **s0u0p7.4** (for usr2), key in the **mknod**(1M) command and the partition information obtained in step 1 as follows:

   ```
   /etc/mknod /dev/dsk/s0u0p7.4 b 64 116
   ```

6. List the contents of the root (/) directory with the following command to ensure that a **/usr2** directory (or name of another **/usr** directory that you are creating) does not already exist.

   ```
   ls /
   ```

7. If this directory does not exist, create it by keying in the following. (This example creates the **/usr2** directory.)

```
mkdir /usr2
```

Creating a File System on a Partition

Once a partition has a device file, a file system must be created on the partition. You can create one of two types of file systems: a standard file system (S51K) or a Fast File System (FFS). S51K file systems contain 1K-byte units, whereas Fast File Systems contain 8K-byte units. For this reason, Fast File Systems read large files more efficiently than standard S51K file systems do. The most efficient use of Fast File Systems is for file systems that contain many large files such as design files. Thus, you can convert file systems such as **/usr2** to Fast File Systems to store and access large files efficiently.

Creating a Standard File System

Follow these steps to create a standard file system on a partition:

1. Use the **mkfs**(1M) command to create a standard file system. The syntax for this command is as follows, where *name* is the partition name and *size* is the partition size (in blocks):

   /etc/mkfs /dev/dsk/*name size*

   For example, to create a file system on the **s0u0p7.4** (usr2) partition, you could key in the following at the super-user prompt:

   ```
   /etc/mkfs /dev/dsk/s0u0p7.4 100000
   ```

   Messages similar to the following appear:

   ```
   MKFS: /dev/rdsk/s0u0p7.4
   (DEL if wrong)
   ```

   After approximately five seconds, messages similar to the following appear:

   ```
   bytes per logical block = 1024
   ```

```
total logical blocks = 21800
total inodes = 5440
gaps (physical blocks) = 1
cylinder size (physical blocks) = 128

mkfs: Available blocks = <size of partition>
```

These messages confirm that the file system has been created.

2. After creating a file system on the desired partition, run **labelit**(1M) to label the partition and attach a logical base directory name to the file system. For instance, to run **labelit**(1M) on the **s0u0p7.4** device (usr2), key in the following at the super-user prompt, where the first usr2 represents the file system's mounted name and the second represents the volume name (which is user-definable):

   /etc/labelit /dev/dsk/s0u0p7.4 usr2 usr2

Creating a Fast File System

Follow these steps to create a Fast File System on a partition:

1. To create a Fast File System use the **newfs**(1M) command. The syntax for this command consists of the following, where *name* is the partition name and *disk* is the disk type:

   /etc/newfs /dev/dsk/*name disk*

   Notice that this command requires a disk type. The following chart displays the disk types that Intergraph supplies.

Disk Size	Disk Type
80 MB (Quantum)	FDSK150
80 MB (Priam)	FDSK131
156 MB	FDSK155
355 MB	FDSK226
584 MB	FDSK211
670 MB	FDSK230

   For example, key in the following at the super-user prompt to create a file system on the **s0u0p7.4** (usr2) partition for a 156-MB hard disk:

```
/etc/newfs /dev/dsk/s0u0p7.4 FDSK155
```

Unlike the **mkfs**(1M) command, the **newfs**(1M) command does not require the file system size. Instead, it derives the file system size from the partition size.

2. After creating a file system on the desired partition, run the **labelit**(1M) program to label the partition and attach a logical base directory name to the file system. For instance, to run **labelit**(1M) on the **s0u0p7.4** device (usr2), key in the following at the super-user prompt where the first usr2 represents the file system's mounted name and the second represents the volume name (which is user-definable):

```
/etc/labelit /dev/dsk/s0u0p7.4 usr2 usr2
```

Mounting a Partition

To access and use a partition, you must mount it on a base directory. Mounting a partition connects the partition to the existing file system, allowing it to be accessed. A partition must be mounted on a base directory (an empty directory that will serve as the top directory for that file system). Partitions can be mounted on standard or Fast File Systems. Follow these steps to mount a partition:

1. The syntax for the **mount**(1M) command is as follows, where *type* is the file system type, *name* is the partition name, and *directory* is the directory where the file system will reside:

   /etc/mount -f*type* **/dev/dsk/***name directory*

   For example, assuming that a standard (S51K) file system now exists on **/dev/dsk/s0u0p7.4**, you could mount the usr2 partition with the following command:

```
/etc/mount -f S51K /dev/dsk/s0u0p7.4 /usr2
```

   The following command mounts a partition with a Fast File System created on it:

```
/etc/mount -f FFS /dev/dsk/s0u0p7.4 /usr2
```

2. To mount the partition and check the file system automatically each time you boot, add the partition to the **/etc/fstab** file. Otherwise, you will need to mount the partition manually every time you boot.

   The following examples modify the appropriate operating system file so that **/dev/dsk/s0u0p7.4** mounts automatically on directory /usr2 when the workstation boots.

   For a standard file system, add a line similar to the following to the **/etc/fstab** file:

   ```
   /dev/dsk/s0u0p7.4 /usr2 S51K
   ```

   For a Fast File System, add a line similar to the following to the **/etc/fstab** file:

   ```
   /dev/dsk/s0u0p7.4 /usr2 FFS
   ```

Loading Application Software with newprod

After you have prepared any nonstandard partitions for use, you must load application software. Now that the system is functional and all partitions are mounted, you can load this software on the hard disk.

Follow these steps to invoke **newprod**(1M) and deliver software:

1. Log in to the system and access the super-user account as follows

   ```
   login: username
   $ su
   #
   ```

NOTE If you restored the /usr file system(s), user accounts will no longer exist. Log in using the **sys** account. You will need to recreate all user accounts.

2. Invoke the **newprod**(1M) utility as follows:

   newprod

   The following prompt then appears:

   ```
   Enter source of installation: n)etwork, f)loppy, t)ape or r)emote
   cdrom or local c)drom --->
   ```

   If you keyed in **n** for network or **r** for remote cdrom, you will be prompted to enter the network connect string. After rebuilding, you cannot key in the node name of the delivery node; you must key in the Ethernet address.

3. The **newprod**(1M) menu appears. From this menu, you may select (highlight) products to download. Select and load all software products that you wish to use on the system. For instructions on using the **newprod**(1M) utility, refer to the "New Product Delivery" procedure.

Restoring Files from Backups

The final step in the process of rebuilding the hard disk is restoring files from backups.

After restoring the software products with the **newprod**(1M) utility, restore all user- and site-specific files that you backed up before beginning the rebuild procedure.

Procedure 2: New Product Delivery

Overview of New Product Delivery

Purpose	To install new software or update existing software on workstations/servers
When Performed	After purchasing software or after receiving a new release of software
Starting Conditions	Log in as super-user
Commands	**newprod**(1M), **makenode**(1M), **runcd**(1M)
Media	New product tapes, floppy disks, or CDROMs
Time	2 - 10 minutes per product
Caution	Do not abort **newprod**(1M) when installing UNIX-BOOT or SYSTEMV. For multinode networks, you should be logged in to the machine you want the software installed on. Download only the products to be updated.
References	**newprod**(1M), **makenode**(1M), **runcd**(1M)

Setting Up the Delivery Source

Intergraph delivers workstation/server software upgrades on the following media types:

- VAX magnetic tape
- Cartridge tape
- 6250 tape
- CDROM

In addition, some software products and product fixes are delivered on floppy disks. You can request a specific media type by calling Intergraph Software Delivery.

After you receive the delivery medium, you must load products from the medium to each Intergraph workstation/server. If you have more than one workstation/server, you probably load software on a VAX or on one workstation/server and allow all other workstations/servers on the network to load the products from the VAX or workstation/server. The machine that other nodes load software from is called a delivery source. This section explains how to prepare a delivery source for storing products in deliverable format.

- If you use a VAX or workstation/server as a delivery source, follow the procedures in this section to prepare the delivery node.
- If you deliver software from CDROM, refer to "CDROM Software Delivery" for instructions in setting up the delivery node and delivering software.
- If you deliver directly from cartridge tape, 6250 tape, or floppy disk, no setup is necessary. Proceed to "Delivering Software Using newprod."

Setting Up an Intergraph VAX Delivery Source

Before you can download product software to the workstation/server from a VAX, you must ensure that the products have been loaded and initialized on the VAX. This section discusses procedures for loading and initializing the workstation/server software products on the VAX. The following steps are required to set up an Intergraph VAX as a delivery source:

1. Deliver workstation/server software products to the VAX.
2. Initialize the VAX-based DELTOOLS product.
3. Initialize all workstation/server products.

The information presented in these VAX setup procedures builds on and refers to the information in the *VMS Delivery Guide* (DDEL001). Refer to this guide for complete instructions on making full baseline deliveries to the VAX or for more information on product deliveries.

Delivering Workstation/Server Software Products to the VAX

The first step in setting up a VAX as a delivery source is delivering the workstation/server software products to the VAX.

The information in this section assumes that all baseline products of the VAX/VMS operating system have been delivered to your delivery VAX. If necessary, consult your VAX system manager for assistance in delivering software products to the VAX.

Follow these steps to deliver the workstation/server software products to the VAX:

1. Make an image backup of the VAX disk(s) to which you will deliver software products.
2. Load the workstation/server software products on the VAX disk from the delivery tapes using the **vaxprod** or **newprod**(1M) utility as described in the *VMS Delivery Guide*.

Initializing VAX-based DELTOOLS

The second step in setting up a VAX as a delivery node is initializing the VAX-based DELTOOLS product. When the VAX-based DELTOOLS product is initialized, the logical name PRO_DD_DELTOOLS is assigned to the DELTOOLS directory ([IGR.DELTOOLS.PRO]) so that DELTOOLS can be accessed by the logical name from this point on.

In the following example and throughout this guide, ZFA0 will be the VAX disk on which software has been loaded. Key in the examples as shown, substituting your VAX disk where required.

Key in the following commands to initialize VAX-based DELTOOLS:

```
set def ZFA0:[IGR.DELTOOLS.PRO]
@initial

Initialize DELTOOLS
Configure DELTOOLS
DELTOOLS-I-CONFIG_EXIT, operation complete
DELTOOLS-I-INITIAL_EXIT, operation complete
```

Initializing Workstation/Server Products

The final step in setting up the VAX as a delivery node is initializing all workstation/server products. When these products are initialized on the VAX, the WS_INITIAL utility found in PRO_DD_DELTOOLS is executed. WS_INITIAL will either read the PRODUCT.DEF file or execute the INITIAL.COM file found in each product's directory. WS_INITIAL will then call WSPROD_ENTRY, which will enter the product into the product list. This list resides in the PRO_DD_CFG directory under the name of either WS_S.PRODS or WS_I.PRODS.

Key in the following commands to initialize all workstation products:

```
set def pro_dd_deltools
@ws_initial

NEW Usage:
@WS_INITIAL ZFA3:[WS_S.AC*]
This will initialize all WS products on ZFA3 that start with "AC"
@WS_INITIAL ZFA3:[WS_*.*]
This will initialize all WS products (WS_I,WS_S) on ZFA3
* NOTE:  Old usage will still work

For which CPU do you want to initialize
I=NS32 CLIPPER, S=C100 CLIPPER? [S]: S
Which disk are the WS_S products on? [ZFA0:]:ZFA0
Executing the initial.com in ZFA0:[WS_S.ACTEM]...
INITIAL — ACTEM/CLIPPER WORKSTATION
Reading the product.def in ZFA0:[WS_S.MSTATION]...
Reading the product.def in ZFA0:[WS_S.UNIXBOOT]...
Reading the product.def in ZFA0:[WS_S.CGH]...</SD>
```

Setting Up a Workstation/Server Delivery Source

You can set up a workstation or server as a delivery source. Then, other workstations/servers can load products from the delivery source through the network.

The following steps are required to set up an Intergraph workstation/server as a delivery source:

1. Load delivery node utilities.
2. Load software through the **makenode**(1M) utility.

Delivery node utilities make setting up a workstation/server delivery source easier. These utilities are located in the DELTOOLS product. Thus, to create a delivery source, first load DELTOOLS through **newprod**(1M). Then, use the **makenode**(1M) utility to load the products that will be available for other workstations and servers to download. As products are downloaded through **makenode**(1M), the delivery node utilities initialize them.

Products downloaded with the **makenode**(1M) utility are placed in the **/usr/ws_s** directory (unless otherwise specified). Using a workstation/server as a delivery source consumes a large amount of disk space because some products will exist in two places on the disk: **/usr/ip32** (in executable, run-time format) and **/usr/ws_s** (in compressed, deliverable format).

Loading Delivery Node Utilities

The delivery node utilities, which reside in the **/usr/ip32/deltools** directory, enable you to set up a workstation/server as a delivery source. These utilities enable products to be initialized as they are downloaded. Thus, you should load these utilities on the workstation/server before you deliver software to the delivery source through the **makenode**(1M) utility. Follow these steps to load the delivery node utilities:

NOTE Refer to "Delivering Software Using newprod" for detailed instructions for using newprod(1M).

1. Log in and access the super-user account.
2. Invoke the **newprod**(1M) utility as follows:

   newprod

3. Respond to the following prompt:

```
Enter source of installation: n)etwork, f)loppy, t)ape,
        r)emote cdrom or local c)drom: —>
```

   Key in **n** to use the network.
   Key in **f** to download from floppy disk.
   Key in **t** to download from tape.
   Key in **r** to download from remote CDROM.
   Key in **c** to download from CDROM.

   You can key in the first letter of each installation option or the entire word. For example, you can key in **n** or **network** to use the network. **newprod**(1M) accepts upper- or lower-case letters.

4. If you are downloading from the network, respond to the following prompts concerning the network address and the *username.password* combination. Then, proceed to step 6.

```
Enter network connect string or address
                (08-00-36-XX-XX-XX) —> 08-00-36-12-34-56
```

   You can enter the network address or node name at the connect string prompt.

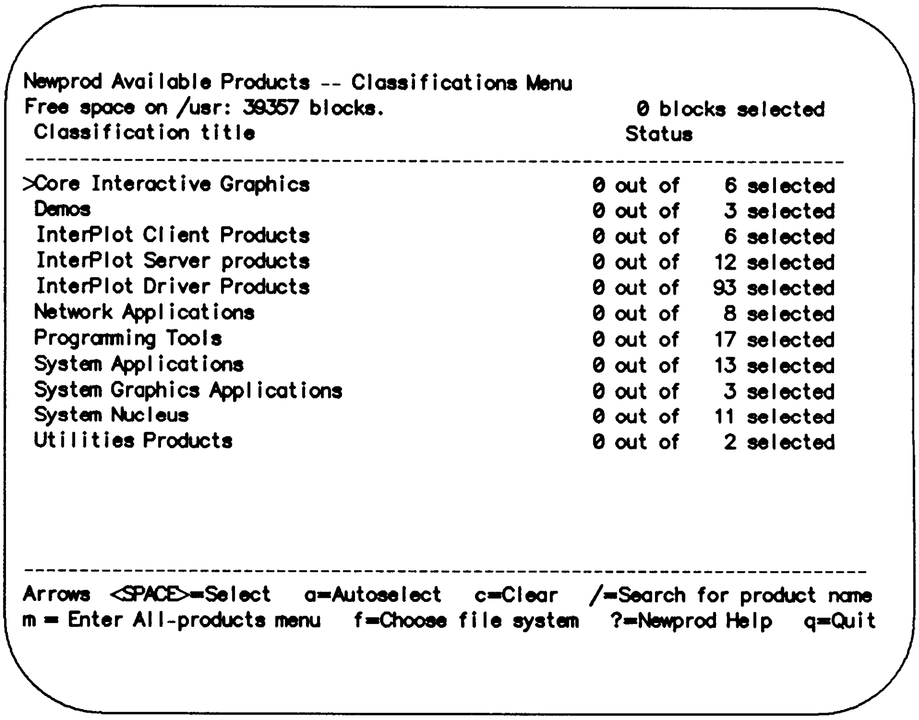
```
Enter "username.password" combination for
                "08-00-36-XX-XX-XX" —> john.doe
```

   If you do not want to display the password, key in *username* followed by a period at the prompt and press <RETURN>. The system automatically prompts for a password, and the password does not display as it is keyed in.

5. If you are downloading from tape or floppy disk, respond to the following prompt:

```
Insert a newprod product floppy/tape and press <RETURN> --->
```

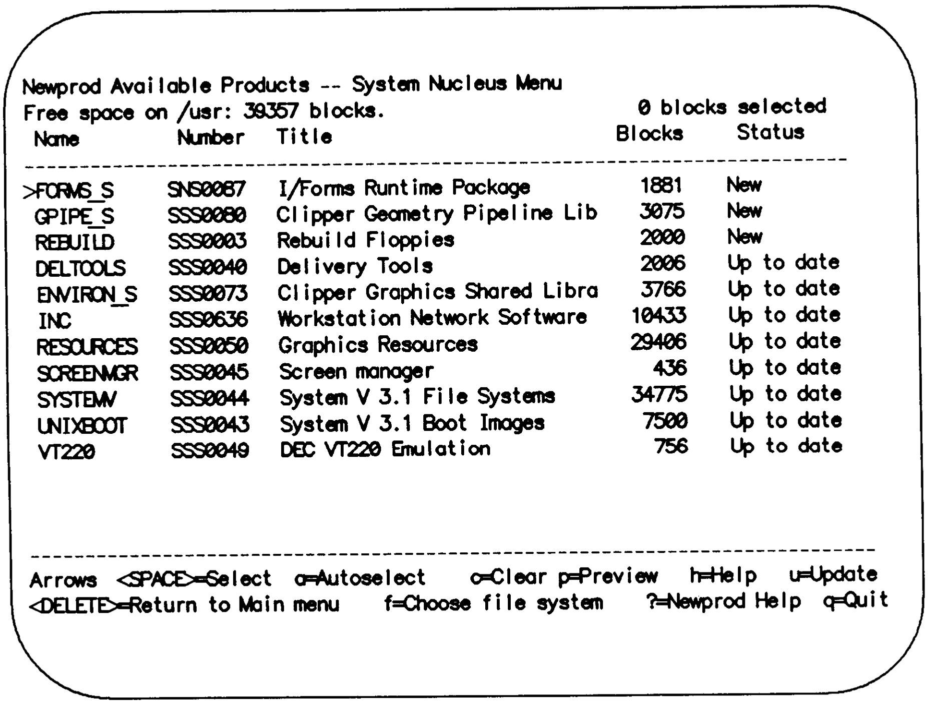
6. The Newprod Available Products -- Classifications Menu appears. This menu contains product groups, or classifications.



```
Newprod Available Products -- Classifications Menu
Free space on /usr: 39357 blocks.                    0 blocks selected
 Classification title                               Status
--------------------------------------------------------------------------
>Core Interactive Graphics                         0 out of   6 selected
 Demos                                             0 out of   3 selected
 InterPlot Client Products                         0 out of   6 selected
 InterPlot Server products                         0 out of  12 selected
 InterPlot Driver Products                         0 out of  93 selected
 Network Applications                              0 out of   8 selected
 Programming Tools                                 0 out of  17 selected
 System Applications                               0 out of  13 selected
 System Graphics Applications                      0 out of   3 selected
 System Nucleus                                    0 out of  11 selected
 Utilities Products                                0 out of   2 selected



--------------------------------------------------------------------------
Arrows <SPACE>=Select  a=Autoselect  c=Clear  /=Search for product name
m = Enter All-products menu  f=Choose file system  ?=Newprod Help  q=Quit
```

Figure P2-1: Newprod Classifications Menu

To enter a classification, use the arrow keys to move the cursor to the classification you wish to enter and press the <SPACE> bar. For example, to load nucleus products (products required to run the System V operating system), move the cursor to the System Nucleus classification. To enter the desired classification, press the <SPACE> bar. The classification submenu appears. A classification submenu lists all products available in that classification.



```
Newprod Available Products -- System Nucleus Menu
Free space on /usr: 39357 blocks.                      0 blocks selected
 Name         Number    Title                          Blocks    Status
--------------------------------------------------------------------------
>FORMS_S      SNS0087   I/Forms Runtime Package         1881     New
 GPIPE_S      SSS0080   Clipper Geometry Pipeline Lib   3075     New
 REBUILD      SSS0003   Rebuild Floppies                2000     New
 DELTOOLS     SSS0040   Delivery Tools                  2006     Up to date
 ENVIRON_S    SSS0073   Clipper Graphics Shared Libra   3766     Up to date
 INC          SSS0636   Workstation Network Software   10433     Up to date
 RESOURCES    SSS0050   Graphics Resources             29406     Up to date
 SCREENMGR    SSS0045   Screen manager                   436     Up to date
 SYSTEMV      SSS0044   System V 3.1 File Systems      34775     Up to date
 UNIXBOOT     SSS0043   System V 3.1 Boot Images        7500     Up to date
 VT220        SSS0049   DEC VT220 Emulation              756     Up to date

--------------------------------------------------------------------------
Arrows <SPACE>=Select  a=Autoselect   c=Clear p=Preview   h=Help   u=Update
<DELETE>=Return to Main menu    f=Choose file system    ?=Newprod Help  q=Quit
```

Figure P2-2: Newprod Classification Submenu

7. Two steps are required to load products. You must first select products that you want to download. Then, you must issue a command to load the products that have been selected.

   To select a product to download, use the arrow keys to move the cursor to the product and then press the <SPACE> bar. Notice that the product is now highlighted (selected).

   To load highlighted products, issue the **u** (update) option. If you decide not to load a highlighted product, move the cursor to that product and press the <SPACE> bar. The product will be unhighlighted (unselected) and will not be loaded when the **u** option is issued.

8. The only product you need to download now is DELTOOLS, because an option in DELTOOLS allows you to set up the delivery source. Enter the System Nucleus classification and use the arrow keys to move the cursor to DELTOOLS.
9. Press the <SPACE> bar to select (highlight) DELTOOLS to be downloaded.
10. Key in **u** to update DELTOOLS.
11. During installation, the following prompt appears:

    ```
    Deltools contains delivery node utilities that facilitate setting
    up this node as a software delivery source for newprod.

    Do you want the delivery node utilities (y/n)? [n]:
    ```

    Respond to this prompt by keying in **y**.
12. After DELTOOLS is installed successfully, press <RETURN> to return to the menu.
13. Key in **q** to quit **newprod**(1M).

Loading Software Through the makenode Utility

The second step in setting up your workstation/server as a delivery source is loading the software through **makenode**(1M). After you load software on the designated workstation/server, other workstations and servers can load software from this delivery source through **newprod**(1M) (assuming that your workstations/servers are connected through a network).

makenode(1M) is modeled after **newprod**(1M). **makenode**(1M) has the same menu interface as **newprod**(1M). **makenode**(1M) contains all **newprod**(1M) interactive mode options and command-line options except for the following:

x - install script debug

y - accept installation defaults

d - accept installation defaults

Refer to the sections on using **newprod**(1M)/**makenode**(1M)'s interactive and command-line options for more information.

A main difference between **newprod**(1M) and **makenode**(1M) is that products delivered through **newprod**(1M) are executable. Products delivered through **makenode**(1M) cannot be executed; they are stored in deliverable format. By default, products downloaded through **makenode**(1M) are placed in **/usr/ws_s**. Each product's status in the **makenode**(1M) menu reflects how the software in the **/usr/ws_s** compares with the software on the delivery medium.

Follow these steps to use **makenode**(1M) to install products from network, floppy disk, or tape:

1. Power up the workstation/server designated as the delivery source.
2. Log in and access the super-user account.
3. To invoke **makenode**(1M), key in the following:

   makenode

4. Respond to the following prompt:

   ```
   Enter source of installation: n)etwork, f)loppy, t)ape,
           r)emote cdrom or local c)drom: —>
   ```

   Key in **n** to use the network.
   Key in **f** to download from floppy disk.
   Key in **t** to download from tape.
   Key in **r** to download from remote CDROM.
   Key in **c** to download from CDROM.

   You can key in the first letter of each installation option or the entire word. For example, you can key in **n** or **network** to download from a node connected through the network. **makenode**(1M) accepts upper- or lower-case letters.

5. If you are downloading through the network, respond to the following prompts concerning the network address and the *username.password* combination for the node you are connecting to. Then, proceed to step 7.

   ```
   Enter network connect string or address
                        (08-00-36-XX-XX-XX) ——> 08-00-36-12-34-56
   ```

   You can enter the network address or node name at the connect string prompt.

   ```
   Enter "username.password" combination for
                        "08-00-36-XX-XX-XX" ——> john.doe
   ```

   If you do not want to display the password, key in *username* followed by a period at the prompt and press <RETURN>. The system automatically prompts for a password, and the password does not display as it is keyed in.

6. If you are downloading from tape or floppy disk, respond to the following prompt:

   ```
   Insert a newprod product floppy/tape and press <RETURN> ——>
   ```

7. At the following prompt, specify the directory to load to or press <RETURN> to choose the default directory:

   ```
   Enter the directory where the products are to be loaded
   [/usr/ws_s]:
   ```

8. The Makenode Available Products -- Classifications Menu appears. This menu contains product groups, or classifications. See Figure P2-1.

   To enter a classification, use the arrow keys to move the cursor to the classification you wish to enter and press the <SPACE> bar. For example, to load nucleus products (products required to run the System V operating system), move the cursor to the System Nucleus classification. To enter the desired classification, press the <SPACE> bar. The classification submenu appears. A classification submenu lists all products available in that classification. See Figure P2-2.

9. Two steps are required to load products. You must first select products that you want to download. Then, you must issue a command to load the products you selected.

   To select a product to download, use the arrow keys to move the cursor to the product and then press the <SPACE> bar. Notice that the product is now highlighted, or selected.

   To load highlighted products, issue the **u** (update) option. If you decide not to load a highlighted product, move the cursor to that product and press the <SPACE> bar. The product will be unhighlighted, or unselected, and will not be loaded when the **u** option is issued.

   The following interactive options are useful for using **makenode**(1M):

   h This option provides help on selected products. Select all products that you want information on and key in **h**. Information on those products will display.

   f This option allows you to specify the file system to load products to. By default, **makenode**(1M) loads products to **/usr**. If you would rather load products to **/usr2** or another file system that has more free space than **/usr**, issue the **f** command. **makenode**(1M) prompts you for the file system to download to.

   m This option changes menu mode. **makenode**(1M) has two menu modes: multimenu and single-screen. Multimenu mode has the classifications menu and submenus for each classification; this is the mode you are placed in when you invoke **makenode**(1M). Single-screen mode lists all products on one menu.

   a This option automatically selects all products in all classifications with the status "Needs to be updated" and clears all selected products. This option switches the interface to single-screen mode. To return to multimenu mode, issue the **m** option.

- c This option automatically clears all selected products in all classifications. All highlighted products become unhighlighted.
- / This option searches for a product. After you key in /, **makenode**(1M) prompts you for the name of the product to search for. From the classifications menu or any submenu, **makenode**(1M) will search another submenu.

10. Select all products that you want to download in deliverable format. Remember, you must download any product that other workstations/servers might want to download.
11. Key in **u** to load all selected (highlighted) products.
12. When all products have been installed, the Available Products Menu appears. Key in **q** to quit **makenode**(1M).
13. Repeat this procedure for all delivery tapes or floppies that you received.

Recall that the first step in setting up a workstation/server as a delivery source is downloading DELTOOLS through **newprod**(1M) to install the delivery node utilities. If you did not load the delivery node utilities before you invoked the **makenode**(1M) utility, you must do so now. You may have downloaded products through **makenode**(1M), but you will not be able to deliver the products to other workstations/servers until you initialize the products. (All products must be initialized on the delivery source before they can be delivered.)

If you installed the delivery node utilities before you downloaded through **makenode**(1M), the products were automatically initialized during installation, and you will not need to complete the following steps. Thus, you will not need to complete the following steps. However, if you installed the delivery node utilities after you downloaded with **makenode**(1M), you must initialize products using the **Initial** command stored in the **/usr/ip32/deltools/node** directory.

If you did not install the delivery node utilities before you loaded products, key in the following to initialize all products:

```
cd /usr/ip32/deltools/node
./Initial /usr/ws_s/*
```

Delivering Software Using newprod

Once you have performed all predelivery setup procedures described in "Setting Up the Delivery Source," you are ready to deliver Intergraph software to the workstation/server. This section describes how you can use the **newprod**(1M) utility (Intergraph's main software delivery utility) to deliver software.

If one of the following circumstances applies to you, you must complete additional steps before you download software using **newprod**(1M):

- You are making a full hardware and software upgrade from a 32032-based workstation to a CLIPPER.

  If so, refer to the *Upgrading 32032-based Workstations to CLIPPERs* guide that comes with the CLIPPER upgrade package for complete instructions. Information on delivering new software is also provided in this guide.

- You are receiving disk media error messages similar to the following:

  ```
  Disk failed: s0u0p07.1 medium error:
  read error at 2933
  ```

  If so, you must rebuild your hard disk. Follow steps in the "System Rebuild" procedure to create current Rebuild Floppy disks, back up files, verify, format, partition, and rebuild your internal workstation/server hard disk before delivering new software products.

Using newprod to Install Products

This section provides steps for using **newprod**(1M) for loading software.

newprod(1M) is Intergraph's main software delivery utility. **newprod**(1M) resides in the **/usr/ip32/deltools** directory and is linked to **/usr/bin** (so that it can be invoked from any directory).

When **newprod**(1M) is invoked, it loads the **ws_s.prods** file in **/usr/tmp**. Then, **newprod**(1M) compares the date of each product listed in **ws_s.prods** with the date of each product's **product.def**. This comparison determines each product's status on the **newprod**(1M) menu. The Status field on the **newprod**(1M) menu contains one of the following descriptions:

- Needs to be updated. The product version on the delivery medium is more recent than the version on your workstation/server.
- Workstation newer. The product version on your workstation/server is more recent than the version on the delivery medium.
- New. The product does not exist on your workstation/server.
- Up to date. The product version on the delivery medium is the same as the version on your workstation/server.

To download a product, you must first select the product and then issue a command to load all selected products. To select a product, use the arrow keys to move the cursor to the product and press the <SPACE> bar. When you attempt to select a product, **newprod**(1M) checks the **product.def** file to determine the product's size. If the workstation has enough free space to load the product, the product highlights when you press the <SPACE> bar. If the product requires more than the available space, the keyboard beeps and the product is not selected (or highlighted). The amount of free space on the file system you are downloading to and the total number of blocks required to load selected products display at the top of the menu.

The **u** (update) command loads all selected products. When the update command is issued, **newprod**(1M) loads all selected products.

When you are downloading multiple products, **newprod**(1M) loads the baseline nucleus products before it loads any supplemental products or applications to ensure that the System V operating system is installed properly. **newprod**(1M) loads baseline nucleus products (products required to run the System V operating system) in the following order:

System V 3.1 File System Boot Images (UNIXBOOT)
Delivery Tools (DELTOOLS)
System V 3.1 File System (SYSTEMV)
CLIPPER Graphics Libraries (ENVIRON_S)
Screen Manager (SCREENMGR)
Workstation Network Software (INC)
Gpipe Host Shared Library (GPIPE_S)
DEC VT220 Terminal Emulation (VT220)
Workstation Graphic Resources (RESOURCES)
Workstation/Server Diagnostics (DIAG)
I/Forms Runtime Package (FORMS_S)
RIS Support Package (RISCCU)

After the nucleus products have been loaded, **newprod**(1M) begins loading baseline supplemental products (products that come free-of-charge with your workstation/server but are not required to be loaded on the hard disk) and all purchased products.

For each product, **newprod**(1M) loads the **README** file and then invokes the product's install script (**install.sh**). The install script performs the following functions:

- Connects to the network (network deliveries only)
- Downloads the product from the delivery source to the **/usr/tmp/***product-number* directory
- Decompresses the product file (if it is compressed format) and extracts it
- Saves user-specific and user-modified files in the current product directory
- Sets default protections on the files
- Copies startup files
- Adds the product to the workstation pull-down menu, if applicable
- Fulfills other miscellaneous installation requirements

After the installation script completes, **newprod**(1M) resumes control. Now, **newprod**(1M) moves the product from **/usr/tmp/***product_number* to the product directory, which is usually **/usr/ip32/***product_name*.

Follow these steps to install products from network, floppy disk, or tape:

NOTE This section assumes that you have completed all delivery steps necessary to set up the delivery source (as described in "Setting Up a Delivery Source").

1. Power up the workstation/server.
2. Log in and access the super-user account.
3. To invoke the **newprod**(1M) utility, key in the following:

   newprod

4. Respond to the following prompt:

```
Enter source of installation: n)etwork, f)loppy, t)ape,
        r)emote cdrom or local c)drom: ——>
```

   Key in **n** to use the network.
   Key in **f** to download from floppy disk.
   Key in **t** to download from tape.
   Key in **c** to download from CDROM.
   Key in **r** to download from remote CDROM.

   You can key in the first letter of each installation option or the entire word. For example, you can key in **n** or **network** to use the network. **newprod**(1M) accepts upper- or lower-case letters.

5. If you are downloading from the network, respond to the following prompts concerning the network address and the *username.password* combination. Then, proceed to step 7.

```
Enter network connect string or address
                    (08-00-36-XX-XX-XX) ——> 08-00-36-12-34-56
```

   You can enter the network address or node name at the connect string prompt.

```
Enter "username.password" combination for
                        "08-00-36-XX-XX-XX" ——> john.doe
```

If you do not want to display the password, key in *username* followed by a period at the prompt and press <RETURN>. The system automatically prompts for a password, and the password does not display as it is keyed in.

6. If you are downloading from tape or floppy disk, respond to the following prompt:

```
Insert a newprod product floppy/tape and press <RETURN> ——>
```

7. The Newprod Available Products -- Classifications Menu appears. This menu contains product groups, or classifications. See Figure P2-1.

   To enter a classification, use the arrow keys to move the cursor to the classification you wish to enter and press the <SPACE> bar. For example, to load nucleus products (products required to run the System V operating system), move the cursor to the System Nucleus classification. To enter the desired classification, press the <SPACE> bar. The classification submenu appears. A classification submenu lists all products available in that classification. See Figure P2-2.

8. Two steps are required to load products. You must first select products that you want to download. Then, you must issue a command to load the products that have been selected.

   To select a product to download, use the arrow keys to move the cursor to the product and then press the <SPACE> bar. Notice that the product is now highlighted, or selected.

   To load highlighted products, issue the **u** (update) option. If you decide not to load a highlighted product, move the cursor to that product and press the <SPACE> bar. The product will be unhighlighted, or unselected, and will not be loaded when the **u** option is issued.

   The following interactive options are useful for using **newprod**(1M):

h This option provides help on selected products. Select all products that you want information on and key in **h**. Information on those products will display.

f This option allows you to specify the file system to load products to. By default, **newprod**(1M) loads products to **/usr**. If you would rather load products to **/usr2** or another file system that has more free space than **/usr**, issue the **f** command. **newprod**(1M) prompts for the file system to download to.

m This option changes menu mode. **newprod**(1M) has two menu modes: multimenu and single-screen. multimenu mode has the classifications menu and submenus for each classification; this is the mode you are placed in when you invoke **newprod**(1M). Single-screen mode lists all products on one menu.

a This option automatically selects all products in all classifications with the status "Needs to be updated" and clears all selected products. This option switches the interface to single-screen mode. To return to multimenu mode, issue the **m** option.

c This option automatically clears all selected products in all classifications. All highlighted products become unhighlighted.

/ This option searches for a product. After you key in /, **newprod**(1M) prompts for the name of the product to search for. From the classifications menu or any submenu, **newprod**(1M) will search another submenu.

9. Select all baseline (free) products that you want to download. The following baseline products must be installed before you can run the System V operating system:

   System V 3.1 File System Boot Images (UNIXBOOT)
   Delivery Tools (DELTOOLS)
   System V 3.1 File System (SYSTEMV)
   CLIPPER Graphics Libraries (ENVIRON_S)
   Screen Manager (SCREENMGR)
   Workstation Network Software (INC)
   Gpipe Host Shared Library (GPIPE_S)
   DEC VT220 Terminal Emulation (VT220)

Workstation Graphic Resources (RESOURCES)
Workstation/Server Diagnostics (DIAG)
I/Forms Runtime Package (FORMS_S)
RIS Support Package (RISCCU)

10. Key in **u** to load all selected (highlighted) products.
11. After the baseline products have been successfully installed, select all other products that you wish to download.
12. Key in **u** to load all selected products.
13. After all products have been installed, key in **q** to quit **newprod**(1M).

Understanding and Using the Menu

Previous sections provided brief procedures for using **newprod**(1M) and **makenode**(1M). This section provides detailed explanations of **newprod**(1M)'s and **makenode**(1M)'s menu features.

Product Status

When you invoke the delivery utility (**newprod**(1M) or **makenode**(1M)), one of four descriptions will display in the Status field for each product in the menu. These four descriptions include the following:

- Needs to be updated
- Workstation newer
- New
- Up to date

A product's status describes how the software on your workstation/server compares with the software on the delivery source. The delivery utility compares the date of the product on the menu with the date of the same product on your workstation/server (in **/usr/ip32** for nondeliverable products loaded through **newprod**(1M) and in **/usr/ws_s** for deliverable products loaded through **makenode**(1M)). Products are listed alphabetically by product name within each status group.

In addition, the following statuses may appear after you have loaded products:

- Downloaded okay (**newprod**(1M) only)
- Made deliverable (**makenode**(1M) only)
- Installation errors
- Load errors

For example, if you download a product through **newprod**(1M), its status will change to "Downloaded okay." If you wish to clear this status so the status that describes whether the product is up-to-date displays, key in **a** (autoselect). The **a** option automatically selects all products with the "Needs to be updated" status and clears all temporary statuses.

Each status is described in the following sections.

Needs to be updated

All products with this status appear at the top of the menu. The "Needs to be updated" status indicates that the date of the product on your workstation/server is older than the date of the same product on the delivery source (VAX, workstation/server, CDROM, tape, or floppy disk). Thus, this product is outdated on your workstation/server. You need to load the product's newer version that is available on the delivery source.

Workstation newer

Products with this status appear second in the menu under products that need to be updated.

The "Workstation newer" status indicates one of the following circumstances:

- The product was previously downloaded from one delivery source and is currently being compared to an older version on another delivery source. The product version on the workstation/server is more recent than the version on the delivery source.

- The product was not initialized on the delivery node (VAX or workstation/server).

In the first case (downloading from multiple delivery nodes), you can download a product with the "Workstation newer" status. (You will be loading an older version of that product.) However, first check the **product.def**, and **README** files to ensure that no compatibility problems between products exist.

In the second case (the product was not initialized), you cannot download the product. The system manager must initialize the product on the delivery node before that product can be downloaded.

Use one of the following methods to initialize products on a VAX delivery source.

- Initialize all products with the following commands:

```
set def pro_dd_deltools
@ws_initial
```

- Initialize individual products by issuing the following commands. (In this example, the MicroStation product is initialized.)

```
set def pro_dd_deltools
@ws_initial ZFA3: [ws_mstation]
```

Use one of the folowing methods to initialize products on a workstation/server delivery node:

- Initialize all products with the following command:

```
/usr/ip32/deltools/node/Initial /usr/ws_s/*
```

- Initialize selected products with the following command. (In this example, the MicroStation product is initialized.)

```
/usr/ip32/deltools/node/Initial /usr/ws_s/mstation
```

Intergraph initializes all products on delivery CDROM, tape, or floppy disk before shipping. Thus, you do not need to initialize products on these media types.

New

Products with this status appear in the menu after any products with the "Workstation newer" status. The "New" status indicates that this product is available to download but does not exist on the workstation/server. To load products with this status, you may select them and load them on your workstation/server.

Up to date

Products with this status appear at the bottom of the menu. The "Up to date" status indicates that the fixes file date of the product on your workstation/server is the same as the fixes file date of the same product on the delivery source (VAX, workstation/server, CDROM, tape, or floppy disk). Thus, the same product version exists on your workstation/server and the delivery source. Products with this status are already up-to-date and do not need to be downloaded. However, you may download a product with this status (such as if you wish to respond to installation prompts differently than when you previously installed the product).

Downloaded okay

If a product is loaded through **newprod**(1M) successfully, the product's status becomes "Downloaded okay." When the Available Products Menu appears, the product will be listed in the location on the menu where it was before installation. Then, the next time you invoke the delivery utility, the product will display the "Up to date" status and will be listed at the bottom of the menu with the other up-to-date products.

Installation errors and Load errors

If a product is not loaded successfully on the hard disk, the product's status will become "Installation errors" or "Load errors." After the product's install script has completed, the message "Unsuccessful installation" appears. If baseline nucleus products are not installed properly, the message "WARNING: Please DO NOT reboot until this product is loaded successfully" displays. When the Available Products Menu reappears, the product will be located where it was before you attempted to install it. Then, the next time you invoke the delivery utility, the product will display its status before you attempted to load it. You must attempt to load the product again before you can use it because it was not copied to your hard disk.

Available Disk Space

The amount of free disk space (in the base directory) available for loading products displays at the top of all **newprod**(1M) or **makenode**(1M) menus. In addition, the amount of space required by selected products also displays at the top of the screen. When you select a product, the number of blocks in the "blocks selected" field increases by the size of the product. As you select products, you can compare the amount of free space to the number of blocks selected.

If you attempt to select a product that requires more than the space available on your workstation/server, the keyboard beeps and the product remains unhighlighted.

When you select products with the "Needs to be updated" status, the number of blocks selected does not increase because the product already resides on the disk. **newprod**(1M)/**makenode**(1M) expects the new product version to overwrite the old product.

Problems arise when new product versions require more space than old product versions. Thus, you should allow approximately 10,000 extra free blocks when loading up-to-date products to ensure that you do not run out of disk space.

If you run out of disk space while loading products, the console window displays disk error messages. At that point, you should delete the console window and use the <CONTROL>-C <CONTROL>-C key sequence to kill the **newprod**(1M)/**makenode**(1M) process.

The following are suggestions for freeing disk space. You are not required to perform all or any of them. However, each step will help free disk space.

1. Product install scripts place products in **/usr/tmp** before they are moved to the appropriate product directory. This directory's contents are removed when you reboot. Thus, you can regain disk space by rebooting or removing the contents. To remove the contents of the **/usr/tmp** directory, key in the following command:

   ```
   rm -r /usr/tmp/*
   ```

2. Core files can be created through several possible occurrences ranging from machine checks to graphics aborts. Use the **find**(1) command to locate all core files on the disk. Any core files found will be printed on the screen as they are located. An example of possible output follows:

   ```
   find / -name core -print
   /usr/ip32/vt200/core
   /usr/john/graphics/core
   ```

   Delete the core files using the remove (**rm**(1)) command as follows:

   ```
   rm /usr/ip32/vt200/core
   rm /usr/john/graphics/core
   ```

3. The file system checker, **fsck**(1M), which checks the file system's connectivity and integrity, places files in the **lost+found** directory. **fsck**(1M) runs automatically at boot time when the system was not shut down properly. This program finds and stores files not linked orderly to the file system. Remove all files in the **lost+found** directory by keying in the following:

   ```
   rm -r /usr/lost+found/*
   ```

4. The following utilities are delivered with the SYSTEMV product. If you do not use any of these utilities (**assist**(1), **help**(1), and **graf**), you can free approximately 8,000 blocks by removing them. Remove these utilities by keying in the following commands:

   ```
   rm -r /usr/lib/assist
   rm -r /usr/lib/help
   rm -r /usr/bin/graf
   ```

5. The disk free command displays free space on root and all mounted **usr** file systems. Key in the **df**(1M) (disk free) command to see how much free space is available on the file system to which you are trying to load products as follows:

   ```
   df
   ```

   If you think you have enough free space to load the products you need, or if you can load some products on an alternate file system (besides **/usr**), invoke **newprod**(1M)/**makenode**(1M) and try again. To load products on an alternate file system (such as **/usr2**), use the **f** option in the **newprod**(1M)/**makenode**(1M) menu to specify the system to download to.

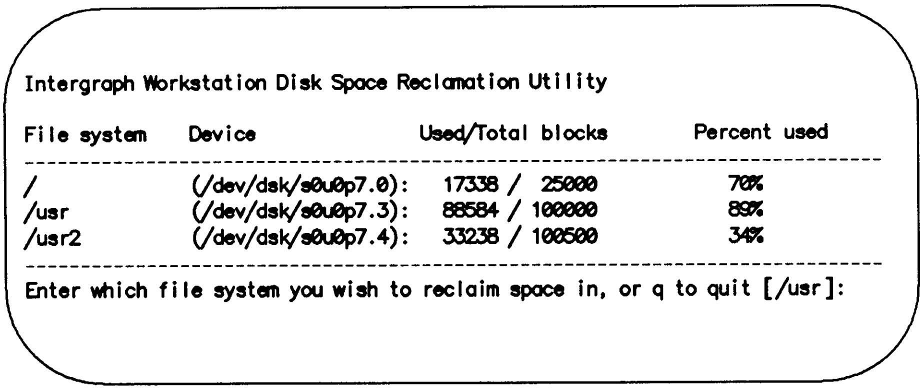
If you think you have enough free space to load the products you need, or if you can load some products on an alternate file system (besides **/usr**), invoke **newprod/makenode** and try again. To load products on an alternate file system (such as **/usr2**), use the **f** option in the newprod/makenode menu to specify the system you will download to.

6. If you still do not have enough free space for downloading products, remove any personal files and products that you do not need.

   To remove products, use the remove shell script as follows:

   /usr/ip32/deltools/remove

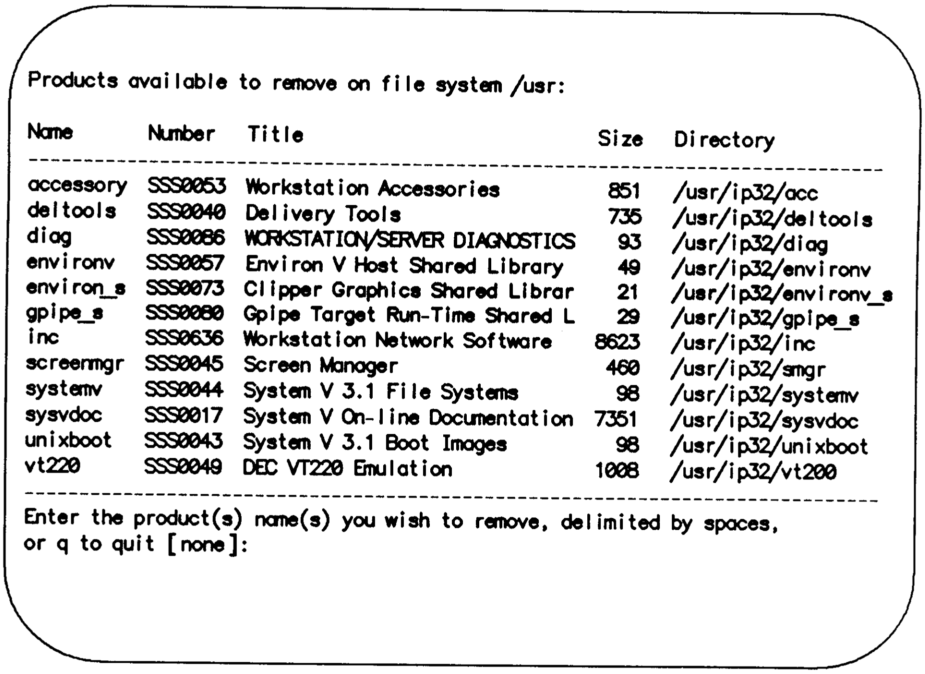
   First, the remove shell script prompts you for the file system to reclaim space in as shown in the following illustrations:



```
Intergraph Workstation Disk Space Reclamation Utility

File system   Device               Used/Total blocks      Percent used
-------------------------------------------------------------------------
/             (/dev/dsk/s0u0p7.0):   17338 /  25000          70%
/usr          (/dev/dsk/s0u0p7.3):   88584 / 100000          89%
/usr2         (/dev/dsk/s0u0p7.4):   33238 / 100500          34%
-------------------------------------------------------------------------
Enter which file system you wish to reclaim space in, or q to quit [/usr]:
```

Figure P2-3: Select File System to Remove Products From

Then, the remove shell script lists products that can be removed. Specify any products you wish to remove.



```
Products available to remove on file system /usr:

Name        Number   Title                            Size  Directory
-----------------------------------------------------------------------------
accessory   SSS0053  Workstation Accessories           851  /usr/ip32/acc
deltools    SSS0040  Delivery Tools                    735  /usr/ip32/deltools
diag        SSS0086  WORKSTATION/SERVER DIAGNOSTICS     93  /usr/ip32/diag
environv    SSS0057  Environ V Host Shared Library      49  /usr/ip32/environv
environ_s   SSS0073  Clipper Graphics Shared Librar     21  /usr/ip32/environv_s
gpipe_s     SSS0080  Gpipe Target Run-Time Shared L     29  /usr/ip32/gpipe_s
inc         SSS0636  Workstation Network Software     8623  /usr/ip32/inc
screenmgr   SSS0045  Screen Manager                    460  /usr/ip32/smgr
systemv     SSS0044  System V 3.1 File Systems          98  /usr/ip32/systemv
sysvdoc     SSS0017  System V On-line Documentation   7351  /usr/ip32/sysvdoc
unixboot    SSS0043  System V 3.1 Boot Images           98  /usr/ip32/unixboot
vt220       SSS0049  DEC VT220 Emulation              1008  /usr/ip32/vt200
-----------------------------------------------------------------------------
Enter the product(s) name(s) you wish to remove, delimited by spaces,
or q to quit [none]:
```

Figure P2-4: Removing Products

Loading Products

Two steps are required to load products. First, you must select the product (or products) that you wish to load. Then, you must issue a command to load all selected products.

Selecting/Unselecting Products

To select a product, move the cursor to the product and press the <SPACE> bar. All selected products are highlighted.

To unselect a product that is currently highlighted, move the cursor to that product and press the <SPACE> bar.

Keep in mind the following interactive commands that facilitate selecting and unselecting products:

- Key in **a** at any **newprod**(1M) or **makenode**(1M) menu to select all products in all classifications with the "Needs to be updated" status.
- To unselect all highlighted products in all classifications, key in **c** at any **newprod**(1M) or **makenode**(1M) menu.

Loading Selected Products

To load all selected (highlighted) products, issue the **u** (update) option from the **newprod**(1M)/**makenode**(1M) menu.

Moving in the Menu

This section includes information concerning moving the cursor, changing menu modes, moving between menus, and searching for products.

Moving the Cursor

newprod(1M) and **makenode**(1M) accept arrow keys, vi keystrokes, and emacs keystrokes for moving the cursor up and down the product list. The following chart shows these three different methods of moving the cursor up and down a list of products.

Action	**Arrow Keys**	**vi Keystroke**	**emacs Keystroke**
Up a line	Up arrow	k	<Ctrl-P>
Down a line	Down arrow	j	<Ctrl-N>
Back a page	Left arrow	<Ctrl-U>	<Ctrl-Z> or <Esc> V
Forward a page	Right arrow	<Ctrl-D>	<Ctrl-V>
Top of list		T or 1	<Esc> <
Bottom of list		B or G	<Esc> >

Changing Menu Modes

When you invoke **newprod**(1M)/**makenode**(1M), you are placed in multimenu mode. This menu mode groups products into classes for downloading. If you prefer to see all products listed in one list, you can change to single-screen mode. To move from multimenu mode to single-screen mode, key in **m** at any **newprod**(1M) or **makenode**(1M) menu. To return to multimenu mode, key in **m** again. You can use either multimenu or single-screen menu mode.

Moving Between Menus

To move from the Classifications Menu to a classification submenu, move the cursor to the submenu you wish to enter and press the <SPACE> bar. To exit a submenu and return to the Classifications Menu, press the <BACKSPACE> or <DELETE> key. See Figure P2-1.

Searching for Products

Another way to move through **newprod**(1M)/**makenode**(1M) menus is to search for a product. You can issue the **/** (search) command at any **newprod**(1M)/**makenode**(1M) menu and you will be prompted for the name of the product to search for. For example, if you are in the Classifications Menu and search for the DELTOOLS product, the search command transfers you to the classification submenu where DELTOOLS can be found (Systems Nucleus). The cursor remains on the DELTOOLS product. You can also search from one submenu to another.

Interactive Options

You can use the following options interactively at any **newprod**(1M)/**makenode**(1M) menu.

?	Access help for interactive options and moving the cursor.
/	Search for a product.
!	Escape to shell.
a	Automatically select all out-of-date products.

c	Clear all selected products.
d	Accept installation defaults automatically (**newprod**(1M) only).
e	Exit.
f	Choose file system to download to.
h	Access help for downloading individual products.
m	Change menu mode.
r	Review product fixes from the **README** file.
q	Quit.
u	Update selected products.
v	Download products in verbose mode.
x	Display install script while loading the product (**newprod**(1M) only).
y	Accept installation defaults automatically (**newprod**(1M) only).

The following items are functions that can be accomplished using these interactive options.

Help

The ? (**newprod**(1M) help) option lists and states the function of **newprod**(1M)/**makenode**(1M) options. From this help screen, if you press the <SPACE> bar, a help screen for moving the cursor displays.

The **h** (product help) option provides help on downloading selected products. Select all products that you want information on and key in **h**. Information such as product dependencies, the order for loading products, and file system dependencies for those products will display.

Product Information

The **h** (product help) option provides help for selected products. Select all products that you want information on and key in **h**. Information on those products will display. As the help information file displays, press <RETURN> to access the next page. To return to the menu without reading the rest of the file, key in **q**.

The **r** (review) option displays the fixes portion of the README file for the product at the cursor. Product fixes are changes or fixes to a product since its last release. As this file displays, press <RETURN> to continue to the next page. To return to the menu without reading the rest of the file, key in **q**. Select all products that you want information on and key in **h**. Information on those products will display.

Choose Menu Style

The **m** (menu) option toggles between multimenu and single-screen mode. When you invoke **newprod**(1M)/**makenode**(1M), you are placed in multimenu mode. If you prefer to see products listed on one screen rather than in product classifications, use this option to change menu modes.

Specify File System to Download To

The **f** (file system) option allows you to choose the file system that you will download to. Products are ordinarily downloaded to **/usr**. Activating this command causes a chart listing available file systems to appear. At the prompt, key in the file system that you will download to.

When you return to the **newprod**(1M)/**makenode**(1M) menu, notice that the chosen file system displays at the top. The free space and status fields reflect the change in file systems. If you have a product loaded on **/usr** and change to **/usr2**, this product's status will display as "New" because it does not reside in **/usr2**. You may load products in more than one file system, but doing so consumes disk space.

newprod(1M)/**makenode**(1M) will load the product on the specified file system, but will symbolically link the product to **/usr**. The symbolic link will cause the product to appear as though it resides in the **/usr** file system. If you list the product directory (in **/usr**) with the **ls -l** command, the product directory will point to the directory in another file system where the product actually resides. For example, if the **news** product was loaded in the **/usr2** file system, the long listing of the **news** product in **/usr/ip32** appears as follows:

```
lrwxrwx  1 root   sys    15 Mat 13 15:14 news -> /usr2/ip32/news
```

The **l** indicates the symbolic link and the arrow -> indicates the directory to which the file is linked.

Execute Shell Commands Inside newprod/makenode

The **!** (escape to shell) option allows you to execute a shell command in **newprod**(1M)/**makenode**(1M). When you activate this command, the product menu clears and the **command line?** prompt appears. At this prompt, you may execute shell commands such as **ls**(1) or **df**(1M). You are automatically placed in the **/usr/tmp** directory when you escape to the shell. To return to the **newprod**(1M) Available Products Menu, press <RETURN> after the prompt.

Search for a Product

The / (search) option searches for the product name specified. When you invoke this command, you are prompted for the name of the product to search for. **newprod**(1M)/**makenode**(1M) searches all product classifications until it finds the product specified. The cursor remains on the specified product.

Select/Unselect Products to be Loaded

To load, preview, or read a help screen for a product, you must select that product. To select a product, move the cursor to the product and press the <SPACE> bar. This action causes the product to highlight. Pressing the <SPACE> bar again causes the product to be unhighlighted, or unselected.

The **a** (autoselect) option selects all products (in all classifications) with the "Needs to be updated" status and clears all previously selected products. If you are in multimenu mode when you issue the **a** option, **newprod**(1M)/**makenode**(1M) switches to single-screen mode and highlights out-of-date products. To return to multimenu mode, key in **m**. Selected products will remain highlighted when you change menu modes.

The **c** (clear) option clears (or unselects) all products (in all classifications).

Installation Defaults Accepted Automatically

The **d** or **y** (installation defaults) option toggles between accepting all installation defaults and prompting for information. The default for this command is off (prompting for information). D appears at the top of the Available Products Menu when you activate this mode. When you invoke this command and download products, you will not be prompted with any questions. Instead, all installation defaults will be accepted. Use this option only if you are sure that the installation defaults are acceptable for your system. For information on responding to product installation prompts, use the **newprod**(1M)/**makenode**(1M) **h** option (help on loading products).

NOTE The y option does not respond to all installation defaults with a "yes." It accepts the default (whether it is "yes" or "no").

Additional Information Displayed During Installation

The **v** (verbose) option toggles the verbose mode between on and off; the default is off. Activating this mode causes a "V" to appear at the top of the Available Products Menu. When you download products in verbose mode, messages explaining the steps to install products appear on the screen. If you are having problems installing a product, use this option (along with the install script debug option) to determine the problem source.

The **x** (install script debug) option toggles the install script debug mode between on and off; the default is off. "X" appears at the top of the Available Products Menu when you activate this mode. When you invoke this command and download products, the installation script for that product scrolls down the screen as products are installed. If you are having problems installing a product, use this option (along with the verbose option) to determine the problem source.

Load Products

The **u** (update) option loads all selected products to the hard disk.

Exit

The **e** (exit) or **q** (quit) option exits **newprod**(1M) or **makenode**(1M).

Command-Line Options

The syntax for command-line mode options is as follows:

NOTE makenode(1M) can be substituted for newprod(1M) in all examples in this section except for the **x, y,** and **d** option examples.

newprod [**-tfcplvxydmRV**] [**-b** *basedir*] [**-nr** *connstr*] [**-F** *prodlist*]
[**-T** *tapedev*] [*selection* ...]

Valid **newprod**(1M) arguments are listed and described below.

Option	**Purpose/Example**
-n	(Network) Allows you to bypass prompts by specifying the node address, user name, and password on the command line (**newprod**(1M) and **makenode**(1M)). **newprod -n 08-00-36-***XX-XX-XX.username.password*
-r	(Remote CDROM) Allows you to bypass prompts by specifying the node address, user name, and password on the command line (**newprod**(1M) and **makenode**(1M)). **newprod -r 08-00-36-***XX-XX-XX.username.password*
-t	(Tape) Allows you to bypass prompts by specifying tape as the installation source on the command line (**newprod**(1M) and **makenode**(1M)). **newprod -t**

-c (CDROM) Allows you to bypass prompts by specifying CDROM as the installation source on the command line (**newprod**(1M) and **makenode**(1M)).

```
newprod -c
```

-f (Floppy) Allows you to bypass prompts by specifying floppy disk as the installation source on the command line (**newprod**(1M) and **makenode**(1M)).

```
newprod -f
```

-b (Base directory) Allows you to specify an alternate file system to install products on (**newprod**(1M) and **makenode**(1M)).

```
newprod -b /usr2
```

-F (Configuration file) Specifies the product configuration file. For example, if you have set up a product list file other than **ws_s.prods**, you must use this option to specify this alternate configuration file. If you do not use the **-F** option, **newprod**(1M) defaults to **ws_s.prods** (**newprod**(1M) and **makenode**(1M)).

```
newprod -F plotting.prods
```

-l (Long) Displays additional information such as the host directory and workstation/server directory in the Available Products Menu (**newprod**(1M) and **makenode**(1M)).

```
newprod -l
```

-p (Page) Displays all products on one long menu. To load products, you must key in the product numbers at the end of the menu (**newprod**(1M) and **makenode**(1M)).

```
newprod -p
```

-R (Review) Allows you to review product fixes to ensure that you wish to install them (**newprod**(1M) and **makenode**(1M)).

```
newprod -R SSSS0040 SSSS0044
```

-T (Tape device) Specifies the tape device. The default is **/dev/rmt/0mn**. This device must be non-rewindable.

```
newprod -T /dev/rmt/mt6n
```

-v (Verbose) Displays additional information about the software products as they are loaded (**newprod**(1M) and **makenode**(1M)).

```
newprod -v
```

-V (Version) Displays the version number of **newprod**(1M)/**makenode**(1M) and exits (**newprod**(1M) and **makenode**(1M)).

```
newprod -V
```

-x (Install script) Displays the install script (**newprod**(1M) only).

```
newprod -x
```

-y or **-d** (Installation default) Loads products and accepts the default response to any installation prompt (**newprod**(1M) only).

```
newprod -y
newprod -d
```

You can use multiple options on the command line. The following are examples of using multiple command-line options.

The following option invokes **newprod**(1M)/**makenode**(1M) through the network, connecting to **08-00-36-12-34-00**, and specifies to load products in the **/usr2** base directory:

newprod -b /usr2 -n 08-00-36-12-34-00.*username.password*

The following option loads the **SSSS0081** product from a local CDROM drive and automatically accepts all installation defaults:

```
newprod -c -d SSSS0081
```

The following option loads the **SSSS0081** product from a remote CDROM drive and automatically accepts all installation defaults. The CDROM drive is connected to a node with the network address **08-00-36-72-90-00.**

```
newprod -r 08-00-36-72-90-00 -d SSSS0081
```

The following option displays the **README** file for the **SSSS0081** product; the file is read from the delivery node **08-00-36-12-34-00.**

```
newprod -n 08-00-36-12-34-00.username.password -R SSSS0081
```

The following option invokes **newprod**(1M)/**makenode**(1M) from a product tape and loads the product **SSSS0096** in verbose mode; as the product is installed, the install script will display.

```
newprod -t -vx SSSS0096
```

CDROM Software Delivery

This section provides step-by-step instructions for delivering software from CDROM (from a local or remote drive).

NOTE

If you deliver software from CDROM, you are not required to set up a delivery source. However, the following are reasons for which you should set up a delivery source instead of delivering software from a remote CDROM:

When you deliver software from CDROM without setting up a delivery source, you will usually leave one delivery disc in the drive continuously. However, if you need to deliver software from multiple delivery discs, you will be required to switch discs.

When you deliver software from CDROM to numerous remote nodes, access time may be slower than it would for a delivery source.

Before you deliver software from CDROM, you must have completed the following:

- Configured the CDROM drive properly, as described in the *CDROM Drive User's Guide*.
- Followed procedures given in Chapter 11, "Rebuilding and Repartitioning the Hard Disk," of the *CLIPPER System Administrator's Guide* if you have just rebuilt the hard disk. The procedures in section 11.12.1 are especially important for preparing the hard disk for CDROM software delivery.
- Read and understood the instructions for inserting the CDROM in the CDROM drive, as described in the *CDROM Drive User's Guide*.
- Read and understood the care and handling instructions on the back of the CDROM case.
- Read and understood the terms explained in the following "CDROM Terminology" section.

CDROM Terminology

The following terms are used throughout this chapter. Refer to these terms when necessary during software delivery procedures.

Baseline software	You received baseline software products free of charge when you purchased an Intergraph workstation or server. There are two categories of baseline products: nucleus and supplemental. Baseline nucleus products must reside on the workstation or server before you can run the System V operating system. Baseline supplemental products are free, but do not need to reside on the workstation or server.
Caddy	An empty CDROM caddy is included with the CDROM drive. This caddy is approximately the same size as the jewel case. You must insert the CDROM in the caddy and insert the caddy in the CDROM drive to be able to access any data.
CDROM	The CDROM (Compact Disc Read Only Memory) media stores approximately 600 MB of data. As a result, you will usually load Intergraph software from only one delivery CDROM rather than from numerous tapes or floppy disks. Remember, this CDROM is read-only; you cannot overwrite the data on the CDROM.
CDROM drive	A CDROM drive allows you to access information from a CDROM but not write information to it. The CDROM drive should be configured for SCSI ID 4. Refer to the documentation that came with the drive for additional information concerning the drive.
CDROM node	The CDROM node is the workstation or server to which the CDROM drive is connected.
Delivery source	A delivery source is a workstation or server that stores products in deliverable format. Other workstation/servers download products from the delivery source across the network through **newprod**(1M). To create a delivery source, download

	delivery node utilities and then deliver products using **makenode**(1M). You are not required to create a delivery source to load software from CDROM.
Jewel case	The jewel case is the clear plastic case in which the CDROM was delivered. Any time you are not using the CDROM, store it in its jewel case.
Load key	A load key is a unique alphanumeric character sequence that enables you to copy the purchased product from the CDROM to the workstation/server hard disk.
Load-key memo	You received a load-key memo with your CDROM delivery. This memo lists the network address of the workstation or server to which your CDROM drive is connected. It also lists your unique load keys that enable you to download software.
makenode	**makenode**(1M) loads software products on a workstation/server so that it can function as a delivery source. This process enables other workstations and servers to download software from the delivery source through a network using **newprod**(1M). Products loaded with **makenode**(1M) are not installed; they serve only as a delivery source for other workstations/servers. To install products so they can be executed locally, use **newprod**(1M).
Network address	Each workstation/server has a unique identification number known as its network address. To display this number, key in **netaddr** at the CLIX shell ($ or #) prompt.
newprod	**newprod**(1M) installs software products on your workstation/server so that they can be executed locally.

Delivering Software

newprod(1M) now has the capability to load software on a workstation/server from a remote CDROM drive, thus eliminating the need to create a delivery source. By not creating a delivery source for storing deliverable products, you will save disk space. The products (in a deliverable format) reside only on the CDROM media.

Notice that (regardless of your hardware configuration) you will first deliver software to the workstation/server that the CDROM drive is connected to. Then, if you must also deliver software to remote nodes, you will move to each remote node and deliver software from the CDROM drive.

The remainder of this section provides step-by-step procedures for delivering software from CDROM without creating a delivery source.

NOTE One disadvantage to delivering software from a remote CDROM drive is that installation tends to be slower when numerous nodes are accessing the CDROM drive at once.

Follow these steps to deliver software to the workstation/server that the CDROM drive is connected to and then (if necessary) to deliver software to remote workstations and servers. For additional information on using **newprod**(1M), refer to "Delivering Software Using newprod."

NOTE Steps 1 through 14 must be performed only on the node that the CDROM drive is connected to.

1. Power up the workstation/server that the CDROM drive is connected to.
2. Log in and access the super-user account.
3. Verify that the network address of the host matches the network address on the load-key memo you received with the delivery CDROM. Locate this memo and keep it handy during the software delivery process. This memo lists all available products and their associated load keys. Each memo is specific to one

workstation/server and to the delivery date. You will receive a new load-key memo with each software delivery. If DECnet is loaded on the node, verify the correct network address from the utility pages. If the node is not running DECnet, issue the following command to ensure that the host's network address corresponds to the network address on your load-key memo:

netaddr

If the network address that appears (with the format, 08-00-36-*XX*-*XX*-*XX*) is not the network address specified on the load-key memo, you are logged in to the wrong machine for the load keys on the memo. Find the machine whose network address corresponds to the network address on the memo.

4. Insert the CDROM in the caddy and the caddy in the CDROM drive.

5. Enter the CDROM Menu by keying in the **runcd**(1M) command as follows:

   runcd

NOTE

If you receive a message saying that runcd(1M) is not found on your hard disk, refer to section 1.5 for instructions on copying runcd(1M) from the CDROM to your workstation/server.

6. If the CDROM drive has not been configured, the following message will appear:

```
Your CDROM drive has not been configured.
The device file (/dev/dsk/cdrom) has not been defined.
Please refer to the CDROM Drive User's Guide for proper
configuration procedures.
```

NOTE If you did not receive this error message, continue to step 7.

You must configure your CDROM drive before you can load software. Check the DIP switches on the rear of the CDROM drive to ensure that the drive is configured for SCSI ID 4. If it is, key in the following commands to create and link the device file:

```
/etc/mknod /dev/dsk/s4u0p0.0 b 72 0
ln /dev/dsk/s4u0p0.0 /dev/dsk/cdrom
```

If the CDROM drive is not configured for SCSI ID 4, refer to the documentation that came with the CDROM drive.

7. The CDROM Menu appears on the screen. This menu provides the following options:

```
1.  View CDROM Software Delivery Documentation
2.  Invoke the "newprod" utility for installing software
3.  Invoke the "makenode" utility to create a delivery node
4.  Set up System V Online Documentation and Intergraph Online News
5.  Exit
```

   Key in **2** to invoke **newprod**(1M).

8. The Newprod Available Products -- Classifications Menu appears. This menu lists product classification submenus. Classification submenus group products in general categories. See Figure P2-1.

   To enter a classification, use the arrow keys to move the cursor to the classification you wish to enter and press the <SPACE> bar. For example, to load nucleus products (products required to run the System V operating system), move the cursor to the System Nucleus classification. To enter the desired classification, press the <SPACE> bar. The classification submenu appears. A classification submenu lists all products available in that classification. See Figure P2-2.

   Many products are listed in each classification submenu. Products stored on the CDROM include baseline (free) products, Intergraph-developed applications, and selected third-party applications.

Each baseline product has an asterisk to the left of its product status. You can load baseline products without entering a load key.

All other products require a load key before they can be installed. Refer to your load key memo for a list of all products that you have purchased and thus can install. If you wish to purchase additional products, contact your Intergraph Sales Representative.

9. Two steps are required to load products. You must first select products that you want to download. Then, you must issue a command to load the products that have been selected.

   To select a product to download, use the arrow keys to move the cursor to the product and then press the <SPACE> bar. Notice that the product is now highlighted (selected).

   At this time, you must load the following products so that the software on the CDROM node will allow you to remotely load products from the CDROM. These products are all baseline (free) products; thus, you will not be required to enter a load key to install them.

   System V 3.1 File System Boot Images (UNIXBOOT)
   Delivery Tools (DELTOOLS)
   System V 3.1 File System (SYSTEMV)
   CLIPPER Graphics Libraries (ENVIRON_S)
   Screen Manager (SCREENMGR)
   Workstation Network Software (INC)
   Gpipe Host Shared Library (GPIPE_S)
   DEC VT220 Terminal Emulation (VT220)
   Workstation Graphic Resources (RESOURCES)
   Workstation/Server Diagnostics (DIAG)
   I/Forms Runtime Package (FORMS_S)
   RIS Support Package (RISCCU)

   Select these baseline products and any products that you have purchased and wish to load. (You do not need to select any products with the "Up to date" status because these products are not outdated.)

10. Key in **u** to update all selected (highlighted) products.

11. The following prompt will appear for each purchased product that you have not previously loaded from the delivery disc:

    ```
    Enter the load key for product-name (product-#):
    ```

    Refer to your load-key memo for each product's load key. This memo lists load keys as follows:

    product-#   *product-name*   *load-key*

NOTE

Each load key is unique to the product, machine, and delivery date of the CDROM. You must use the correct memo for the machine and delivery date.

Once you have entered the load key for a product, **newprod**(1M) stores the key in the **/usr/ip32/deltools/keys** directory, under the file name of the delivery date (such as 083189). Then, the next time you attempt to load that product (for example, on a remote node), you will not be required to enter the load key.

12. After all products have been successfully installed, press <RETURN> to return to the **newprod**(1M) menu. Key in **q** to quit **newprod**(1M). If you invoked **newprod**(1M) from the CDROM Menu, key in **5** to exit this menu.
13. Reboot the system to activate the new System V software.
14. If you have any remote nodes that you will deliver software to, continue to step 15. Otherwise, you have completed the software delivery process.

NOTE

The remaining steps must be performed for each remote node.

15. Move to a remote workstation or server that needs its software updated.
16. Log in and access the super-user account.
17. Invoke **newprod**(1M) using the remote CDROM option (**-r**) as follows to connect to the CDROM node:

    newprod -r *nodename.username.password*

NOTE *Nodename*, *username*, and *password* are for the machine that the CDROM drive is connected to.

This method of invoking newprod(1M) allows you to load products remotely from the CDROM.

18. If you received an error message similar to the following, you must update the version of **newprod**(1M) on the workstation or server. If you did not receive this error message, proceed to step 22.

```
newprod: illegal option — r
```

19. Copy the version of **newprod**(1M) from the CDROM node to the node on which you are currently logged in by keying in the following command, where *nodename.username.password* represents the node name, user name, and password of the workstation or server that the CDROM drive is connected to:

    fmu *nodename.username.password* **receive /usr/bin/newprod**

20. Link the updated version of **newprod**(1M) to the **getfile** utility by keying in the following command:

    ln /usr/bin/newprod /usr/bin/getfile

21. Invoke **newprod**(1M) as follows, where *nodename.username.password* represents the node name, user name, and password of the machine that the CDROM drive is connected to:

    newprod -r *nodename.username.password*

22. The Newprod Available Products -- Classifications Menu appears. This menu lists product classification submenus. Classification submenus group products in general categories. See Figure P2-1.

    To enter a classification, use the arrow keys to move the cursor to the classification you wish to enter and press the <SPACE> bar. For example, to load nucleus products (products required to run the System V operating system), move the cursor to the System

Nucleus classification. To enter the desired classification, press the <SPACE> bar. The classification submenu appears. A classification submenu lists all products available in that classification. See Figure P2-2.

Many products are listed in each classification submenu. Products stored on the CDROM include baseline (free) products, Intergraph-developed applications, and selected third-party applications.

Each baseline product has an asterisk (*) to the left of its product status. You can load baseline products without entering a load key.

All other products require a load key before they can be installed. Refer to your load-key memo for a list of all products that you have purchased and thus can install. If you wish to purchase additional products, contact your Intergraph Sales Representative.

23. Two steps are required to load products. You must first select products that you want to download. Then, you must issue a command to load the products that have been selected.

    To select a product to download, use the arrow keys to move the cursor to the product and then press the <SPACE> bar. Notice that the product is now highlighted (selected).

    Select all baseline nucleus products (these products must be updated before you can update the System V operating system) and any other products that you want to download. (Refer to step 9 for a list of baseline nucleus products.) You do not need to select any products with the "Up to date" status because these products are not outdated.

24. Key in **u** to load all selected (highlighted) products. The following prompt will appear for each purchased product that you have not previously loaded from the delivery disc:

    ```
    Enter the load key for product-name (product-#):
    ```

    Refer to your load-key memo for each product's load key. This memo lists load keys as follows:

    product-number   *product-name*   *load-key*

NOTE

Each load key is unique to the product, machine, and delivery date of the CDROM. You must use the correct memo for the machine and delivery date.

Once you have entered the load key for a product, **newprod**(1M) stores the key in the **/usr/ip32/deltools/keys** directory, under the file name of the delivery date (such as 083189). Then, the next time you attempt to load that product (for example, on a remote node), you will not be required to enter the load key.

25. In some cases (such as if you just rebuilt the hard disk), you may receive one of the following error messages when you attempt to load products:

    ```
    install.sh: getfile not found
    stdin: not in compressed format
    Unsuccessful installation
    ```

    OR

    ```
    install.sh: test: argument expected
    Unsuccessful installation
    ```

    If you receive either of these messages, you probably need to link **newprod**(1M) to **getfile**. To do so, key in the following command:

    ln /usr/bin/newprod /usr/bin/getfile

26. When all products have been downloaded, quit **newprod**(1M) by keying in **q**.
27. Reboot the system to activate the new System V software.
28. Repeat steps 15 through 27 for each workstation/server in the network.

Setting Up a Delivery Source

A delivery source is a workstation or server that stores products loaded with the **makenode**(1M) utility. Other nodes can connect to the delivery source and install software.

In the past, if you needed to load software from one CDROM drive to multiple workstations, you were required to create a delivery source. Now, **newprod**(1M) allows you to load products on workstations and servers from a remote CDROM drive, thus eliminating the need for a delivery source. For instructions on delivering software without creating a delivery source, follow steps in "Delivering Software."

Creating a delivery source and delivering software are complicated processes. However, a delivery source may provide quicker installation if you will be loading software on numerous nodes simultaneously. In addition, if you do not wish to keep the delivery disc in the drive at all times, you may wish to create a delivery source.

Using the CDROM Menu

The CDROM Menu displays when you issue the **runcd**(1M) command. This menu contains the following options:

```
1.  View CDROM Software Delivery Documentation
2.  Invoke the "newprod" utility for installing software
3.  Invoke the "makenode" utility to create a delivery node
4.  Set up System V Online Documentation and Intergraph Online News
5.  Exit
```

Option 1: View CDROM Software Delivery Documentation

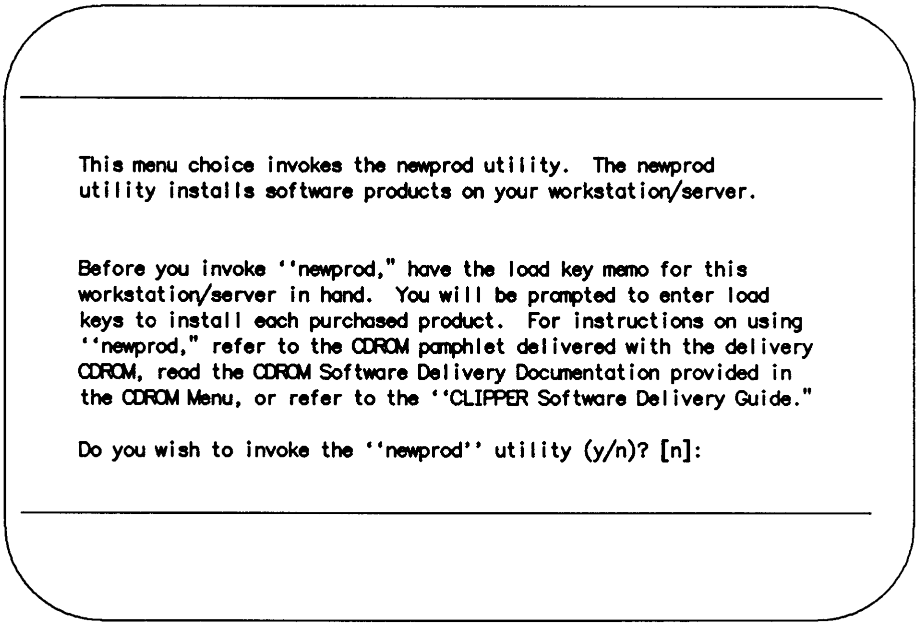
When you choose this option, the following information appears on the screen:

```
This menu choice allows you to view CDROM Software Delivery
Documentation.  This documentation includes the following topics:

-  CDROM terminology
-  Delivering software from a dedicated CDROM drive
-  Setting up a delivery source and delivering software
-  Understanding and using the newprod/makenode menu
-  Using newprod/makenode interactive options
-  Using newprod/makenode command-line options
-  Automating the load key process

You should be familiar with this information before you attempt to
install software on your workstation/server from delivery CDROM.

Do you wish to view the CDROM Software Delivery Documentation
(y/n)? [n]:
```

This documentation contains the information presented in the *CLIPPER Software Delivery Guide*. If you wish to read this online documentation, key in **y** at the prompt. If you do not, key in **n** to return to the CDROM Menu.

Option 2: Invoke the "newprod" utility for installing software

When you choose this option, the following information appears on the screen:



```
This menu choice invokes the newprod utility.  The newprod
utility installs software products on your workstation/server.


Before you invoke ''newprod," have the load key memo for this
workstation/server in hand.  You will be prompted to enter load
keys to install each purchased product.  For instructions on using
''newprod," refer to the CDROM pamphlet delivered with the delivery
CDROM, read the CDROM Software Delivery Documentation provided in
the CDROM Menu, or refer to the ''CLIPPER Software Delivery Guide."

Do you wish to invoke the ''newprod'' utility (y/n)? [n]:
```

If you wish to invoke **newprod**(1M), key in **y** at the prompt. If not, key in **n** to return to the CDROM Menu.

Option 3: Invoke the "makenode" utility to create a delivery node

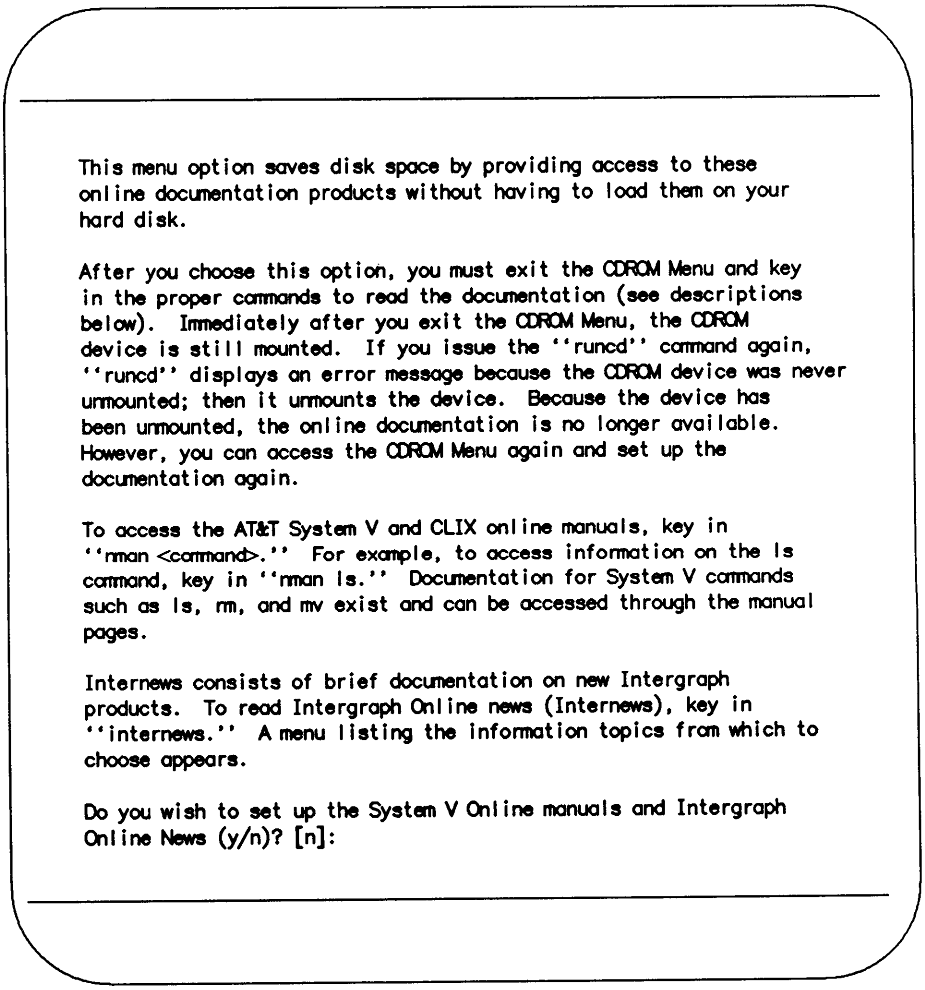
When you choose this option, the following information appears on the screen:

```
This menu choice invokes the ''makenode'' utility.  The makenode
utility installs software products (in deliverable format) on a
workstation/server that functions as a delivery source.  This
process enables other workstations/servers to download software
from the delivery node through a network.

In the past, if you were loading products from one CDROM drive to
multiple nodes, you were required to use ''makenode''to set up a
delivery source.  Now, the capability exists to load products
directly from a remote CDROM drive using ''newprod'' (without
creating a delivery source).  Thus, you are not required to use
''makenode'' in order to load products to multiple nodes.

SETTING UP A DELIVERY SOURCE IS NOT REQUIRED IN ANY SITUATION.
DO NOT USE THIS UTILITY IF YOU DO NOT WISH TO USE YOUR
WORKSTATION/SERVER AS A DELIVERY SOURCE.  INSTEAD, USE THE NEWPROD
UTILITY (CHOICE 2).

Before you invoke ''makenode,'' have the load key memo for the
workstation/server in hand.  You will be prompted to enter the load
keys to install each purchased product.  For instructions on using
''makenode,'' refer to the CDROM pamphlet delivered with the
delivery CDROM, read the CDROM Software Delivery Documentation
provided in the CDROM Menu, or refer to the ''CLIPPER Software
Delivery Guide''.

Do you wish to invoke the ''makenode'' utility (y/n)? [n]:
```

If you wish to invoke **makenode**(1M), key in **y** at the prompt. If not, key in **n** to return to the CDROM Menu.

Option 4: Set up System V Online Documentation and Intergraph Online News

When you choose this option, the following information appears on the screen:



```
This menu option saves disk space by providing access to these
online documentation products without having to load them on your
hard disk.

After you choose this option, you must exit the CDROM Menu and key
in the proper commands to read the documentation (see descriptions
below).  Immediately after you exit the CDROM Menu, the CDROM
device is still mounted.  If you issue the ''runcd'' command again,
''runcd'' displays an error message because the CDROM device was never
unmounted; then it unmounts the device.  Because the device has
been unmounted, the online documentation is no longer available.
However, you can access the CDROM Menu again and set up the
documentation again.

To access the AT&T System V and CLIX online manuals, key in
''rman <command>.''  For example, to access information on the ls
command, key in ''rman ls.''  Documentation for System V commands
such as ls, rm, and mv exist and can be accessed through the manual
pages.

Internews consists of brief documentation on new Intergraph
products.  To read Intergraph Online news (Internews), key in
''internews.''  A menu listing the information topics from which to
choose appears.

Do you wish to set up the System V Online manuals and Intergraph
Online News (y/n)? [n]:
```

If you wish to set up the online documentation so that you can access it from the delivery CDROM rather than by loading it on your hard disk, key in **y** at the prompt. If not, key in **n** to return to the CDROM Menu.

Option 5: Exit

This option exits the CDROM Menu. It also unmounts the CDROM device unless online documentation has been set up (option 4).

Restoring runcd

runcd(1M) is a shell script used to mount a CDROM and initiate the CDROM Menu. To execute **runcd**(1M), key in the following:

```
runcd
```

If you receive a message saying that **runcd**(1M) was not found on your workstation/server, follow these steps to copy **runcd**(1M) from the CDROM to your hard disk:

1. Insert the delivery CDROM in the drive.
2. Mount the correct device. If your CDROM drive is not connected to SCSI ID 4, you must create the CDROM drive device file using **mknod**(1M). Refer to the *CDROM Drive Installation Guide* for the appropriate command line. If your CDROM drive is connected to SCSI ID 4, the device file has already been created.
3. Link the device file (**/dev/dsk/s4u0p0.0**) to the CDROM drive (**/dev/dsk/cdrom**) by keying in the following command line. This example assumes that the CDROM drive is connected to SCSI ID 4. If it is not, substitute the proper device file for **/dev/dsk/s4u0p0.0**.

   ```
   ln /dev/dsk/s4u0p0.0 /dev/dsk/cdrom
   ```

4. Mount the device by keying in the following, where *X* represents the SCSI ID of the CDROM drive:

   ```
   mount -rf FFS /dev/dsk/sXu0p0.0 /del
   ```

5. Copy **runcd**(1M) by keying in the following:

   ```
   cp /del/runcd /usr/bin
   ```

6. Unmount the CDROM by keying in the following:

   ```
   umount /del
   ```

7. Execute **runcd**(1M) as follows:

   ```
   runcd
   ```

8. Continue the software delivery procedure.

3. System Reconfiguration

Procedure 3: System Reconfiguration

Overview of System Reconfiguration

Purpose	To instruct the System Administrator on reconfiguring a system
Commands	**sysconfig**(1M)
Caution	An errant configuration change can result in a non-bootable system.

This chapter describes how to reconfigure the CLIX system. Many parameters control the CLIX system environment. For example, the number of possible open files, the number of mounted file systems, and the total number of processes can be adjusted. Also, there are several drivers that may or may not be included in your system.

The **sysconfig**(1M) utility is a menu interface for configuring the CLIX system. It simplifies system configuration by categorizing all the tunable parameters and configurable drivers and providing a simple interface to modify them. **Note: sysconfig**(1M) must be purchased from Intergraph.

Once **sysconfig**(1M) has been used to configure the system as desired, the changes are saved, and new kernels are automatically created according to the specifications. Two kernels are created for each machine: one with the Network File System and Remote File Sharing support and one without any type of remote file system capability.

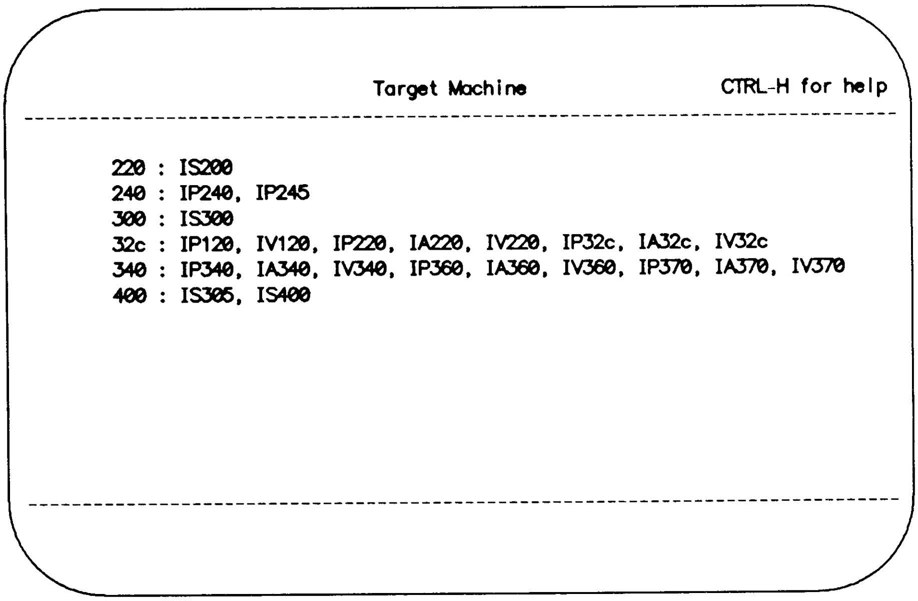
WARNING Changing a system configuration can easily result in a system that will not boot. Therefore, extreme care must be taken when making changes. Do not make a change without understanding its effects on the system.

Getting Started

Invoking Sysconfig

You must be in the correct directory to access **sysconfig**(1M). The **sysconfig**(1M) command must be invoked from the parent of the **master.d** and **build** directories. If not, **sysconfig**(1M) will print an error message and exit.

When **sysconfig**(1M) is invoked, a **Machine Menu** similar to that shown in Figure P3-1 will appear on the screen.



```
                              Target Machine                  CTRL-H for help
------------------------------------------------------------------------------

        220 : IS200
        240 : IP240, IP245
        300 : IS300
        32c : IP120, IV120, IP220, IA220, IV220, IP32c, IA32c, IV32c
        340 : IP340, IA340, IV340, IP360, IA360, IV360, IP370, IA370, IV370
        400 : IS305, IS400







------------------------------------------------------------------------------
```

Figure P3-1: Machine Menu

Cursor Movement

The cursor is always the currently highlighted or colored line of text on the menu. To move the cursor, use any of the following keys:

move down:	<DOWN-ARROW>	or	<CONTROL>-N
move up:	<UP-ARROW>	or	<CONTROL>-P
move right:	<RIGHT-ARROW>	or	<CONTROL>-F
move left:	<LEFT-ARROW>	or	<CONTROL>-B

When the cursor is on the object you wish to select, press <RETURN>.

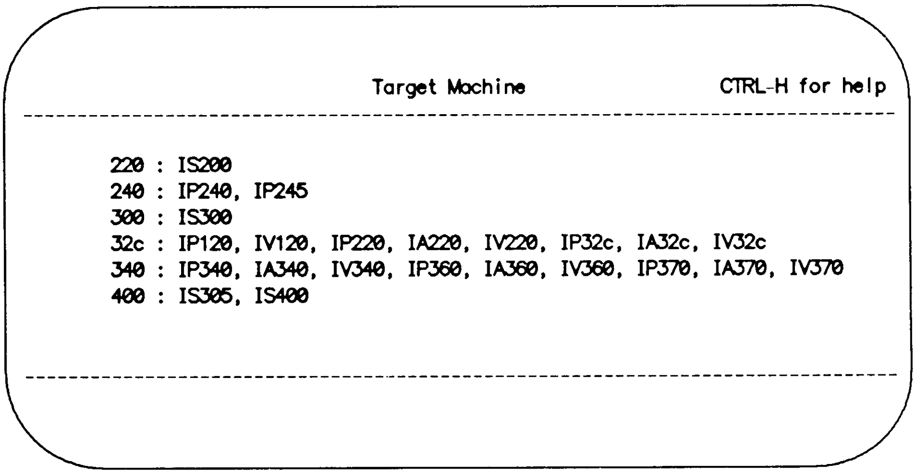
Menus

Menu Hierarchy

The **sysconfig**(1M) menus are designed to allow quick movement to the areas where changes are to be made. This is accomplished by arranging items into several levels of categories and subcategories.

The menus can be conceived as a set of file drawers. To find something in a set of files, you first choose the drawer containing the item you want. Then, within that drawer, you choose the folder that contains the item. Individual items or groups of items may be paper clipped together within the folder. If the folder has groups of items, you choose the group with the item you need. This same process of narrowing down an item's location is used in **sysconfig**(1M).

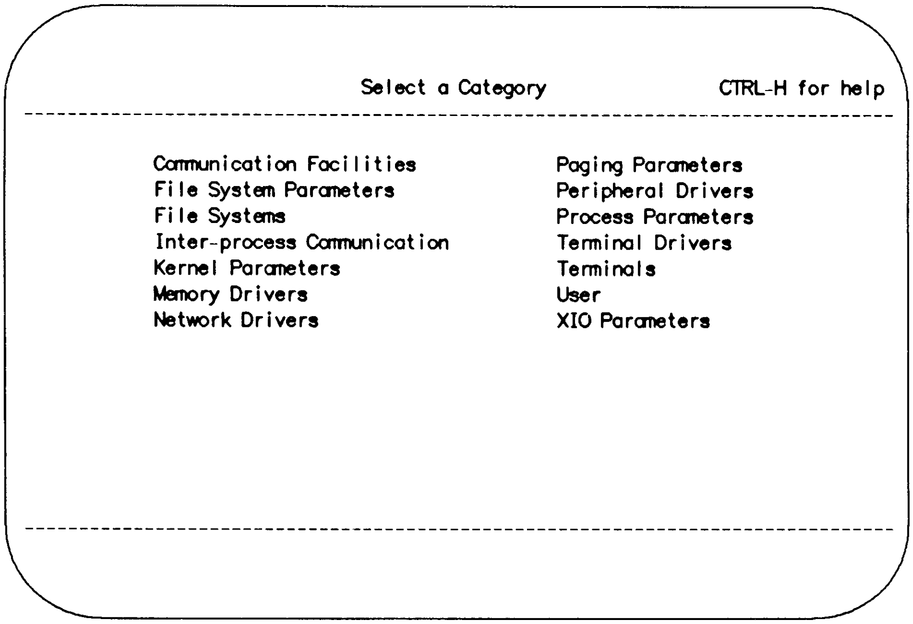
The first menu level, corresponding to the file drawers, is the **Machine Menu**. This is where you select the machine for which you will be changing items. The **Machine Menu** is similar to that shown in Figure P3-2.



```
                         Target Machine                  CTRL-H for help
--------------------------------------------------------------------------

        220 : IS200
        240 : IP240, IP245
        300 : IS300
        32c : IP120, IV120, IP220, IA220, IV220, IP32c, IA32c, IV32c
        340 : IP340, IA340, IV340, IP360, IA360, IV360, IP370, IA370, IV370
        400 : IS305, IS400


--------------------------------------------------------------------------
```

Figure P3-2: Machine Menu

The second level, corresponding to the file folders, is the **Category Menu**. The **Category Menu** is similar to that in shown in Figure P3-3.

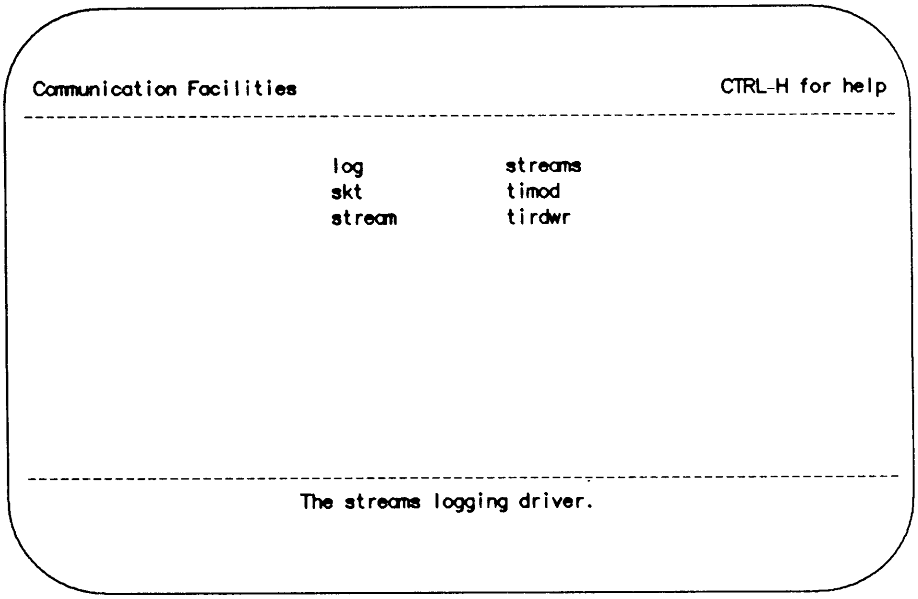


```
                    Select a Category              CTRL-H for help
-----------------------------------------------------------------------

        Communication Facilities          Paging Parameters
        File System Parameters            Peripheral Drivers
        File Systems                      Process Parameters
        Inter-process Communication       Terminal Drivers
        Kernel Parameters                 Terminals
        Memory Drivers                    User
        Network Drivers                   XIO Parameters

-----------------------------------------------------------------------
```

Figure P3-3: Category Menu

Each entry in this menu is an overall category of items that have commonality. For instance, the "File System Parameters" category contains all tunable parameters that deal directly with file systems. These include items such as the total number of files that may be open on a system (NFILE), the number of file systems that can be mounted (NMOUNT), and the number of system buffers available for caching file blocks (RNBUF).

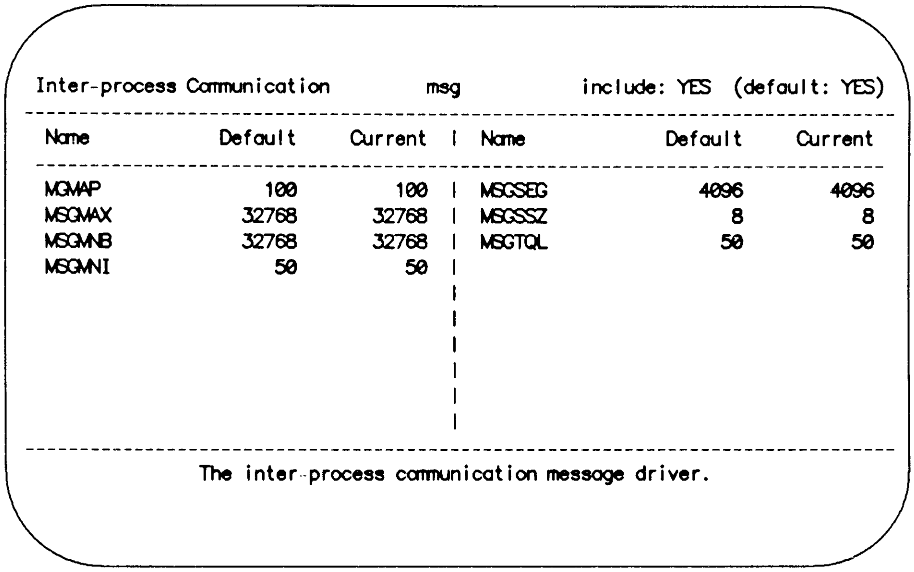
The third level is either a **Group** or a **Data Menu**, depending on the category selected at the second level. A **Group Menu** corresponds to groups of items paper clipped together and is where the group is chosen. The **Group Menu** appears similar to that shown in Figure P3-4.



```
Communication Facilities                                    CTRL-H for help
---------------------------------------------------------------------------
                              log           streams
                              skt           timod
                              stream        tirdwr

---------------------------------------------------------------------------
                         The streams logging driver.
```

Figure P3-4: Group Menu

The lowest level is a **Data Menu** and corresponds to the item groups paper clipped together within a file folder. (If a **Data Menu** is found at the third level, the items are not grouped.) The **Data Menu** is where the items are viewed and changed or where a whole group can be deleted. A **Data Menu** is similar to that shown in Figure P3-5.

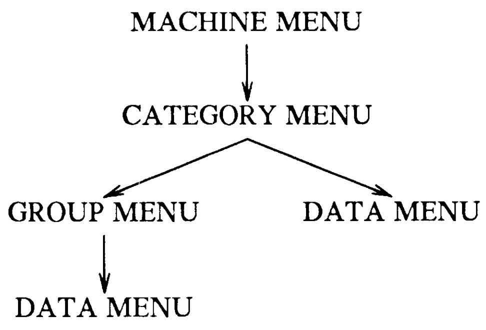


```
Inter-process Communication        msg          include: YES  (default: YES)
-----------------------------------------------------------------------------
Name                 Default     Current  |  Name               Default     Current
------------------------------------------------------------------------------
MGMAP                    100         100  |  MSGSEG                4096        4096
MSGMAX                 32768       32768  |  MSGSSZ                   8           8
MSGMNB                 32768       32768  |  MSGTQL                  50          50
MSGMNI                    50          50  |
                                          |
                                          |
                                          |
                                          |
                                          |
                                          |
------------------------------------------------------------------------------
              The inter-process communication message driver.
```

Figure P3-5: Data Menu

Hierarchy Diagram

The following diagram graphically represents the menu-level hierarchy:



Changed Item Display

Any time an item is changed so that its value is different than it was when **sysconfig**(1M) was invoked, the machine, category, and group containing the item, as well as the item itself, are displayed in bold or colored text. This allows you to quickly see the number and location of changes made.

Menu Organization

All **sysconfig**(1M) menus are organized into the following three kinds of areas or sections:

Title Line
Data Lines
Help Lines

Title Line

The *Title Line*, a single line across the top of the screen, identifies the menu you are using. Any time you are working within a category, the category name is displayed on the left side of the Title Line. If you are working on parameters within a group, the group name appears in the middle of the Title Line. If the group may be optionally included, its status is displayed to the right of the Title Line.

In addition, the Title Line indicates that you can invoke Help any time by pressing <CONTROL>-H or the <HELP> key. When you press one of them, **sysconfig**(1M) displays a Pop-up menu showing a summary of **sysconfig**(1M) commands.

Data Lines

The *Data Lines* show the objects available for selection. They are located in the middle of the display. The objects will be one of four types:

1. Machines. These are various machines that a CLIX kernel can be configured for.
2. Categories. These are the categories that the CLIX parameters and groups are in.

3. Groups. These are groups within a category that may be optional, have tunable parameters associated with them, or both.
4. Parameters. These are tunable parameters within a category or group.

Help Lines

The *Help Lines* offer helpful information for any group or parameter that is currently highlighted. In addition, any messages or prompts **sysconfig**(1M) displays are put on these lines. The Help Lines consist of either one or three lines at the bottom of the display, depending on the current menu.

Commands Common to All Menus

Pop-up Help

Help is available any time within **sysconfig**(1M) by pressing <CONTROL>-H or the <HELP> key. When you are finished using the pop-up Help, press any key to remove it.

Cursor Movement

The cursor is highlighted. To move it, use any of these keys:

move down:	<DOWN-ARROW>	or	<CONTROL>-N
move up:	<UP-ARROW>	or	<CONTROL>-P
move right:	<RIGHT-ARROW>	or	<CONTROL>-F
move left:	<LEFT-ARROW>	or	<CONTROL>-B

Menu Movement

To move up to the previous menu, press <CONTROL>-Z. If you press <CONTROL>-Z while in the top-level menu (**Machine Menu**), all changes are saved and new kernels are created for the machines that have changes.

To move down to the next menu, press <RETURN> while the cursor is on the object you want to move to. (This only works if there is a lower menu to move to.)

Session Change Removal

You can reset all items in an object to the values they had when **sysconfig**(1M) was invoked by moving the cursor over the object and pressing <CONTROL>-R.

The reset feature is useful if you made changes in several categories or several groups and decide to return some to their original values. To do this, move the cursor over the machine, category, or group to reset and press <CONTROL>-R. You are prompted on the *Help Lines* with "Remove session changes?." To reset the items, respond with a **Y** or a **y** followed by <RETURN>. No action will be taken for other responses.

The reset feature is also useful if you change an item and later decide that you did not really want to change it. Simply move the cursor over the item and press <CONTROL>-R. The item is changed back to its original value.

Default Values

You can reset all items in an object to their default values by moving the cursor over an object and pressing <CONTROL>-U. An object may be either a machine, category, group, or item. Unless the cursor is directly on an item, you are prompted on the Help Lines with a "Use default values?" question. If you are sure you want to use default values for the object, respond with a **Y** or a **y** followed by <RETURN>. No action will be taken for other responses.

Advanced Mode

Many items known as "advanced" topics are not displayed unless advanced mode is toggled on. This mode can be toggled on any time by pressing <CONTROL>-A. To toggle advanced mode off, press <CONTROL>-A again. The state of advanced mode (on or off) is displayed in the pop-up Help. The advanced mode defaults to off.

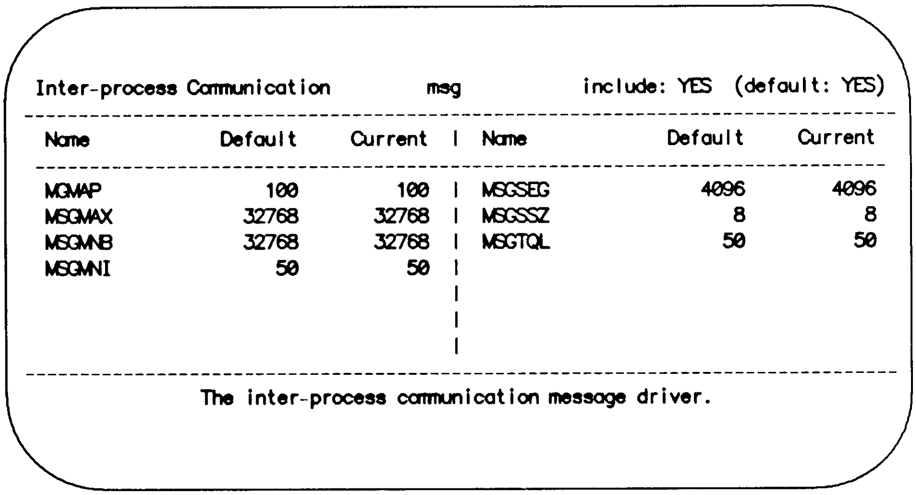
Immediate Exit

You can exit **sysconfig**(1M) any time by pressing <CONTROL>-C. All changes are lost and **sysconfig**(1M) does not ask for confirmation when you choose immediate exit.

Data Menu

Data Menu Layout

Figure P3-6 is a sample **Data Menu.**



```
Inter-process Communication        msg            include: YES  (default: YES)
------------------------------------------------------------------------------
Name              Default     Current  |  Name              Default    Current
------------------------------------------------------------------------------
MQMAP                 100         100  |  MSGSEG               4096       4096
MSGMAX              32768       32768  |  MSGSSZ                  8          8
MSGMNB              32768       32768  |  MSGTQL                 50         50
MSGMNI                 50          50  |
                                       |
                                       |
                                       |
------------------------------------------------------------------------------
                The inter-process communication message driver.
```

Figure P3-6: Data Menu

The top right section is where you can specify whether a group will be included. This section does not appear if the **Data Menu** is accessed from the **Category Menu** or if the group's inclusion is not optional.

The middle section of the window shows the tunable parameters with their default and current values. If a value contains too many characters to fit in the space allocated, it is marked with a ">" on its right. For example, the Kernel Parameter "NREGION" defaults to ((NPROC*7)/2). This is too long to be displayed completely, so it appears in the menu as follows:

```
((NPROC*>
```

To see the rest of the value, move the cursor over it and repeatedly press the ">" key. The value shifts left one character with each key press.

Eventually, it looks like this:

```
<OC*7)/2)
```

A "<" appears on the right as soon as the value shifts and the ">" on the right disappears when the rightmost character of the value is displayed. The value can be shifted left and right as much as desired. The display returns to showing the left part of the value when the cursor is moved.

Help for the currently highlighted item is displayed in the bottom section of the window.

Changes

Changing Group Inclusion

If a **Data Menu** is accessed from a **Group Menu**, group inclusion may be optional. The current inclusion state is displayed on the right side of the top line and may be changed.

In most cases, a group corresponds to a driver. Therefore, including a group also includes a driver in the system. Similarly, excluding a group also excludes a driver. For example, in Figure P3-6 the group name is "msg" and corresponds to the "Inter-Process Communication Message" driver shown in the Help line.

When the menu is first displayed, the cursor is over the first letter of the value next to "include:." This value is either "YES" if the group is to be included in the kernel, or "NO" if it is not.

Press **n** to change the value to "NO" or **y** to change it to "YES." The cursor can be moved between this area and the parameters using the arrow keys in the normal manner.

Parameter Changes

To change the value of a parameter, move the cursor over it, key in the new value, and press <RETURN>, <UP-ARROW>, or <DOWN-ARROW>. If <RETURN> or <DOWN-ARROW> is pressed, the cursor moves down to the next parameter. If <UP-ARROW> is pressed, the cursor moves up to the previous parameter.

If the parameter value is blank when input is completed (you erased everything you entered), the value will be the same as it was before.

When the first character is entered, the old value is erased and the new character followed by the cursor is displayed. Each additional character entered pushes the previous characters left, while the cursor remains in the same position. The maximum length of a value is 29 characters.

If a parameter value has too many characters to fit in the display area, a < appears to the left of the display area to denote more characters exist to the left, and a > appears to the right to denote more characters to the right. To see the hidden characters, the <RIGHT-ARROW> and <LEFT-ARROW> keys can be used to shift the value right or left, respectively.

While entering a parameter value, several editing commands similar to those used in the EMACS text editor are available. The available commands are as follows:

<CONTROL>-A	Move the cursor to the beginning of the parameter value.
<CONTROL>-B	Move the cursor one character left.
<CONTROL>-D	Delete the character under the cursor.
<CONTROL>-E	Move the cursor to the end of the value string.
<CONTROL>-F	Move the cursor one character to the right.
<CONTROL>-H	Delete the character to the left of the cursor.
<CONTROL>-K	Erase the current parameter value. The last parameter value erased can be called back with <CONTROL>-Y.
<CONTROL>-N	Save the parameter value and move the cursor down to the next parameter.
<CONTROL>-P	Save the parameter value and move the cursor up to the previous parameter.
<CONTROL>-Y	Replace the current parameter value with the last one erased by <CONTROL>-K. If <CONTROL>-K has not been used yet, replace the current parameter value with the value it had before editing.

System Creation

After the changes are complete, a new kernel can be created. To do this, move to the **Machine Menu** and press <CONTROL>-Z. You are prompted with the "Save changes and make kernels?" question. If you are certain that the changes are correct, type **Y** or **y** and a <RETURN>.

After you press a <RETURN> **sysconfig**(1M) first saves all changes and then creates two kernels for each machine that you changed, one with remote and network file sharing and one without. As each kernel is created, you see the commands issued to create it. After all kernels are created, their file locations are displayed and **sysconfig**(1M) exits.

To install a newly created kernel, move the kernel (either with or without remote file system support) to **/unix** of the target machine and reboot the target machine. The new kernel will be loaded and executed during the machine boot.

You should retain the old kernels by renaming them rather than overwriting them. If a new kernel causes problems, you can reboot the machine using the old kernel that worked correctly.

To reboot from a kernel other than **/unix** on a graphics workstation (not an InterServe™), press <CONTROL>-C immediately after choosing System V from the System Startup (blue page) Menu. You will be prompted for the kernel name at which time you may enter the name of the saved kernel and press <RETURN>.

To reboot from a kernel other than **/unix** on an InterServe, press any key after the "Hit any key in 5 seconds" prompt. Enter **unix**, press <RETURN>, and immediately press <CONTROL>-C. You will be prompted for the kernel name. At this time you may enter the name of the saved kernel and press <RETURN>.

Example

This example will demonstrate the process to install a new kernel on an InterServe 200 after changing the number of concurrent processes a nonsuper-user is allowed to run.

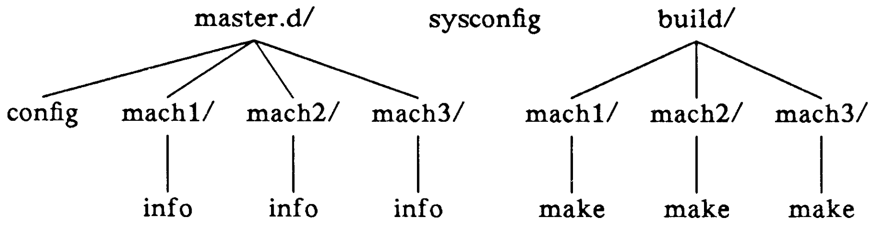
1. Move to the directory containing the **master.d** and **build** subdirectories (most likely the **/usr/src/uts/clipper** directory).

2. Execute **sysconfig**(1M).
3. Choose "200" as the target machine.
4. Choose the "Process Parameters" category.
5. Move to the parameter to be changed. The parameter is "MAXUP" in the "Process Parameters" category.
6. Change "MAXUP" to the new value by typing in the new value.
7. Move back to the **Machine Menu** and exit **sysconfig**(1M) with <CONTROL>-Z. Answer **yes** to the "Save changes and make kernels?" question.
8. **sysconfig**(1M) indicates that it is saving your change. It will then show the commands issued to make the new kernels. Finally, it indicates that the new kernels were created and stored in **build/200/unix** and **build/200/unixfs**.
9. Rename the **/unix** kernel so it is not overwritten with the new one (in case the new one does not work correctly).
10. Move either **build/200/unix** or **build/200/unixfs** to **/unix**.
11. Reboot the workstation.

The machine should now run under a new kernel with "MAXUP" set to the new value.

Appendix A: Directory Structure

The directory structure delivered with the Configurable Kernel product is as follows:



Usually, this tree will be in the **/usr/src/uts/clipper** directory. However, location is not important so long as the structure remains the same.

The **master.d** directory contains all configuration information about the various machines. An additional file, **config**, in the **master.d** directory contains data **sysconfig**(1M) uses to determine what machines can be configured and where the configuration information for each machine is located.

The **build** directory contains all libraries and support files for creating new kernels for various machines.

Appendix B: What sysconfig Is Actually Doing

The **sysconfig**(1M) utility is a smart editor that produces a kernel from a custom configuration. It follows specific rules for locating parameters that can be changed and files that can be optionally included. When you chang values using **sysconfig**(1M), **sysconfig**(1M) rewrites the new values in the location of the old ones and initiates the creation of the new kernel.

The files with the information that **sysconfig**(1M) needs are stored in the **master.d** directory under directories for each machine type. Three kinds of information are in these files: C program code and data, **sysconfig**(1M) tokens, and **mkconfig**(1M) tokens. The **mkconfig** tokens and C text describe kernel creation. The **sysconfig**(1M) tokens describe the items that may be changed in each file.

Parameters changed with **sysconfig**(1M) are actually arguments in #define lines in the **master.d** files. To find current parameter settings, it searches all files for tunable #define lines and reads the current values. To make the necessary parameter changes, **sysconfig**(1M) overwrites the old #define arguments with the new ones.

Groups that can be included or excluded are actually the names of files within the **master.d** machine directories. An excluded file will be ignored when a kernel is created. The names of included files are kept in the **master.d** machine directories in files named **LIST***machine*, where *machine* is the name of the corresponding machine. For example, the InterServe 200 file is called **LIST200**. This list is later used to determine which files need to be processed to create a kernel. When **sysconfig**(1M) is first executed, it reads the **LIST***machine* files to determine current kernel configurations.

After **sysconfig**(1M) has written all changes, it executes **make**(1) using the makefile in the **build/***machine* directory. The makefile gathers information needed to make a new kernel and invokes **mkconfig**. **mkconfig** processes the configuration files and generates the C source code needed to make the kernels. The makefile then continues with the compilation and linking of the kernels.

The newly created kernels will be in the directory **build/***machine*. The two kernels created for each machine are **unix** and **unixfs**. The difference between these two kernels is that **unixfs** contains the code needed to support both the Network File System (NFS™) and Remote File Sharing (RFS), while the **unix** kernel does not contain this code.

4. FFS Installation

Procedure 4: FFS Installation

Overview of FFS Installation

Purpose	To instruct the system administrator in installing a Fast File System to organize data on a disk for more efficient access
When Performed	When users do not need to access the file system being upgraded
Starting Conditions	Log in as super-user
Commands	**newfs**(1M), **ffsmkfs**(1M), **umount**(1M), **mount**(1M), **labelit**(1M)
Caution	If the Fast File System being created already is a standard file system, back up all files that you want to save. The procedure for creating a Fast File System deletes all files in the existing file system. Also, the existing file system must be unmounted using **umount**(1M) before **newfs**(1M) is executed.
Reference	FFS Tutorial FFS Check Tutorial

The Fast File System (FFS) is a way of organizing data stored on a magnetic disk drive to enhance the speed of disk read and write operations. All file systems on the workstation/server other than /(root) and **/usr** can be converted to FFS provided they do not contain symbolic links. The current implementation of FFS does not support symbolic links. File systems can be converted to FFS while the system is running by following this procedure.

Installing a Fast File System

WARNING The following procedure will destroy all data currently on the file systems being upgraded to FFS.

Step 1: Back up all files on the existing file system that you want to save. **cpio**(1), **tar**(1), or **backup**(1) may be used to perform the backup. The conversion process will erase any pre-existing data on the file system(s).

Step 2: Find out the Intergraph part number of the drive you are using. These numbers have the form FDSK*xxx* or MESA*xxx* and can be found on the Disk Maintenance page of the Utility pages. You need the part number for the disk corresponding to both the SCSI ID and logical unit number (LUN) where the partition being upgraded resides.

Step 3: Create the Fast File System. If you are on an Intergraph-supplied disk, use the **newfs**(1M) utility.

If you are using a non-Intergraph-supplied disk, you must use **ffsmkfs**(1M). Refer to your hardware manual for your drive and **ffsmkfs**(1M) for details on building a Fast File System.

The arguments to **newfs**(1M) are as follows:

newfs [-**v**] [-**N**] [ffsmkfs-options] *special part_number*

The components of these arguments are explained below.

v	Prints the **ffsmkfs**(1M) command executed to build the FFS and then builds the *fs*.
N	Does not create a FFS; it prints out parameters that would be used.
ffsmkfs-options	Are not recommended but can be used to override defaults in unusual cases (see **ffsmkfs**(1M) and **newfs**(1M)).

special	Is the block device file associated with the partition.
part-number	Is the Intergraph part number found on the Disk Maintenance menu of the Utility pages.

newfs(1M) automatically calculates the size of the file system to be the size of the partition the file system is built on.

For example, to make a FFS on a CDC WREN III drive, device file /dev/dsk/s2u0p7.3, 150,000 blocks, enter the following at the # prompt:

```
newfs /dev/dsk/s2u0p7.3 fdsk155
```

The file system is now ready to be mounted and used.

CAUTION If the Fast File System being created exists, the file system must be unmounted using **umount**(1M) before executing **newfs**(1M) or **ffsmkfs**(1M).

Step 4: Mount the Fast File System using the **mount**(1M) command with the **-f FFS** option. To mount the file system used in the previous example on **/usr2**, enter the following at the # prompt:

```
mount -f FFS /dev/dsk/s2u0p7.3 /usr2

mount: warning: <> mounted as </usr2>
```

To suppress the warning use **labelit**(1M) to label the file system **usr2**.

Step 5: After rebooting, restore any backups made in step 1.

Automation

Fast File Systems can be checked and mounted automatically. For the file system to be checked automatically after system failures, enter the block device used for the file system in **/etc/checklist**. For example, to check our example file system, add the following line to the **/etc/checklist** file:

```
/dev/dsk/s2u0p7.3
```

Fast File Systems can be automatically mounted by putting them in **/etc/fstab** with one additional parameter. To add our example file system, **/usr2**, to the *automounts*, add the following line to the **/etc/fstab** file:

```
/dev/dsk/s2u0p7.3  /usr2  FFS
```

The **FFS** at the end of the line will insert **-f FFS** in the **mount**(1M) command invoked to mount the file system.

For additional information on Fast File Systems see the system administrator's "FFS Tutorial." For additional information on sanity-checking Fast File Systems, see the system administrator's "FFS Check Tutorial."

5. BSD Network Configuration

Procedure 5: BSD Network Configuration

Overview of BSD Network Configuration

Purpose	To describe the form of an Internet address, the steps needed to install it, and the format and use of certain network configuration files.
When Performed	Upon installation of software which uses the TCP/IP networking protocols, e.g. NQS and TCPIP.
Starting Conditions	Log in as super-user.
Commands	**clh**(1) **namex**(1M) **getinet**(1M)
Time	Approximately 20 minutes. (The system may need to be rebooted.)
Caution	The Internet address must be installed before software requiring the address can function.
Reference	See specific software requiring an Internet address for more information.

This procedure describes the configuration requirements for software using the Transmission Control Protocol/Internet Protocol (TCP/IP) communication protocols. Such software includes NQS, **lpr**(1), the remote command utilities in the TCPIP product, and most programs which use the "sockets" inter-process communication facility (see **socket**(2B)) and related routines in the "CLIX Programmer's and User's Reference Manual," as well as the socket tutorials in the "CLIX System Guide"). Topics discussed are the formation and installation of the Internet address used by TCP/IP and the use and maintenance of configuration files which affect the behavior of certain programs using the protocols.

TCP/IP protocols are independent of the Xerox Network Systems (XNS) protocols used on Intergraph systems and use a different addressing scheme. Unlike XNS addresses, which are permanently encoded in hardware, TCP/IP addresses exist only in software and must be configured and initialized before the protocols can be used by software that depends on them.

A TCP/IP address is also referred to as an Internet address, and a machine using the TCP/IP protocols is identified by its Internet address. This address must be supplied by the network administrator. The first section of this tutorial describes Internet addresses and their construction. A list of applications requiring Internet addresses is also provided.

The Internet Protocol (IP) Address Broker Service can generate Internet addresses for Intergraph nodes. If the Broker service is not available, the **getinet**(1M) routine can be used to generate the address.

Certain configuration files control address resolution, access permissions, and other parameters for TCP/IP-based network applications. These files are described in the "Network Configuration Files" section.

The Internet Address

The TCP/IP protocols identify systems by a configurable address known as the *Internet address* or *TCP/IP address*. (The terms are used interchangeably.) A strict convention dictates the format of this address and malformed addresses can cause network problems. Procedures and programs mentioned in this tutorial are provided to assist with address assignment; nevertheless, the network administrator generally needs to be very familiar with the addressing scheme and to take care in assigning addresses to systems on a network.

Products that require an Internet address usually prompt the installer for an Internet address during installation if none exists for the machine. Rebooting the machine initializes the address. The following is a partial list of Intergraph applications that use the TCP/IP protocols and require the machine to have an Internet address.

Applications Requiring an Internet Address

Product	*Description*
lpr (System V)	A remote print spooling program
NFS	Network File System
NQS	Network Queuing System
TCPIP	A group of applications including ftp, rcmd, rcp, rlogin, ruptime, rwho, telnet, tftp, and udpecho
XWINDOWS	The X-Window system

NOTE The product TCPIP does not contain the actual TCP/IP protocols, but rather a group of utilities which use the TCP/IP protocols. The product TCPIP is not needed for any of the other products to run.

In general, Intergraph network applications are either XNS- or TCP/IP-based. An application based on BSD-style socket calls (see **socket**(2B)) is probably TCP/IP-based, thus requiring an Internet address.

Since malformed addresses can cause problems for network nodes other than the one to which the address is assigned, it is important that the following guidelines be kept in mind:

- If the host will not be connected to an official Department of Defense (DoD) network, any valid Internet address can be assigned to the machine.
- If the host will be connected to a DoD network, it must be assigned a registered Internet address from the DoD.

  The Network Information Center (NIC) located at SRI International assigns all Internet addresses. NIC assigns only the network portion of the address and the requesting organization assigns host numbers. (See below for definitions of the "network" and "host" components of the address.) More information is available from the following address:

  Network Information Center
  SRI International
  333 Ravenswood Avenue
  Menlo Park, CA 94025
  (415) 859-3695

Internet Address Notation

An Internet address is a 32-bit number ordinarily expressed as four fields separated by periods, each consisting of three digits, as in the following:

nnn.nnn.nnn.nnn

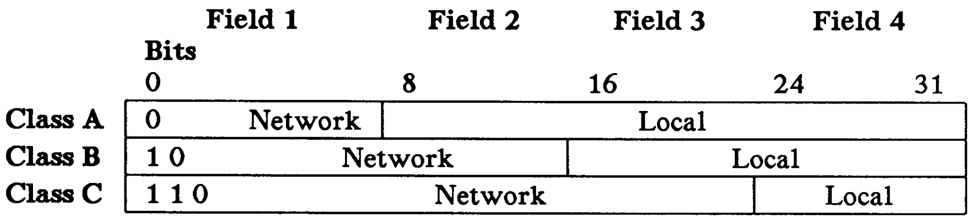
This convention is known as *Internet address notation*. Each field represents eight bits of the 32-bit address. Within each field, leading zeroes can be omitted. The largest permissible value in a field is 255. The following are examples of Internet address notation:

1.1.1.1
1.0.0.12
1.2.123.255

Each address has two portions. The leftmost portion is a network number, which identifies the local network on which the node resides. The rightmost portion is a host number, which identifies an individual node within a network.

NOTE The network portion of the Internet address must be the same for all nodes on a Local Area Network (LAN).

The leftmost three bits of the Internet address are used to divide the address space into three network classes. The allocation of the remaining bits to the network and host portions depends on the class to which the network belongs. The three classes are defined as follows:



Within Class A, only a few networks can exist, each having a large number of nodes. In contrast, Class C provides a large number of networks, each having relatively few nodes (254 maximum). The scheme limits permissible values for individual fields within each class to the following:

	Field 1	Field 2	Field 3	Field 4
Class A	0-127	0-255†	0-255†	0-255†
Class B	128-191	0-255	0-255†	0-255†
Class C	192-223	0-255	0-255	1-254†

† The host number cannot contain all binary 0s or all binary 1s because these values have special meaning within the TCP/IP protocols. In terms of Internet address notation, this means that fields composing the host number cannot all be 0 or all be 255.

NOTE A network number of 127 should not be used because it has the meaning *loopback*.

The Department of Defense protocols allow omission of leading fields in the local portion if they contain all zeros. (A network ID of 0 indicates the local network.) For example, the following Class A Internet address can be shortened to 126.111 because the two fields of zeros are part of the local address.

		Network	**Local**
126.000.000.111	or	126.	000.000.111

Internet Address Installation

Users of network applications generally wish to reference systems by name rather than address. This need for binding names to addresses and for propagating these bindings to all nodes on the network is handled differently by XNS and TCP/IP. The Intergraph implementation of XNS relies on the clearinghouse facility (see **clh**(1)) to automate the maintenance of name bindings. The TCP/IP counterpart of this task is, in most implementations, performed by manually entering name and address pairs in a text file called **/etc/hosts** (see **hosts**(4)). The task may be performed in this way on Intergraph systems as well, but extensions to **clh**(1) allow the system administrator to maintain all name and address information in one place. An Internet address may be added to the clearinghouse entry for a host and automatically propagated by **namex**(1M) to **/etc/hosts** on the local node and throughout the network as well.

Entering an Internet Address in the Clearinghouse

To enter an address in the clearinghouse, a line is added to the clearinghouse object for the local host. This line has the following form, where *nnn.nnn.nnn.nnn* is the Internet address:

tcp_address: *nnn.nnn.nnn.nnn*

The address can be entered using the **clh**(1) program to edit the primary entry listed under OWNED NETWORK OBJECTS. This is the preferred method for adding the TCP/IP address to the clearinghouse.

The following command displays the clearinghouse entry, allowing verification that the Internet address has been correctly added to the clearinghouse:

```
clh lookup node-name /full
```

Node-name is the machine's primary name. The Internet address should appear in the display in the same form in which it was entered. Along with other information, the display should contain at least the following lines:

```
Owned:

node_name
Address           : 00012345.00-00-00-00-00-00
tcp_address       : nnn.nnn.nnn.nnn
```

Updating /etc/hosts

TCP/IP applications obtain addressing information from **/etc/hosts**, so an Internet address entered in the clearinghouse must also be entered there. **namex**(1M) performs this service automatically once a day, updating **/etc/hosts** from the information in the clearinghouse. To force this update to occur without waiting for the scheduled execution of **namex**(1M), enter the following command at the super-user prompt:

```
/usr/ip32/inc/namex
```

NOTE **/etc/hosts can be updated by editing it directly rather than running namex(1M) but the system administrator should verify that the Internet addresses match in the clearinghouse and /etc/hosts. A discrepancy between the two is likely to cause problems.**

Propagation of the Internet Address

In order for TCP/IP applications on a node to be fully operational with all other nodes on a network, the node's name and address must be known to all other nodes, and their names and addresses must be known to it. This means that the node's **/etc/hosts** must contain the names of the other nodes, and **/etc/hosts** on the other systems must contain an entry for it.

This propagation proceeds in two steps:

- Each node broadcasts its clearinghouse entry to the other machines on the network. This happens automatically once an hour; it may be forced to happen by execution of **clh -u** *node name* or when the EXIT AND UPDATE option is selected from the **clh**(1) menu.
- **namex**(1M) runs on each machine to update **/etc/hosts** from the clearinghouse. As noted above, this update occurs automatically once a day, but may be forced by explicit execution of **namex**(1M).

When an Internet address has been installed for the first time, the system must be rebooted to reinitialize the TCP/IP protocols with the new address. It is recommended that before this reboot the new address be broadcast using **clh -u** *node name*. If it is inconvenient to wait for **namex**(1M) to perform its next automatic update of **/etc/hosts** on the other nodes, **namex**(1M) must be run manually on all the nodes with which the local node needs to communicate.

Addressing Non-Intergraph Systems

Because non-Intergraph systems do not have the clearinghouse facility, manual editing of **/etc/hosts** is required in order for Intergraph systems to communicate with non-Intergraph systems. The names and addresses of foreign systems must be entered in **/etc/hosts** on the Intergraph systems, and vice versa.

To add a host name entry to **/etc/hosts**, use a text editor to enter a line in the following format:

Internet-addr *hostname* [*alias1 alias2* ...]

On Intergraph systems, care must be taken to avoid editing the file in such a way as to interfere with the operation of **namex**(1M). If the file contains the following line, the address should be added **above** it:

```
#namex entries follow:  DO NOT EDIT BELOW THIS LINE!!
```

namex(1M) modifies only entries below this line, and ignores those above this line.

Changing an Internet Address

To change an Internet address, follow these steps:

1. Update the address in the **/etc/hosts** file by editing it or by running **namex**(1M).
2. Change the address in the clearinghouse using the **clh**(1) program.
3. Reboot the system to initialize the TCP/IP protocols with the new address.

Assigning Internet Addresses and Node Names with the IP Address Broker Service

When downloading some Intergraph products, the user may be prompted to enter the node's Internet address. If the node has not yet been assigned an address, the Internet Protocol (IP) Address Broker Service can be used to assign the address. This section describes the service and how to use it. For more information on Internet addresses and the IP Address Broker Service, please refer to the *CLIX TCP/IP User's Guide*.

NOTE

Serious network problems can occur, both on the Local Area Network (LAN) and on any other network to which the LAN is connected, if addresses are assigned improperly. To avoid this, exercise caution when using the Internet Protocol (IP) Address Broker to assign TCP/IP addresses. The preferred method is for the network administrator to officially request and then assign valid Internet addresses to all nodes on the LAN.

The broker server should be configured to run on only one node on the user's network. If more than one broker server is configured, the files used to assign Internet addresses will be invalid.

Overview of the Broker Service

The IP Address Broker Service is a server program run by a request from an IP Address Broker client. These programs use the XNS Sequenced Packet Protocol (SPP) to communicate over the network. Together, they assign an IP address for the requesting client.

The broker and client are delivered as part of the Intergraph Network Core (INC) product. The broker client is invoked by the installation script of every product that requires an Internet address. Some of these products are the Network Queuing System (NQS), Network File System (NFS), TCPIP, and XWINDOWS. If the system which invoked the client does not already have an Internet address, the client will attempt to contact the server on the broker's host node.

The node on which the broker server resides must be a CLIX system having the alias **ibroker**. The Broker Service does not support VMS or personal computer (PC) nodes. The broker server can be installed on any node, as long as the alias **ibroker** is assigned to the node. The broker host and server must be configured before the broker client is invoked by any host.

Address Broker Terminology

An understanding of the following terms is important for correct use of the Broker Service:

Internet Protocol (IP) Address
The four-byte IP Address is more commonly referred to as an *Internet address*. This address is described more completely in the section entitled "The Internet Address".

Ethernet Address
An Ethernet address is a six-byte address, written as six two-digit hexadecimal numbers, separated by dashes. An XNS LAN number is a four-byte value, written as a single hexadecimal number. An XNS address is a LAN number separated by a period from an Ethernet address (as in 134A1.08-00-36-

00-F2-0A).

Every Intergraph system has a unique Ethernet number. The XNS LAN number is configured in routers and gateways. Each Intergraph system determines the system's LAN number at boot time by polling the routers on the network. Zero or the absence of a LAN number implies the local LAN. Internet addresses are not hard-coded into the system because the format of the host number and network number portions cannot be predetermined.

Node name

Every system has a **node name** by which the system is known. This name can be translated into an XNS address or an Internet address. The node can also be known by one or more additional names referred to as *aliases*.

Network Mask

A network mask is a bitmask in the form of an Internet address, with the bits set to 1 in the portion of the number that is part of the network number and reset to 0 in the portion that is the host number.

A network number can be extended into the host number portion to create a subnet number. A subnet mask has the bits of the network number and the subnet turned on. A network or subnet mask might be written as 255.255.255.0.

Broker Service Files

You have the option of loading the Broker Service when installing the INC product. If loaded on your system, the Broker Service maintains its database in several text files in the **/usr/ip32/inc/broker** directory. A text file must exist for each LAN that the broker services. The file names are in the format *XXXXXXXX.bcf*, where *XXXXXXXX* is the hexadecimal XNS LAN number for that LAN. These files are generated by running the configuration shell script **bconfig.sh**.

The file **broker.config** contains the network mask and the mapping of XNS network numbers to Internet address numbers. Multiple Internet network numbers can be used on a single XNS LAN.

The Broker Service also uses a file named **broker.nets**, which is generated by the configuration script, **bconfig.sh**.

Format of the XXXXXXXX.bcf Files

These files have the same format as the **/etc/hosts** file, which is used on each system to associate node names with Internet addresses.

The files have one node entry per line. Fields in the entry must be separated by at least one space or tab. The first field is an Internet address; the second field is the node name for that address. Optional aliases and a comment, which must begin with a pound sign (#), can follow the node name. This file will be returned to the requesting client as the client's **/etc/hosts** file and the **namex**(1M) utility will then add entries for other hosts on the network.

Format of the broker.nets File

This file is generated by the broker configuration script. The Broker Service uses a specially formatted network name containing the XNS LAN number that is associated with an Internet network number. When a client asks the Broker Service to assign an Internet address, the client sends its XNS LAN number to the Broker Service. The Broker Service then assigns an Internet address from the Internet network number for the specified XNS LAN.

In this file, the broker configuration script makes an entry, named after the XNS LAN number, which resolves to the Internet network (or subnetwork) number of the XNS LAN. The script makes another entry that resolves to the Internet (or subnetwork) mask for the XNS LAN.

The format of the field for the Internet network number is the string **netname1_** (including the underscore), followed immediately by the XNS LAN number in hexadecimal (with no space after the underscore).

The Broker Service server can accommodate multiple Internet network numbers on each XNS LAN and each entry will have a new number appended to the LAN number, as shown in the following:

```
netname1_00013498 129.135.100.0
```

```
netname1_00013498 129.135.100.0
netname2_00013498 129.135.101.0
```

The field for the Internet network mask is similar, but begins with **netmask_**. For example, if the XNS LAN number for one network is 134A1 and its Internet subnetwork number is 129.135.199, the entry in the file might look like the following:

```
netname1_000134A1 129.135.199.0 bldg4net bobsnet #Bob's subnet
netmask_000134A1 255.255.255.0 #netmask for Bob's LAN
```

NOTE The XNS LAN number must have leading zeros to pad the number to eight characters. Also, the entire name, including hexadecimal digits in the XNS LAN number, must be uppercase. If the XNS LAN number is zero, enter eight zeros for the network.

Configuring the Broker Service

To configure the Broker Service on a broker host, perform the following steps at the super-user prompt.

1. Change to the **/usr/ip32/inc/broker** directory. If no TCP/IP applications are in use and if there are no Internet network numbers, skip step 2.
2. In the directory **hostfiles**, put a copy of the file **/etc/hosts** from one machine on each LAN to be serviced by the broker host. These files will be merged and used to generate the files used by the Broker Server. These files should be named *XXXXXXXX.hosts*, where *XXXXXXXX* represents the XNS LAN number of each LAN from which the files are copied.
3. Edit the file **/usr/ip32/inc/broker/broker.admin** and put in the correct name for the person administering the broker.
4. Edit the file **/usr/ip32/inc/broker/broker.config** to set up the correct associations between XNS LANs and Internet networks. At least one line must contain the string **mask:** *X.X.X.X*, where *X.X.X.X* is the Internet network mask to be applied to the addresses. If this line appears more than once, only the last one

read will be used. The text **mask:** must start in the first column, must be lowercase, and must be separated from the address by a space.

The **bconfig.sh** utility supports the use of a single mask only. If more than one mask is in use, the **.bcf** files must be built manually. For each LAN to be used, one or more lines in the **broker.config** file will contain the XNS LAN number and an Internet network number for that LAN. More than one Internet network number maybe assigned to each XNS LAN number.

Each network number in use will have one line in the **broker.config** file. These lines have the following format:

lan: *LLLLLLLL X.X.X.X*

In this example, **lan** must be left-justified and lowercase, and all fields must be separated by spaces. The field *LLLLLLLL* is the hexadecimal XNS LAN number. This field must be padded to eight characters with leading zeros. The number *X.X.X.X* is an Internet network number to be associated with the XNS LAN. We supply an example **broker.config** file.

5. Run the shell script **bconfig.sh**. Any nodes with Internet network numbers that were not in the **broker.config** file will be listed in a file named **BAD_ADDS.bcf**.

NOTE

If the system is not already using Internet network numbers and TCP/IP applications, the .bcf files will be created but will all be empty. This is correct. The Broker Server will add the addresses that it assigns to these files.

6. Add the following line to the **/usr/ip32/inc/server.dat** file:

   !44!D!/usr/ip32/inc/broker/brokerd!root!

7. Create a new Clearinghouse owned object that assigns the name **ibroker** as an alias for the system name. The example, *clh.entry*, shows what this object should look like. This entry would be named **ibroker**, and the **Node name:** field would contain the well-known node name of the system to be the broker host.

If the Broker Service is not installed on the user's LAN, the user can manually supply an Internet address or use the **getinet**(1M) utility to generate an Internet address for the node. Again, the user should use caution and follow directions when using either the Broker Service or **getinet**(1M).

Network Configuration Files

This section describes the network configuration files referenced by TCP/IP-based (socket-based) network application programs. Two types of network configuration files can exist on an Intergraph system that supports the socket service interface to the TCP/IP protocols. One type is referenced by user programs that call network library routines. The other is referenced only by application programs. In addition, the **/etc/inetd.conf** file is referenced by **inetd**(1M), a daemon that can service user-written servers and those delivered in the TCPIP product.

General-Purpose Configuration Files

Network library routines included in the BSD library, **/usr/lib/libbsd.a**, reference particular network configuration files and return network information to the calling program. The application programs in the TCPIP product call some of these routines, but they all can be called by other user programs.

The general-purpose configuration files and the network library routines that reference them are as follows:

File	**Network Library Routines**
/etc/hosts	gethostent, gethostbyname, gethostbyaddr, sethostent, endhostent
/etc/services	getservent, getservbyname, getservbyport, setservent, endservent
/etc/protocols	getprotoent, getprotobyname, getprotobynumber, setprotoent, endprotoent
/etc/networks	getnetent, getnetbyname, getnetbyaddr, setnetent, endnetent

In these files, characters to the right of a # are treated as a comment.

NOTE If one of these files does not exist and a user calls a routine that references it, the routine returns zero and *errno* is set to ENOENT.

The /etc/hosts File

The **/etc/hosts** file lists hosts available to the local machine, their Internet addresses, and optionally their aliases. The local machine should be listed in this file. If an Intergraph machine with TCP/IP does not have an entry in the local **/etc/hosts** file, the **namex**(1M) program appends an entry for that machine to the file. An entry has the following format:

Internet-addr *hostname* [*alias1 alias2* ...]

A line can contain only one entry. Each item in an entry can be separated by blanks or tabs. An example of an **/etc/hosts** file follows:

```
#
# Hosts that are available to this machine
#
191.196.200.1   ted                             # Ted Johnson's machine
191.196.200.2   myip32c         local
191.196.200.3   susan
191.196.200.4   sys49a
```

Since the **/etc/hosts** file will contain site-dependent information, a skeleton file is delivered with the SYSTEMV product if the **/etc/hosts** file did not exist before delivery.

NOTE Although the **namex**(1M) program modifies the /etc/hosts file, the system manager should verify that this file is set up correctly. Most TCP/IP application programs reference this file.

The /etc/services File

The **/etc/services** file lists network services available on the local machine, the port numbers where they reside, the protocols that they use, and optionally their aliases. Note that this file lists well-known services and their port numbers. For a partial list of services that are actually implemented as servers, refer to the **/etc/inetd.conf** file. Entries can be added to the **/etc/services** for new or experimental services. An entry in this file has the following format:

service port_number/protocol [*alias1 alias2* ...]

A line can contain only one entry. Each item in an entry can be separated by blanks or tabs. An example of an **/etc/services** file follows:

```
#
# Network services, Internet style
#
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
daytime         13/udp
netstat         15/tcp
chargen         19/tcp          ttytst source
chargen         19/udp          ttytst source
ftp-data        20/tcp
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
name            42/udp          nameserver      # IEN 116
whois           43/tcp          nicname         # usually to sri-nic
domain          53/udp
domain          53/tcp
hostnames       101/tcp         hostname        # usually to sri-nic
sunrpc          111/udp
sunrpc          111/tcp
erpc            121/udp                         # rpc listener
#
# Host specific functions
```

continued

```
#
tftp            69/udp
rje             77/tcp
finger          79/tcp
link            87/tcp          ttylink
supdup          95/tcp
csnet-ns        105/tcp
uucp-path       117/tcp
untp            119/tcp         usenet
ntp             123/tcp
ingreslock      1524/tcp
#
# UNIX specific services
#
exec            512/tcp
login           513/tcp
shell           514/tcp         cmd             # no passwords used
```

NOTE The /etc/services file is not site-dependent and is delivered with the SYSTEMV product. Many TCP/IP application programs reference this file.

The /etc/protocols File

The **/etc/protocols** file lists network protocols available on the network, their protocol numbers, and optionally their aliases. Entries can be added to this file for new or experimental protocols. An entry in this file has the following format:

protocol protocol_number [*alias1 alias2* ...]

A line can contain only one entry. Each item in an entry can be separated by blanks or tabs. The following is an example of an **/etc/protocols** file:

```
#
# Internet (IP) protocols
#
ip      0    IP     # internet protocol, pseudo protocol number
icmp    1    ICMP   # internet control message protocol
ggp     3    GGP    # gateway-gateway protocol
tcp     6    TCP    # transmission control protocol
pup     12   PUP    # PARC universal packet protocol
udp     17   UDP    # user datagram protocol
```

NOTE The /etc/protocols file is not site-dependent and is delivered with the SYSTEMV product.

The /etc/networks File

The **/etc/networks** file lists networks connected to the local network, their network addresses, and optionally their aliases. An entry in this file has the following format:

network network_addr [*alias1 alias2* ...]

A line can contain only one entry. Each item in an entry can be separated by blanks or tabs. An example of an **/etc/networks** file follows:

```
#
# Internet networks
#
bldg2          129.137
arpanet        10              arpa
ucb-ether      46              ucbether
```

Since the **/etc/networks** file will contain site-dependent information, a skeleton file is delivered with the SYSTEMV product if the **/etc/networks** file did not exist before delivery.

NOTE The system manager should verify that the /etc/networks file is set up correctly.

Application Configuration Files

Many of the applications that use TCP/IP reference configuration files to validate user login information. The configuration files and the application programs that reference them are as follows:

File	Application Programs
/etc/hosts.equiv ~/.rhosts /.rhosts	rcmd, rcp, NQS, lpr, rlogin
~/.netrc	ftp
/etc/hosts.lpd	lpr (lpd)
/etc/ftpusers	ftp (ftpd)

The **/etc/hosts.equiv** and **~/.rhosts** files are general login validation files used for network security. These files are discussed in the "Network Security" section.

NOTE None of these configuration files are delivered with the products. Therefore, the system manager and individual users are responsible for setting them up correctly.

/etc/hosts.equiv and ~/.rhosts

The **/etc/hosts.equiv** and **~/.rhosts** files are general remote request validation files used by several applications. The "Network Security" section discusses these files.

Ftp: the .netrc File

ftp(1) allows the user to specify a host name that **ftp**(1) will attempt to connect to. After the connection is successfully opened (and if the **-n** option was not specified), **ftp**(1) attempts to send user login information to the remote host. To determine the information that should be sent, **ftp**(1) follows an auto-login process. This process entails checking the **.netrc** file in the local user's home directory. If this file does not exist or if this file contains an error, **ftp**(1) aborts the auto-login process and prompts the user for the login information.

The entries in the **.netrc** file can have the following formats. The items in each entry can be separated by spaces, tabs, or newlines.

machine *name*	Identify a remote machine name. The auto-login process searches the **.netrc** file for a **machine** entry that matches the remote machine specified on the **ftp**(1) command line or as an argument on the **ftp**(1) **open**(2) command. Once a match is found, the subsequent **.netrc** entries are processed, stopping when the end of file is reached or another **machine** entry is encountered.
login *name*	Identify a user on the remote machine. If this entry is present, the auto-login process will initiate a login using the specified name.
password *string*	Supply a password. If this entry is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process. Note that if this entry is present in the **.netrc**

file, **ftp**(1) will abort the auto-login process if the **.netrc** file can be read by anyone besides the user.

macdef *name* — Define a macro. This entry functions as the **ftp**(1M) **macdef** command functions. A macro is defined with the specified name; its contents begin with the next **.netrc** line and continue until a null line (consecutive newline characters) is encountered. If a macro named init is defined, it is automatically executed as the last step in the auto-login process.

For example, suppose a user frequently logs in to myip32c as user curt and then invokes the **ftp**(1) commands **cd stuff** and **pwd**. The user could create the following **.netrc** file in his or her home directory. Since this file contains a password entry, only the user should be able to read it.

```
machine      myip32c
login        curt
password     asdjkl
macdef       init
cd stuff
pwd
(a blank line here)
```

Ftpd: the /etc/ftpusers File

When the **ftp**(1) server, **ftpd**(1M), receives user login information from a remote ftp client, it authenticates the user according to these rules:

- The user name must be in the **/etc/passwd** file and must have a password. The **ftp**(1M) client must provide the correct password before any file operations can be performed.
- The user name must not appear in the **/etc/ftpusers** file. This file is often set up to prevent users from logging in as uucp.
- If the user name is ftp or anonymous, it can appear in the **/etc/passwd** file without a password. In this case, the user is allowed to log in without specifying a password.

The second rule implies that a system manager can tell **ftpd**(1M) to prevent particular user names from logging in to the local host using **ftp**(1). For example, to prevent the users root, baduser, and uucp, from logging in, edit the **/etc/ftpusers** file as follows:

```
root
baduser
uucp
```

The third rule says that **ftpd**(1M) must restrict the access privileges of the ftp and anonymous user. So that system security is not breached, the system manager should construct the ftp or anonymous user directory with care. The following rule is recommended:

~ftp	Make the home directory owned by ftp or anonymous and make it unwritable by anyone else.

lpr: the /etc/hosts.lpd File

The **/etc/hosts.lpd** has the same format as the **/etc/hosts.equiv** file discussed in "Network Security." The **/etc/hosts.lpd** allows the clients listed in the file to access the print spooling system, **lpr**(1). This gives access privileges to the remote users who cannot otherwise access the system through **/etc/hosts.equiv** or **.rhosts**.

Suppose the following set of files exists on the local system:

/etc/hosts.equiv	mary/.rhosts	/.rhosts	/etc/hosts.lpd
red	blue ted bill purple mary	yellow	green

The following can access printers from a remote system:

- Any user on system red with the same user name on the local host.
- No user from machines blue, purple, or yellow because **.rhosts** is not checked by **lpr**(1).
- All users on system green with the same user names on the local host.

Inetd: the inetd.conf File

The **inetd**(1M) program listens for requests to communicate with any of the TCP/IP servers listed in the **/etc/inetd.conf** file. When a request is received, **inetd**(1M) invokes the appropriate server. **inetd**(1M) invokes a connection-oriented (TCP) server each time a connection is made by creating a process. This process is passed the connection as file descriptor 0 and is expected to call the **getpeername**(2B) routine to obtain the source host and port.

inetd(1M) invokes a datagram-oriented (UDP) server when a datagram arrives. A process is created and passed a pending message on file descriptor 0. Datagram servers may either connect to their peer, freeing the original socket for **inetd**(1M) to receive further messages on, or take over the socket, processing all arriving datagrams and eventually timing out. The first type of server is multi-threaded; the second type is single-threaded.

inetd(1M) uses the **/etc/inetd.conf** file, which is read at startup and possibly later in response to a hangup signal. Each entry in this file has the following format:

service socket protocol thread user program [*args*]

Service is a service name listed in the **/etc/services** file. *Socket* is a socket type specified as stream or dgram. *Protocol* is a protocol name listed in the **/etc/protocols** file. *Thread* is either **wait** for single-threaded or **nowait** for multi-threaded. *User* is the name of the user to run the server. *Program* is the full path name of the server. Servers that are incorporated in **inetd**(1M) are specified as internal. *Args* is the arguments to be passed to the server. No more than five arguments can be given.

Characters to the right of # are treated as comments. Continuation lines for an entry must begin with a space or tab. An example of an **/etc/inetd.conf** file follows:

```
#
# Internet server configuration database
#
ftp        stream  tcp  nowait  root  /usr/ip32/tcpip/ftpd     ftpd
telnet     stream  tcp  nowait  root  /usr/ip32/tcpip/telnetd  telnetd
exec       stream  tcp  nowait  root  /usr/ip32/tcpip/rexecd   rexecd
login      stream  tcp  nowait  root  /usr/ip32/tcpip/rlogind  rlogind
shell      stream  tcp  nowait  root  /usr/ip32/tcpip/rshd     rshd
#echo      stream  tcp  nowait  root  internal
#discard   stream  tcp  nowait  root  internal
#chargen   stream  tcp  nowait  root  internal
#daytime   stream  tcp  nowait  root  internal
#time      stream  tcp  nowait  root  internal
#echo      dgram   udp  wait    root  internal
#discard   dgram   udp  wait    root  internal
#chargen   dgram   udp  wait    root  internal
#daytime   dgram   udp  wait    root  internal
#time      dgram   udp  wait    root  internal
```

NOTE

The /etc/inetd.conf file is delivered with the TCPIP product. This file can be modified to include user-written servers.

Network Security

Network security is provided to applications through the **/etc/hosts.equiv** and **~/.rhosts** (located in a user's home directory). These two files validate requests from remote client applications and decide whether to grant access to a user account. When system access is granted, the user does not need to supply a password. *Yellow Pages* also provides a similar type of network security. See the "YP Tutorial" for more information.

The use of these two files varies in scope. The system administrator maintains **/etc/hosts.equiv** and defines system-level access to the local machine. On the other hand, **.rhosts** is a user-level file that exists in user account home directories. This file controls request validation for a user's account.

Applications that use these security measures include NQS, **lpr**(1), **rcp**(1), and **rcmd**(1). When a request arrives from a remote machine from one of these applications, security checking is performed as follows:

- First, the password file is checked to determine whether the requested local user exists. Many of the applications default to equivalent remote and local user names. **lpr**(1) bypasses this check because the print service requires only temporary file space while it is waiting to print. Otherwise, the request is denied access to the system.
- For requests with an entry in the **passwd** file, the password is first checked. If the user name does not have a password, the application is granted. Otherwise, the **/etc/hosts.equiv** file is checked to verify that the remote machine (or its alias) is listed. If it is listed and the remote and local user names are identical, the application is granted access. Requests for the **root** bypass this check and proceed to the **/.rhosts** file in the root file system.
- If the host name is not found in the **/etc/hosts.equiv** file, the **~/.rhosts** file in the requested user's home directory is checked. The **.rhosts** file contains host name entries along with an optional user access list. If the remote host name is found and there is no user access list, access is granted only when the remote and local user names are identical. If the remote host name is found along with a user name, access is granted only when the remote (client-end) user name matches this user name. If not, the remote user must be listed to obtain access. **lpr** does not perform this check. See the **hosts.lpd** file in "Applications Configuration Files."

- If all of these checks fail but the requested user has a password entry, the application may request a password. Applications treat password entries without a password differently.
- A failure to successfully complete these checks will usually result in a "Permission Denied" message for applications such as **rcmd**(1), **rcp**(1), NQS, and **lpr**(1).

For example, to allow all users on hosts ted, myip32c, and susan to access the local host as the same user name, the system manager should verify that each user has an entry in the **/etc/passwd** file and should edit the **/etc/hosts.equiv** file so that it looks like this:

```
ted
myip32c
susan
```

To allow only select users on a host to access the local host, the system manager should not add the host name to the **/etc/hosts.equiv** file. Instead, the system manager or the user should add the host name to the **.rhosts** file in each user's home directory. This file must be owned by root or the particular user.

The **/etc/hosts.equiv** file contains only host names. However, the **~/.rhosts** and **/.rhosts** files can contain both host names and user names.

For example, to allow user joe on host sys49a to gain access as user joe on the local host, edit the **.rhosts** file in joe's home directory to contain the following line:

```
sys49a
```

To allow only user mary on host sys49a to gain access as user john, edit the **.rhosts** file in john's home directory to contain the line:

```
sys49a          mary
```

NOTE

Users will not be able to use rcmd(1) and rcp(1) if the /etc/hosts.equiv and .rhosts files are not set up correctly. In addition, an entry in either of these files is invalid if it contains trailing white space—blanks or tabs. NQS and lpr(1) will reference these files but do not require them.

WARNING

A .rhosts file without user name entries is dangerous because anyone from the remote systems listed can access that account. /etc/hosts.equiv files are also dangerous because all users from the remote system will map to the local system by user name.

WARNING

If a user on the local host does not have a password, the /etc/hosts.equiv and ~/.rhosts files are not referenced. Rather, all users from all hosts will be able to gain access as this user. As a security precaution, a system manager should encourage all users to have a password. Users without passwords should have restricted privileges.

6. NFS/YP Installation

Procedure 6: NFS/YP Installation

Overview of NFS/YP Installation

Purpose	To instruct the system administrator on the setup and use of both the Network File System and the Yellow Pages
Starting Conditions	Valid Internet address
Media	Intergraph product ss081
Time	Approximately 30 minutes
Reference	RPC/XDR Tutorial in the Programmer's & User's Tutorials YP Tutorial in the System Administrator's Tutorials

This procedure introduces the network services. The services currently available are described, and some terms in the network environment are defined.

Following that, the two types of service now available on the network (Network File System (NFS) service and Yellow Pages (YP) service) are introduced and explained. Within each of these two sections, information about periodic maintenance and troubleshooting for the service under discussion will be found.

While some of this material is theoretical, its specific implications will be seen reqeatedly as you become familiar with system administration. For example, when running the Yellow Pages you must understand that some typical UNIX procedures have changed in the Yellow Pages environment. That is also true of using the network file system. This chapter covers only aspects of network services necessary for performing the duties of system administration.

Terminology

Any machine that provides one network service is a *server*. A single machine may provide more than one service. A typical configuration would be for one machine to act as both an NFS and a YP server.

In each network service, servers are entirely passive. The servers wait for clients to call them; they never call the clients.

A *client* is any entity that accesses a network service. The term entity is used because the accessor may be an actual machine or a process generated by software.

The degree to which clients are bound to their servers varies with each of the network services. For example, a YP client binds randomly to one of the YP servers by broadcasting a request. At any point, the YP client may decide to broadcast for a new server. However, an NFS client selects a server to mount a given file system from.

In all cases, the client initiates the binding. The server completes the binding subject to access control rules specific to each service. Since most network administration problems occur at bind time, a system administrator should know how a client binds to a server and what (if any) access control policy each server uses.

UNIX and Network Services

Unlike many recently marketed distributed operating systems, UNIX was originally designed without knowledge that networks existed. This "networking ignorance" presents three impediments to linking with currently available high-performance networks:

- UNIX was never designed to yield to a higher authority (like a network authentication server) for critical information or services. As a result, some UNIX semantics are hard to maintain "over the net." For example, it may not always be appropriate to trust user ID 0 (root).
- Some UNIX execution semantics are difficult. For example, UNIX allows you to remove an open file, yet the file does not disappear until closed by everyone. In a network environment, a client UNIX

machine may not own an open file. Therefore, a server may remove a client's open file.

- When a UNIX machine crashes, it takes all its applications down with it. When a network node crashes (whether client or server) it should not drag all of its bound neighbors down. The treatment of node failure on a network raises difficulties in any system and is especially difficult in the UNIX environment. A system of "stateless" protocols has been implemented to circumvent the problem of a crashing server dragging down its bound clients. Stateless here means that a client is independently responsible for completing work, and that a server need not remember anything from one call to the next. In other words, the server keeps no state. With no state remaining on the server, no state needs to be recovered when the server crashes and comes back up. From the client's point of view, a crashed server appears no different than a very slow server.

In implementing UNIX over the network, System V NFS remains compatible with UNIX when possible. However, certain incompatibilities have been introduced. These are typically of two kinds: first, issues that would make a networked UNIX evolve into a distributed operating system, rather than a collection of network services, and second, issues that would make crash recovery extremely difficult from both the implementation and administration point of view.

All incompatibilities are documented in the appropriate sections of this administration manual.

Debugging UNIX in the Network Environment

Most problems involving System V NFS network services are in the one of the following four areas, which are listed in order of probability.

1. The network access control policies do not allow the operation or architectural constraints prevent the operation.
2. The client software or environment is broken.
3. The server software or environment is broken.
4. The network is broken.

The following sections present instructions on how to check for these causes of failure in the NFS and YP environments.

NFS: The Network File System

What is the NFS Service?

The NFS enables users to share file systems over the network. A client may mount or unmount file systems from an NFS server machine. The client always initiates the binding to a server's file system by using the **mount**(1M) command. Typically, a client remembers specific remote file systems and their mount points by placing lines like these in the file **/etc/fstab**:

```
titan:/usr2 /usr2 NFS rw,hard
venus:/usr/man /usr/man NFS rw,hard
```

See **fstab**(4) for a full description of the format.

Since clients initiate all remote mounts, NFS servers control who may mount a file system by limiting named file systems to desired clients with an entry in the **/etc/exports** file. For example:

```
/usr/local                          # export to any machine
/usr2  bigmo larry curley           # export to only these machines
```

Note that path names given in **/etc/exports** must be the mount point of a local file system. See **exports**(4) for a full description of the format.

How NFS Works

Two remote programs implement the NFS service — **mountd**(1M) and **nfsd**(1M). A client's **mount**(1M) request talks to **mountd**(1M), which checks the access permission of the client and returns a pointer to a file system. After the **mount**(1M) completes, access to that mount point and below goes through the pointer to the server's **nfsd**(1M) daemon using **rpc**(4). Client kernel file access requests (delayed-write and read-ahead) are handled by the **biod**(1M) daemons on the client.

Becoming an NFS Server

An NFS server is a machine that exports a file system or systems. The following steps must be taken to enable any machine to export a file system.

1. The super-user must place the mount-point path name of the file system to be exported in the file **/etc/exports**. See **exports**(4) for file format details. For example, to export **/usr/lbin**, the export file would contain the following:

   ```
   /usr/lbin
   ```

   Of course, an NFS server may only export file systems of its own.

2. **/etc/mountd**(1M) must be running for a remote mount to succeed. This is started from the system startup script, usually **/etc/init.d/nfs**.

3. Remote mount also needs some number of **nfsd**(1M) NFS daemon processes to be running on the NFS server. The actual number depends on the number of client NFS requests that the server should be able to handle concurrently and thus depends on the speed and capacity of the server machine. This example shows four **nfsd**(1M) daemons. The system startup script, such as **/etc/init.d/nfs**, should be checked for lines like these:

   ```
   /etc/nfsd 4  >/dev/console
   ```

After these steps, the NFS server should be able to export the named file system.

Remote Mounting a File System

Any exported file system can be remote mounted on a machine, so long as its server can be reached over the network and the machine is included in the **/etc/export** list for that file system. On the machine where the file system is to be mounted, the super-user should type the following:

mount -f NFS *server_name:/file_system /mount_point*

For example, to mount the manual pages from remote machine **elvis** on the directory **/usr/elvis.man** enter the following:

```
mount -f NFS elvis:/usr/man /usr/elvis.man
```

To ensure the file system is mounted where it is expected to be, use the **mount**(1M) command without any arguments. This displays the currently mounted file systems.

Frequently used file systems are listed with any needed options in the file **/etc/fstab**. See **fstab**(4) for the syntax and contents of the file.

Debugging the Network File System

Before trying to debug the NFS, read the section on how the NFS works and also **mount**(1M), **nfsd**(1M), **biod**(1M), **showmount**(1M), **rpcinfo**(1M), **mountd**(1M), **fstab**(4), **mnttab**(4) and **exports**(4). It is not necessary to understand them fully, but you should be familiar with the names and functions of the various daemons and database files.

When tracking an NFS problem keep in mind that, like all network services, there are three main points of failure: the server, the client, or the network itself. The debugging strategy outlined below tries to isolate each individual component to find the one that is not working.

For example, consider a sample mount request made from an NFS client machine:

```
mount -f NFS krypton:/usr/src /krypton.src
```

The example asks the server machine **krypton** to return a file handle (**fhandle**) for the directory **/usr/src**. This **fhandle** is then passed to the kernel in the **mount**(2) system call. The kernel looks up the directory **/krypton.src** and, if everything is correct, ties the **fhandle** to the directory in a mount record. From now on, all file system requests to that directory and below will go through the **fhandle** to the server **krypton**.

The above describes howthe system should work. We will now look at what may go wrong: first, some general pointers and then a list of the possible errors and their possible causes.

General Hints

When network or server problems occur, programs that access hard mounted remote files will fail in different ways to those that access soft-mounted remote files. Hard-mounted remote file systems cause programs to retry until the server responds again. Soft-mounted remote file systems return an error after trying for a while. **mount**(1M) is like any other program; if the server for a remote file system fails to respond, it will retry the mount request until it succeeds. A soft mount will try once in the foreground then background itself and keep trying.

Once a hard mount succeeds, programs that access hard-mounted files will hang so long as the server fails to respond. In this case, NFS should print a "server not responding" message on the console. On a soft-mounted file system, programs will get an error when a file whose server is dead is accessed.

If a client is having NFS trouble, the first check must be to make sure the server is running. From a client, the following should be typed to see if the server is up:

rpcinfo -p *server_name*

It should print a list of program, version, protocol, and port numbers that resembles the following:

```
[program, version, protocol, port]:

[100005, 1, 17, 1072]
[100001, 2, 17, 1081]
[100001, 1, 17, 1081]
[100002, 1, 17, 1078]
[100008, 1, 17, 1075]
[100007, 1, 17, 1035]
[100007, 1,  6, 1027]
[100004, 1,  6, 1026]
[100004, 1, 17, 1024]
```

rpcinfo(1M) can also be used to check if the **mountd**(1M) server is running by typing the following:

```
rpcinfo -u server_name 100005 1
```

This should return the following:

```
proc 100005 vers 1 ready and waiting
```

If these steps fail, a login should be tried on the server's console to see if it is running.

If the server is alive but a client machine cannot reach it, the Ethernet connections between the machines should be checked.

If the server and the network are alive, **ps**(1) should be used to check the client daemons. A **portmap**(1M), **ypbind**(1M), and several **biod**(1M) daemons should be running. For example, typing the following should print lines for **/etc/portmap**, **/etc/ypbind**, and **biod**.

```
ps -ef
```

The four sections below deal with the most common types of failure. The first covers the steps to be taken if a remote mount fails and the next three discuss servers that do not respond when file systems are mounted.

Remote Mount Failed

This section deals with problems related to mounting. If **mount**(1M) fails for any reason, the sections below should be checked for specific details about what to do. They are arranged according to where they occur in the mounting sequence and are labeled with the error message likely to be displayed. It is assumed that YP is in use.

mount(1M) can get its parameters either from the command-line or from the file **/etc/fstab** (see **mount**(1M)). The example below assumes command line arguments, but the same debugging techniques would apply if **/etc/fstab** was the source of the options.

The interaction of the various parts of the **mount**(1M) request should be considered. In the example **mount**(1M) request,

```
mount -f NFS krypton:/usr/src /krypton.src
```

mount(1M) goes through the following steps to mount a remote file system.

1. **mount**(1M) opens **/etc/mnttab** and checks that this mount has not already been done.
2. **mount**(1M) parses the first argument into host **krypton** and remote directory **/usr/src**.
3. **mount**(1M) may call the Yellow Pages binder daemon **ypbind**(1M) to determine which server machine to find the Yellow Pages server on. It may then call the **ypserv**(1M) daemon on that machine to get the Internet protocol (IP) address of **krypton**.
4. **mount**(1M) calls **krypton**'s port mapper to get the port number of **mountd**(1M).
5. **mount**(1M) calls **krypton**'s **mountd**(1M) and passes it **/usr/src**.
6. **krypton**'s **mountd**(1M) reads **/etc/exports** and looks for the exported file system that contains **/usr/src**.
7. **krypton**'s **mountd**(1M) may call the Yellow Pages server **ypserv**(1M) to expand the host in the export list for **/usr/src**.
8. **krypton**'s **mountd**(1M) performs a **getfh**(2) system call on **/usr/src** to get the **fhandle**.
9. **krypton**'s **mountd**(1M) returns the **fhandle**. On the client, **mount**(1M) performs a **mount**(2) system call with the **fhandle** and **/krypton.src**.
10. **mount**(1M) checks to determine whether the caller is super-user and whether **/krypton.src** is a directory.
11. **mount**(1M) performs a **statfs**(2) call to **krypton**'s NFS server (**nfsd**(1M)).
12. **mount**(1M) opens **/etc/mnttab** and adds an entry.

Any of these steps can fail, some of them in more than one way. The sections below describe the failures associated with specific error messages in detail.

```
mount: cannot open /etc/mnttab
```

The table of mounted file systems is kept in the file **/etc/mnttab**. This file must exist before mount can succeed. **/etc/mnttab** is created when the system is booted, and is maintained automatically afterward by the **mount**(1M) and **umount**(1M) commands.

```
mount: /dev/nfsd is already mounted, ... is busy,
        or allowable number of mount points exceeded
```

This message reveals an attempt to mount a file system that is already mounted, or for which an entry is already in **/etc/mnttab**. **mount**(1M) requests that fail with this message will display the name **/dev/nfsd** (a byproduct of the implementation) regardless of the actual mount request.

```
mount: ... or ...., no such file or directory
```

The "-f NFS" or "krypton:" part of the following command was probably omitted.

mount -f NFS krypton:/usr/src /krypton.src

The **mount**(1M) command assumes that a local mount is being performed unless the **-f** flag is used on the command line or the requested directory as listed in **/etc/fstab** specifies file system type NFS.

More simply, this message also appears when, for a correct mount request, the specified local mount point is not an existing directory.

```
mount: cannot open </etc/fstab>
```

mount(1M) tried to look up the information needed to complete a mount request in **/etc/fstab** but no such file existed. This file should be created by the system administrator as part of initial system setup.

```
... not in hosts database
```

The system name specified on the mount request suffixed by the ":" is not listed in the file **/etc/hosts**. The spelling of the host name and placement of the colon in the mount call should be checked.

```
mount: directory argument ...  must be a full path name
```

The second argument to **mount**(1M) is the path of the directory to be covered. This must be an absolute path starting at "/".

```
mount: ... server not responding(1)
: RPC_PMAP_FAILURE - RPC_TIMED_OUT
```

Either the server to which the mount is being attempted is down or its portmapper is dead or hung. You should attempt to log in to that machine: if that attempt succeeds, the problem may be in the portmapper. Run the following from your system as super-user to test the portmapper on the server system:

```
rpcinfo -p [hostname]
```

The result should be a list of registered programs. If not, the remote portmapper must be killed and restarted. Restarting the portmapper is a complicated process because all registered services are lost, and their associated daemons must be restarted also. This is accomplished by the super-user as follows. The following finds the process ids of **portmap**(1M) and other service daemons:

```
ps -ef
```

The following kills the daemons:

```
kill -9 portmap_pid daemon_id1 daemon_id2
```

The following is an example to start new daemons:

```
/etc/portmap
/etc/mountd
/etc/nfsd
```

Another alternative is to reboot the server when convenient. Because of the stateless nature of the server implementation, there should be no adverse effect on the clients of the system other than the time that they will suspend awaiting the server to return.

If the server is up but it you cannon**rlogin** to it, the client's Ethernet connection should be checked by trying to **rlogin** to some other machine. The server's Ethernet connection should also be checked.

```
mount: ... server not responding: RPC_PROG_NOT_REGISTERED
```

This means that **mount**(1M) got through to the portmapper, but the NFS mount daemon **mountd**(1M) was not registered. The server should be checked to ensure that **/etc/mountd** exists and is running.

```
mount: /dev/nfsd or ..., no such file or directory
```

Either the remote directory does not exist on the server or the local directory does not exist. Again, **/dev/nfsd** will always be printed to represent the remote directory.

```
mount: access denied for  ...: ...
```

Your machine on which the mount attempt is being made is not in the server's export list for the file system to be mounted. A list of the server's exported file systems can be obtained by running the following as super-user:

showmount -e [*hostname*]

If the file system wanted is not in the list or the machine name is not in the user list for the file system, the **/etc/exports** file on the server should be checked for the correct file system entry. A file system name that appears in the **/etc/exports** file but not in the output from **showmount**(1M) indicates a failure in **mountd**(1M). Either it could not parse that line in the file, it could not find the file system, or the file system name was not a locally mounted file system. See **exports**(4) for more information.

This message can also indicate that authentication failed on the server. It may be displayed because the machine attempting the **mount**(1M) is not in the server's export list, the server is not aware of the machine, or the server does not believe the identity of the machine. The server's **/etc/exports** should be checked.

```
mount: ...: no such file or directory
```

The remote path on the server is not a directory.

```
mount: not super user
```

mount(1M) can only be run by the super-user because it affects the file system for the whole machine.

Programs Hung

If programs hang while performing file related work, the NFS server may be dead. The message may be displayed on the machine's console. The message includes the name of the NFS server that is down.

```
NFS server sysname not responding, still trying
```

This is probably a problem either with one of the NFS servers or with the Ethernet. Programs can also hang if a YP server dies.

If a machine hangs completely, the server(s) from which file systems were mounted should be checked. If one (or more) of them is down, client machines may hang. When the server comes back up, programs will continue automatically and will not be affected.

If a soft-mounted server dies, other work should not be affected. Programs that time-out trying to access soft mounted remote files will fail, but it should still be possible to work on other file systems.

If other clients of the server seem to be functioning correctly, the Ethernet connection and connection of the server should be checked.

Everything Works Slowly

If access to remote files seems unusually slow, the server should be checked by entering (on the server) the following as super-user:

```
ps -ef
```

If the server is functioning and other users are getting good response, block I/O daemons on the client should be checked by typing **ps -ef** (on the client) and looking for **biod**(1M). If the daemons are not running or are hung, they should be killed by typing the following as super-user. The following finds the process ids:

```
ps -ef | grep biod
```

The following kills the daemons:

```
kill -9 pid1 pid2 pid3 pid4
```

The daemons should then be restarted with the following:

```
/etc/biod 4
```

To determine whether the daemons are hung, **ps**(1) should be used as above, then a large file should be copied. Another **ps**(1) will show whether the **biods** are accumulating CPU time: if not, they are probably hung.

If **biod**(1M) appears to be functioning correctly, the Ethernet connection should be checked. **nfsstat -c** and **nfsstat -s** can be used to discover whether a client is doing a lot of retransmitting. A retransmission rate of five percent is high. Excessive retransmission usually indicates a bad Ethernet board, a bad Ethernet tap, a mismatch between board and tap, or a mismatch between the client machine's Ethernet board and the server's board.

Incompatibilities with Earlier UNIX Versions

A few things work differently, or not at all, on remote NFS file systems. The next section discusses the incompatibilities and offers suggestions on working around them.

No SU Over the Network

Under NFS, a server exports file systems it owns so that clients may remotely mount them. When a client becomes super-user, it is denied permission on remote mounted file systems. Consider the following example:

```
$ cd
$ touch test1 test2
$ chmod 777 test1
$ chmod 700 test2
$ ls -l test*
-rwxrwxrwx  1 jsbach       0 Mar 24 16:12 test1
-rwx------  1 jsbach       0 Mar 24 16:12 test2
```

The example is tried again by the super-user:

```
$ su
Password:
# touch test1
# touch test2
touch: test2: Permission denied
# ls -l test*
-rwxrwxrwx  1 jsbach       0 Mar 21 16:16 test1
-rwx------  1 jsbach       0 Mar 21 16:12 test2
```

The problem usually shows up during the execution of a set-uid root program. Programs that run as root cannot access files or directories unless the permission for "other" allows it.

Another aspect of this problem is that ownership of remote mounted files cannot always be changed, specifically, if they are on a server that does not permit users to execute **chown**(1). Since root is treated as the "other" user for remote accesses, only root on the server can change the ownership of remote files. For example, consider a user trying to **chown**(1) a new program, **a.out**, which must be set-uid root. This will not work, as shown below:

```
$ chmod 4777 a.out
$ su
Password:
# chown root a.out
a.out: Not owner
```

To change the ownership, the user must either log in to the server as root and then make the change, or move the file to a file system owned by the user's machine and make the change there. (For example **/usr/tmp** will usually be owned by the local machine.)

File Operations Not Supported

File locking of directories is not supported on remote file systems.

In addition, append mode and atomic writes are not guaranteed to work on remote files accessed by more than one client simultaneously.

Cannot Access Remote Devices

In the NFS it is not possible to access a remote mounted device, any other character or block special file, or named pipes.

Clock Skew in User Programs

Since the NFS architecture differs in some minor ways from earlier versions of UNIX, users should be aware of places where their programs could run up against these incompatibilities. The previous section "Architectural Incompatibilities" discusses features that will not work over the network.

Because each machine keeps its own time, the clocks may be out of sync between the NFS server and client. This might cause problems. For example, consider the following.

Many programs assume that an existing file could not be created in the future. For example, the command **ls -l** has two basic forms of output, depending on how old the file is:

```
# date
Sat Apr 12 15:27:48 GMT 1986
# touch file2
# ls -l file*
-rw-r--r--  1 root          0 Dec 27  1984 file
-rw-r--r--  1 root          0 Apr 12 15:27 file2
```

The first type of output from **ls**(1) prints the year, month, and day of last file modification if the file is more than six months old. The second form prints the month, day, hour, and minute of the last file modification if the file is less than six months old.

ls(1) calculates the age of a file by simply subtracting the modification time of the file from the current time. If the result is greater than six months, the file is "old."

Assume that the time on the server is Apr 12 15:30:31, which is three minutes ahead of the local machine's time:

```
# date
Apr 12 15:27:31 GMT 1986
# touch file3
# ls -l file*
-rw-r--r--  1 root        0 Dec 27  1983 file
-rw-r--r--  1 root        0 Apr 12 15:26 file2
-rw-r--r--  1 root        0 Apr 12  1986 file3
```

The difference between the current time and the library's modify time is an unsigned number, equal to −180 seconds.

Thus, **ls**(1) believes the new file was created long ago.

ls(1) was modified to deal with files created a short time in the future.

In general, users should remember that applications that depend on local time and/or the file system timestamps will have to deal with clock skew problems if remote files are used.

YP: The Yellow Pages Service

What is the Yellow Pages Service?

The Yellow Pages is a distributed network lookup service:

- YP is a distributed system; the database is fully replicated at several sites, each of which runs a server process for the database. These sites are known as YP servers. At steady state, it does not matter which server process answers a client request; the answer will be the same all over the network. This allows multiple servers per network and gives YP service a high degree of availability and reliability.
- YP is a lookup service. It maintains a set of databases that may be queried. A client may ask for the value associated with a particular key within a database and may enumerate every key-value pair within a database.
- YP is a network service. It uses a standard set of access procedures to hide the details of where and how data is stored.

The YP Map

The YP system serves information stored in YP "maps." Each map contains a set of keys and associated values. For example, in a map called *hosts*, all the host names within a network are the keys and the Internet addresses of these host names are the values. Each YP map has a *mapname* used by programs to access it. Programs must know the format of the data in the map. Many of the current maps are derived from ASCII files traditionally found in **/etc**: **hosts**(4), **group**(4), **passwd**(4), and a few others. The format of the data within the YP map is identical, in most cases, to the format within the ASCII file. Maps are implemented as files located in the subdirectories of the directory **/etc/yp** on YP server machines.

The YP Domain

A YP domain is a named set of YP maps. Users can determine and set their YP domains with the **domname**(1) command. Note that YP domains differ from both Internet domains and sendmail domains. A YP domain is a directory in **/etc/yp** where a YP server holds all YP maps. The name of the subdirectory is the name of the domain. For example, maps for the literature domain would be in **/etc/yp/literature**.

A domain name is required for retrieving data from a YP database. A domain name must be set on all machines, both servers and clients. Further, a single name should be used on all machines on a network.

Masters and Slaves

In the Yellow Pages environment only a few machines have a set of YP databases. The YP service makes the database set available over the network. A YP client machine runs YP processes and requests data from databases on other machines. Two kinds of machines have databases: a YP slave server and a YP master server. The master server updates the databases of the slave servers. Changes should be made only to databases on the YP master server. The changes will propagate from the master server to the YP slave servers. If YP databases are created or changed on slave server machines instead of master server machines, the YP's update algorithm will be broken. All database creation and modification should be done on the master server machine.

A server may be a master with regard to one map and a slave with regard to another. Random assignment of maps to server machines could introduce confusion. Users are strongly urged to make a single server the master for all maps created by **ypinit**(1M) within a single domain. This document assumes that one server is the master for all maps in the database.

Yellow Pages Overview

The Yellow Pages can serve any number of databases. Typically these include some files that were found in **/etc**; such as, programs used to read the **/etc/hosts** file to find an Internet address. When a new machine was added to the network, a new entry had to be added to every machine's **/etc/hosts** file. With the Yellow Pages, programs that need to look at the **/etc/hosts** file now perform a Remote Procedure Call (RPC) to the servers to get the information.

Most information describing the structure of the YP system and the commands available for that system is contained in manual pages and is not repeated here. For quick reference, the manual pages and an abstract of their contents is given below.

- **ypserv**(1M) describes the processes that compose the YP system. These are **ypserv**(1M), the YP database server daemon, and **ypbind**(1M), the YP binder daemon. **ypserv**(1M) must run on each YP server machine. **ypbind**(1M) must run on all machines that use YP services, both servers and clients.
- **ypfiles**(4) describes the database structure of the YP system.
- **ypinit**(1M) is a database initialization tool. Many maps must be constructed from files located in **/etc**, such as **/etc/hosts**, **/etc/passwd**, and others. **ypinit**(1M) performs all such construction automatically. In addition, it constructs initial versions of maps required by the system but not built from files in **/etc**; an example is the map "ypservers." This tool should be used to set up the master YP server and the slave YP servers for the first time. It should not be used as a general administrative tool for running systems.
- **ypmake**(1M) describes the use of **/etc/yp/Makefile**, which builds several commonly changed components of the YP's database. These are the maps built from several ASCII files normally found in **/etc**: **passwd**(4), **hosts**(4), **group**(4), and **rpc**(4).
- **makedbm**(1M) describes a low-level tool for building a **dbm** file that is a valid YP map. Databases not built from **/etc/yp/Makefile** may be built or rebuilt using **makedbm**(1M). **makedbm**(1M) may also be used to "disassemble" a map so that the key-value pairs that compose it can be seen. The disassembled form may also be modified with standard tools (such as editors, **awk**(1), **grep**(1) and **cat**(1)), and has the form required for input back in **makedbm**(1M).
- **ypxfr**(1M) moves a YP map from one YP server to another, using the YP itself as the transport medium. It can be run interactively or periodically from **crontab**(1). In addition, **ypserv**(1M) uses **ypxfr**(1M) as its transfer agent when it is asked to transfer a map.

- **yppush**(1M) describes a tool to administer a running YP system. It is run on the master YP server. It requests each **ypserv**(1M) process within a domain to transfer a particular map, waits for a summary response from the transfer agent, and prints the results for each server.
- **ypset**(1M) tells a **ypbind**(1M) process (the local one by default) to get YP services for a domain from a named YP server.
- **yppoll**(1M) asks any **ypserv**(1M) for the information about a single map that it holds internally.
- **ypcat**(1) dumps the contents of a YP map. It should be used when it does not matter which server's version is seen. If a particular server's map is required, users must **rlogin**(1) to that server (or use **rsh**(1)) and use **makedbm**(1M).
- **ypmatch**(1) prints the value for one or more specified keys in a YP map. Once again, there is no control over which server's version of the map is seen.
- **ypwhich**(1) tells which YP server a node is using at the moment for YP services or which YP server is master of some named map.

Yellow Pages Installation and Administration

Setting Up a Master YP Server

To create a new master server, the super-user should **cd**(1) to **/etc/yp**. **ypinit**(1M), with the **-m** option, should then be run. The default domain name and the host name must be set up. **ypinit**(1M) will prompt for a list of other hosts that also will be YP servers. (Initially, this will be the set of YP slave servers, but at some future time any of them might become the YP master server.) It is possible, but not necessary, to add other hosts at this time.

Before you run **ypinit**(1M), the following files in **/etc** should be complete and up-to-date: **passwd**(4), **hosts**(4), **group**(4), and **rpc**(4).

For security reasons, access to the master YP machine may be restricted to a smaller set of users than that defined by the complete **/etc/passwd**. To do so, copy the complete file to a location other than **/etc/passwd**, and delete the undesired users deleted from the remaining **/etc/passwd**. For a security-conscious system, this smaller file should not include the YP escape entry discussed in the next section.

To start providing Yellow Pages services, invoke **/etc/yp/ypserv**. It can be started automatically from the system startup script, **/etc/init.d/yp**, on subsequent reboots by editing this file and removing the comment (#) before **/etc/yp/ypserv**.

Adding YP to Clients

Once the decision has been made to serve a database with the YP, all nodes in the network should access the YP's version of the information rather than the potentially out-of-date information in their local files. That policy is enforced by running a **ypbind**(1M) process on the client node (including nodes that may be running YP servers) and by abbreviating or eliminating the files that traditionally implemented the database. The files in question are **/etc/passwd**, **/etc/hosts**, and **/etc/group**. The treatment of each file is discussed in this section.

- **/etc/hosts** must contain entries for the local host's name and the local loopback name. These are accessed at boot time when the YP service is not yet available. After the system is running, and after the **ypbind** process is up, the **/etc/hosts** file is never accessed. An example of the hosts file for YP client **zippy** is as follows:

```
127.1               localhost
192.9.1.87          zippy          # John Q. Random
```

- **/etc/passwd** should contain entries for root and the primary users of the machine, and an escape entry to force the use of the YP service. A few additional entries are recommended: **daemon**, to allow file-transfer utilities to work; **sync**, to run **sync** on a machine before rebooting; and **operator**, to let a dump operator log in. A sample YP client's **/etc/passwd** file follows:

```
root:wAm0Y4lEnf6:0:10:God:/:/bin/csh
jrandom:uHP1gQ2:1429:10:J Random:/usr2/jrandom:/bin/csh
operator:VyZr6V9:333:20:sys op:/usr2/operator:/bin/csh
daemon:*:1:1::/:
```

```
sync::1:1::/:/bin/sync
+::0:0:::
```

The last line tells the library routines to use the YP service rather than give up the search. Entries which exist in **/etc/passwd** will mask analogous entries in the YP maps. In addition, earlier entries in the file will mask later ones with the same user name or the same uid. The order of the entries for **daemon** and for **sync** (that have the same uid) should be noted and duplicated in users' files.

- **/etc/group** may be reduced to a single line:

```
+:
```

This will force all translation of group names and group ids to be made via the YP service. This is the recommended procedure.

Setting Up a Slave YP Server

The network must be working to set up a slave YP server.

To create a new slave server, the super-user should **cd**(1) to **/etc/yp**. From there, **ypinit**(1M) with the **-s** switch should be run, and a host already set up as a YP server should be named as the master. Ideally, the named host is the master server, but it can be any host that has its YP database set up. The host must be reachable. The default domain name on the machine intended to be the YP slave server must be set up and must be set to the same domain name as the default domain name on the machine named as the master.

After running **ypinit**(1M), copies should be made of **/etc/passwd**, **/etc/hosts**, and **/etc/group**. For instance, type the following on the YP slave:

```
cp /etc/passwd /etc/passwd-
```

To ensure that processes on the slave server will actually use the YP services rather than the local ASCII files the original files should be edited in accordance with the section above on altering the client's database. This ensures that YP slave server is also a YP client. Backup copies of the edited files should be made. For instance, type the following on the YP slave:

```
cp /etc/passwd /etc/passwd+
```

After the YP database is set up by **ypinit**(1M), you should enter **/etc/yp/ypserv** to start YP services. On subsequent reboots, it will start automatically from **/etc/init.d/yp**.

Setting Up a YP Client

To set up a YP client, edit the local files as described above in the section on altering a YP client's file database. If **/etc/yp/ypbind** is not running, it should be started. With the ASCII databases of **/etc** abbreviated and **/etc/yp/ypbind** running, the processes on the machine will be clients of the YP services. At this point, a YP server must be available; many processes hang if no YP server is available while **ypbind**(1M) is running. The possible alterations to the client's **/etc** database (discussed above in the section on altering the client) should be noted. Because some files may not exist or may be specially altered, the ways in which the ASCII databases are used are not always obvious. The escape conventions used within those files to force inclusion and exclusion of data from the YP databases are found in the following manual pages: **passwd**(4), **hosts**(4), and **group**(4). In particular, note that changing passwords in **/etc/passwd** (by editing the file or by running **passwd**(1)) will only affect the local client's environment. The YP password database should be changed by running **yppasswd**(1).

Modifying Existing Maps

Databases served by the YP must be changed *on the master server*. The databases expected to change most frequently, such as **/etc/passwd**, may be changed by first editing the ASCII file and then running **make**(1) on **/etc/yp/Makefile** (see **ypmake**(1M)).

Databases expected to change rarely, or databases for which no ASCII form exists (such as, databases that did not exist before the YP) may be modified "manually." In this case, **makedbm**(1M) with the **-u** switch is used to disassemble them into a form that can be modified using standard tools (such as **awk**(1), **sed**(1), or **vi**(1)). A new version should be built using **makedbm**(1M). This may be done by hand in two ways:

- The output of **makedbm**(1M) can be redirected to a temporary file that can be modified and then fed back into **makedbm**(1M).

- The output of **makedbm**(1M) can be operated on within a pipeline that feeds into **makedbm**(1M), again directly. This is appropriate if the disassembled map can be updated by modifying it with **awk**(1), **sed**(1) or a **cat**(1) append, for instance.

Suppose you wish to create a nonstandard YP map called *mymap*. The map is to consist of key-value pairs in which the keys are strings such as al, bl, cl, etc. and the values are ar, br, cr etc.

You may follow two possible procedures when creating new maps. The first is to use an existing ASCII file as input; the second is to use standard input.

For example, consider an existing ASCII file named **/etc/yp/mymap.asc**, created with an editor or a shell script on a machine *ypmaster*. The map is located in the subdirectory **home_domain**. The YP map for this file can be created by typing the following on the YP master:

```
cd /etc/yp
makedbm mymap.asc home_domain/mymap
```

If you wish to include another 2-tuple, to the map, the modification can be made simply.

In all situations like this, the map must be modified by first modifying the ASCII file. Modifications to the map, rather than to the ASCII file, will be lost. The modification should be made on the YP master as follows:

```
cd /etc/yp
make editorial change to mymap.asc
makedbm mymap.asc home_domain/mymap
```

If no original ASCII file exists, the YP map can be created on the YP master as follows. (The default domain is **home_domain**.)

```
cd /etc/yp
makedbm - home_domain/mymap
al ar
bl br
cl cr
<CONTROL>-D
```

To modify the map, **makedbm**(1M) can be used to create a temporary ASCII intermediate file that can be edited using standard tools. For instance, type the following on the YP master:

```
cd /etc/yp
makedbm -u home_domain/mymap > mymap.temp
```

At this point **mymap.temp** can be edited to contain the correct information. A new version of the database is created by entering the following commands on the YP master:

```
makedbm mymap.temp home_domain/mymap
rm mymap.temp
```

The preceding paragraphs explained how to use some tools, but almost everything can be done automatically by **ypinit**(1M) and **/etc/yp/Makefile** unless nonstandard maps are added to the database or the set of YP servers is changed after the system is running.

Whether the Makefile in **/etc/yp** or some other procedure is used, the goal is the same: a new pair of **dbm** files must end in the domain directory on the master YP server.

Propagating a YP Map

Propagating a map means moving it from place to place — in general, moving it from the master YP to a slave YP server. Initially, the map is moved by **ypinit**(1M) as described above. After a slave YP server is initialized, updated maps are transferred from the master server by **ypxfr**(1M). **ypxfr**(1M) may be run in three different ways: periodically by **cron**(1M), by **ypserv**(1M), and interactively by a user.

Maps have differing rates of change; for instance, **protocols.byname** may not change for long periods of time, but **passwd.byname** may change several times a day in a large organization. **crontab**(4) entries can be set up to periodically run **ypxfr**(1M) at a rate appropriate for any map in a YP database. **ypxfr**(1M) will contact the master server and transfer the map only if the master's copy is more recent than the local copy.

To avoid a **crontab**(4) entry for each map, several maps with approximately the same change characteristics can be grouped in a shell script, and the shell script can be run from a single crontab script file. Suggested groupings, mnemonically named, can be found in **/etc/yp**:

ypxfr_1perhour, **ypxfr_1perday**, and **ypxfr_2perday**. If the rates of change are inappropriate for a particular environment, these shell scripts can be easily modified or replaced.

The same shell scripts should be run at each YP slave server in the domain. The time of execution from one server to another should be altered to prevent the checking from bogging down the master. To transfer the map from a particular server rather than master can be specified using **ypxfr**(1M)'s **-h** option within the shell script. Finally, maps having unique change characteristics can be checked and transferred by explicit invocations of **ypxfr**(1M) within **crontab**(1).

ypxfr(1M) is also invoked by **ypserv**(1M), responding to a "Transfer Map" request. Such a request is made as an RPC call from **yppush**(1M). **yppush**(1M) is run on the master YP server. It enumerates the YP map **ypserver** to generate a list of YP servers in the domain. To each of the named YP servers, it sends a "Transfer Map" request. **ypserv**(1M) spawns a copy of **ypxfr**(1M), invoking it with the **-C** option, and passes it the information needed to identify the map and to call back the initiating **yppush**(1M) process with a summary status.

In the cases mentioned above, **ypxfr**(1M)'s transfer attempts and the results can be captured in a log file. If **/etc/yp/ypxfr.log** exists, results will be appended to it. No attempt to limit the log file is made: the user is responsible for this. To turn off logging, remove the log file.

In the third case, the user runs **ypxfr**(1M) as a command. Typically, this is done only in exceptional situations, such as when setting up a temporary YP server to create a test environment or when quickly updating a YP server that has been out of service.

Making New YP Maps

Adding a new YP map entails getting copies of the map's **dbm** files in the domain directory on each of the YP servers in the domain. The actual mechanism is described previously. This section will only describe the work required to get the proper mechanisms in place so that the propagation works correctly.

After deciding which YP server is the master of the map, **/etc/yp/Makefile** should be modified on the master server so that the map can be conveniently rebuilt. Typically, a human-readable ASCII file is filtered through **awk**(1), **sed**(1), and/or **grep**(1) to make it suitable for input to **makedbm**(1M). The existing **Makefile** may be consulted as a source for programming examples. Use of the mechanisms already in place in **/etc/yp/Makefile** is recommended when deciding how to create dependencies that **make**(1) will recognize; specifically, the use of **.time** files allows users to see when the **Makefile** was last run for the map.

Support on the YP slave servers for propagating the new maps consists of appropriate entries either in **/usr/lib/crontab**, or in one of the **ypxfr**(1M) shell scripts mentioned in the previous section. To get an initial copy of the map, run **ypxfr**(1M) by hand on each of the slave servers. The map must be globally available before clients begin to access it. If the map is available from some YP servers, but not all, unpredictable behavior will be seen from client programs.

Adding a New YP Server

To add a new YP slave server, some maps on the master YP server must be modified. If the new server is a host that has not been a YP server before, the host's name must be added to the map **ypservers** in the default domain. The sequence for adding a server named **ypslave** to domain **home_domain** on YP master is as follows:

```
cd /etc/yp
(makedbm -u home_domain/ypservers;
 echo ypslave ypslave) | makedbm - tmpmap
mv tmpmap.dir home_domain/ypservers.dir
mv tmpmap.pag home_domain/ypservers.pag
yppush ypservers
```

Note that some commands are displayed on two lines. These may be entered as one long command (even if the line wraps on the screen) or the return and newline may be escaped with a backslash, as shown here.

The new slave YP server's databases should be set up by copying the databases from YP master server ypmaster. Following a remote login to the new YP slave, **ypinit**(1M) should be used as follows on the YP slave:

```
cd /etc/yp
ypinit -s ypmaster
```

The steps described above in the section "Setting Up a Slave YP Server" should then be completed.

Changing the Master Server

To change a map's master, the map at the new master must first be built. Because the old YP master's name occurs as a key-value pair in the existing map, it is not sufficient to use an existing copy at the new master or to send a copy there with **ypxfr**(1M). The key must be reassociated with the new master's name. If the map has an ASCII source file, it should be present in its current version at the new master. The YP map (**example.map**) should be remade locally on the new YP master with the following sequence:

```
cd /etc/yp
make example.map
```

/etc/yp/Makefile must be set up correctly for this to work. If the old master is to remain a YP server, the **/etc/yp/Makefile** should be edited so that **example.map** is no longer made there. This is done by commenting out the section of "oldmaster:/etc/yp/Makefile" that made **example.map**.

If the map only exists as a **dbm** database, it can be remade on the new master by disassembling an existing copy from any YP server and running the disassembled version back through An example follows:

```
cd /etc/yp
ypcat -k example.map | makedbm - mydomain/example.map
```

After making the map on the new master, you must send a new copy of the map to the other (slave) YP servers. However, **yppush**(1M) should not be used, as it will cause the other slaves to try to get new copies from the old master, rather than from the new one.

A typical method is to become super-user on the old master server and type the following:

```
/etc/yp/ypxfr -h newmaster example.map
```

This places a copy on the old master. **yppush**(1M) may now be run. The remaining slave servers still believe that the old master is the current master and will attempt to get the current version of the map from the old master. When they do so, they will get the new map, which names the new master as the current master.

If the method above fails, another alternative must be used. On each YP server machine, the super-user must execute the command sequence shown above. This should be considered the worst case solution.

Debugging a Yellow Pages Client

This section is divided into two parts: the first discusses problems seen on a YP client, and the second discusses problems seen on a YP server.

On Client: Commands Hang

The most common problem at a YP client node is for a command to hang and generate console messages such as the following:

```
yp: server not responding for domain wigwam. Still trying
```

Occasionally, many commands will be seen to hang, even though the system as a whole appears to be working correctly and new commands can be run.

The message above indicates that **ypbind**(1M) on the local machine is unable to communicate with **ypserv**(1M) in the domain "wigwam." This often happens when machines that run **ypserv**(1M) have crashed. It may also occur if the network or the YP server machine is so overloaded that **ypserv**(1M) cannot get a response back to the local machine's **ypbind**(1M) within the timeout period. Under these circumstances, all other YP client nodes on the network will show the same or similar problems. The condition is temporary in most cases: the messages will usually go away when the YP server machine reboots and **ypserv**(1M) returns or when the load on the YP server nodes and/or the Ethernet decreases.

However, in the circumstances described below, the situation will not improve.

- The YP client has not set, or has incorrectly set, *domainname* on the machine. Clients must use a domain name that the YP servers know. **domname**(1) should be used to see the client domain name, and this should be compared with the domain name set on the YP servers. When the domainname is incorrectly set, the following steps should be taken: the super-user on the machine in question should set the domainname with a proper domain name (this assures domain name will be correct every time the machine boots) and set domainname "by hand" so it is fixed immediately. This is done by typing the following at the super-user prompt:

```
domname good_domain_name
```

- If the domain name is correct, the local net should be checked to ensure that it has at least one YP server machine. Users can only bind to a **ypserv**(1M) process on the local network, not on another accessible network. At least one YP server for each machine's domain must be running on the local network. Two or more YP servers will improve availability and response characteristics for YP services.

- If the local network has a YP server, it should be checked to ensure that it is running. Other machines on the local network should be checked. If several client machines have problems simultaneously, a server problem should be suspected. The **ypwhich**(1M) command should be tried on a client machine that is running. If **ypwhich**(1M) never returns an answer, it should be killed. On the YP server machine, the following should be typed to discover **ypserv**(1M) and **ypbind**(1M) processes:

  ```
  ps -ef | grep yp
  ```

  If the server's **ypbind**(1M) daemon is not running, it should be started by typing:

  ```
  /etc/yp/ypbind
  ```

  If a **ypserv**(1M) process is running, a **ypwhich**(1M) should be performed on the YP server machine. If **ypwhich**(1M) returns no answer, **ypserv**(1M) has probably hung and should be restarted. The super-user should kill the existing **ypserv**(1M) process and start **/etc/yp/ypserv**:

  ```
  kill -9 some pid # from ps
  /etc/yp/ypserv
  ```

  If **ps**(1) shows no **ypserv**(1M) process running, one should be started.

On Client: YP Service Unavailable

If YP services are unavailable on one machine although other machines on the network appear to be functioning, many different symptoms may be displayed, including:

- Some commands appear to operate correctly while others terminate, printing an error message about the unavailability of YP.
- Some commands enter a recovery mode particular to the program involved.
- Some commands or daemons crash with obscure messages or no message at all.

For example, messages like the following may be displayed:

```
$ ypcat myfile
ypcat: can't bind to yp server for domain <wigwam>.
       Reason: can't communicate with ypbind.


my_machine$ /etc/yp/yppoll myfile
Sorry, I can't make use of the yellow pages. I give up.
```

In such cases,

ls -l

should be run on a directory containing files owned by many users, including users not in the local machine's **/etc/passwd** file, such as **/usr**. The **ls -l** reporting file owners not in the local machine's **/etc/passwd** file as numbers, rather than names, is one more symptom that YP service is not working.

These symptoms usually indicate that the **ypbind**(1M) process is not running. The command **ps -ef** may be used to check for one. If it is not found, the following should be entered to start it:

```
/etc/yp/ypbind
```

YP problems should disappear.

On Client: Ypbind Crashes

If **ypbind**(1M) crashes almost immediately each time it is started, problems probably lie in some other part of the system. The presence of the **portmap**(1M) daemon should be ascertained by typing the following:

```
ps -ef | grep portmap
```

If it is not running, the machine should be rebooted.

If **portmap**(1M) will not stay up or behaves strangely, problems are probably more fundamental. The state of the network should be checked.

It may be possible to talk to the **portmap**(1M) on the local machine from a machine operating normally. From such a machine, the following should be typed:

rpcinfo -p *your_machine_name*

If **portmap**(1M) is functioning correctly, the output should be as follows:

```
[program, version, protocol, port]:

[100005, 1, 17, 1046]
[100001, 2, 17, 1055]
[100001, 1, 17, 1055]
[100002, 1, 17, 1052]
[100008, 1, 17, 1049]
[100007, 1, 17, 1027]
[100007, 1, 6, 1026]
[100007, 2, 17, 1031]
[100007, 2, 6, 1030]
```

Note that port numbers will be different on different machines. The four entries that represent the **ypbind**(1M) process are as follows:

```
[100007, 1, 17, <port_#>]
[100007, 1, 6, <port_#>]
[100007, 2, 17, <port_#>]
[100007, 2, 6, <port_#>]
```

If these entries are not there, **ypbind**(1M) has been unable to register its services. The machine should be rebooted. If they are there and change each time an attempt is made to restart **/etc/yp/ypbind**, the system should be rebooted even if the **portmap**(1M) is up.

On Client: Ypwhich Inconsistent

When **ypwhich**(1M) is used several times at the same client node, the answer that is returned varies — the YP server changes. This is normal. The binding of YP client to YP server will change eventually on a busy network, and when the YP servers are busy. When possible, the system stabilizes when all clients get acceptable response time from the YP servers. As long as the client machine receives YP service, it does not matter where the service comes from. Often a YP server machine gets its own YP services from another YP server on the network.

Debugging a Yellow Pages Server

Different Versions of a YP Map

Since YP works by propagating maps among servers, different versions of a map may be found at servers on the network. This version skew is normal only if transient.

Most commonly, normal update is prevented when some YP server or some gateway machine between YP servers is down during a map transfer attempt. When all YP servers and all gateways between them are running, **ypxfr**(1M) should succeed.

If a particular slave server has problems updating, **ypxfr**(1M) should be run interactively on that server. If **ypxfr**(1M) fails, a reason will be given. This assists in understanding and fixing the problem. If **ypxfr**(1M) fails intermittently, a log file should be created to enable logging of messages. The program below saves all output from **ypxfr**(1M).

```
cd /etc/yp
touch ypxfr.log
```

The output looks much like the output generated when **ypxfr**(1M) is run interactively, but each line in the log file is timestamped. The timestamp shows when **ypxfr**(1M) began its work. If copies of **ypxfr**(1M) ran simultaneously, but their work took differing amounts of time, they may write their summary status line to the log files in an order different to the orderin which they were invoked. Any pattern of intermittent failure will appear in the log. When the problem is fixed, logging should be turned off by removing the log file. If this is not done, the log file will grow without limit.

Also, on the problem YP slave server, the **/usr/lib/crontab** file and the **ypxfr*** shell scripts that it invokes should be inspected. Typing mistakes in these files will cause propagation problems, as will failures to refer to a shell script within **crontab**(1) or failures to refer to a map within any shell script.

A check must be made to ensure the YP slave server is in the map **ypservers** within the domain. If it is not, it will function correctly as a server, but will not be notified by **yppush**(1M) when a new copy of a map exists.

Ypserv Crashes

When the **ypserv**(1M) process crashes almost immediately and will not stay up even with repeated activations, the method for debugging is virtually identical to that described in the previous section "On Client: Ypbind Crashes." The **portmap**(1M) daemon should be checked on the YP server:

```
ps -ef | grep portmap
```

If it is not found, the server should be rebooted. If it is present, the following should be typed on the YP server:

rpcinfo -p

Output should be similar to the following example.

```
[program, version, protocol, port]:

[100001, 2, 17, 1062]
[100001, 1, 17, 1062]
[100002, 1, 17, 1060]
[100008, 1, 17, 1058]
[100005, 1, 17, 1056]
[100007, 1, 17, 1032]
[100007, 1, 6, 1027]
[100004, 1, 6, 1026]
[100004, 1, 17, 1024]
[100004, 2, 6, 1043]
[100004, 2, 17, 1040]
```

The four entries representing the **ypserv**(1M) process are as follows:

```
[100004, 1, 6, <port_#>]
[100004, 1, 17, <port_#>]
[100004, 2, 6, <port_#>]
[100004, 2, 17, <port_#>]
```

If they are not there, **ypserv**(1M) has been unable to register its services. The machine should be rebooted. If they are there and they change each time an attempt is made to restart **/etc/yp/ypserv**, the machine should be rebooted.

Yellow Pages Policies

This section describes the policies set by the C library routines when they access the following files on a system running the Yellow Pages.

/etc/passwd
Always consulted. If there are + or - entries, the YP password map is consulted. Otherwise YP is unused.

/etc/group
Always consulted. If there are + or - entries, the YP group map is consulted. Otherwise YP is unused.

/etc/hosts
Never consulted. The data in the YP database is used instead.

Security Under the Yellow Pages

Global and Local Database Files

Of the YP databases, three were formerly in **/etc**: **/etc/passwd**, **/etc/group**, and **/etc/hosts**. (Note that a site may add database files of its own.) The Yellow Pages is divided into local and global file types. A local file is looked for first on the local machine and then in the Yellow Pages. A global file is checked for only in the Yellow Pages. **/etc/passwd** and **/etc/group** are the local files in the Yellow Pages database. The other Yellow Pages files are global.

For example, a program that calls **/etc/passwd** (a local file) will first look in the password file on the local machine; the Yellow Pages password file will only be consulted if the local machine's password file contains + (plus sign) entries. The **/etc/passwd** file is local to provide local control of passwords. The only other local file is **/etc/group**.

The remaining Yellow Pages file (**hosts**(4)) is a global file. The information in this file is network-wide data and is accessed only from the Yellow Pages. However, when booting, each machine needs an entry in **/etc/hosts**. In summary, if Yellow Pages is running, global files are only checked in the Yellow Pages; a file on a local machine is not consulted.

Security Implications

An **/etc/passwd** file and **/etc/group** file may also have + entries. A line in an **/etc/passwd** file such as the following pulls in an entry for **nb** from the Yellow Pages.

```
+nb::::Napoleon Bonaparte:/usr2/nb:/bin/sh
```

It gets the uid, gid, and password from the Yellow Pages, and gets the gecos, home directory, and default shell from the + entry itself. On the other hand, an **/etc/passwd** entry such as the following gets all information from the Yellow Pages.

```
+nb:
```

Note that the following

```
+nb::1189:10:Napoleon Bonaparte:/usr2/nb:/bin/sh
```

differs from

```
nb::1189:10:Napoleon Bonaparte:/usr2/nb:/bin/sh
```

In the first of the two examples, the password field is obtained from the Yellow Pages. In the second, the user **nb** has no password. If no entry for **nb** is in the Yellow Pages, the effect of the first example is as if no entry for **nb** is present.

Special YP Password Change

When a password is changed with the **passwd**(1) command, the entry given in the local **/etc/passwd** file is changed. If the password is not given explicitly but rather is pulled in from the Yellow Pages with a + entry, the **passwd**(1) command will print the following error message:

```
Not in passwd file
```

To change a password in the Yellow Pages, you must use the **yppasswd**(1) command. To enable this service, the system administrator must start the daemon **yppasswdd**(1M) server on the machine serving as the master for the Yellow Pages password file.

Manual Pages Covering Security

More details may be found on the following manual pages: **yppasswd**(1), **exports**(4), **passwd**(4), **group**(4), **yppasswdd**(1M).

What if the Yellow Pages is Not Used?

If you decide not to use the Yellow Pages, the following procedure for bypassing the software implementation must be carried out. In the system startup script, the following line must be commented out:

```
/etc/yp/ypbind
```

The result will be as follows:

```
#/etc/yp/ypbind
```

In addition, the machine's domain name should be reset by typing the following as super-user:

```
domname ""
```

Adding a New User to a Machine

Adding a new user to a machine involves adding an entry to the password file and creating a home directory on the new user's machine as described in the steps below.

Edit the /etc/passwd File

Typically, for a new user, a password file entry should be added to every machine on the local network. The super-user must do this, starting on the master YP server machine. The first step is to edit the master YP server's **/etc/passwd** file. Later, the password file entry for the user will be copied to the **/etc/passwd** file on the new client's partition; without an entry in it, the person administering the new client machine would not be able to log in should the Yellow Pages fail.

On the master YP server, a new line must be added to the password file. **/etc/passwd** is a readable ASCII file with a one-line entry for each valid user on the system. Each entry is separated into fields by colons (**:**); seven fields are on each line and some fields may be left blank by placing two colons back to back. Using certain characters in the password file must be avoided: these are single and double quotes (' ' "), backslashes (\), and parentheses (( )). **passwd**(4) gives more information about the file format.

If the new user's name is **Mr. Chimp** and his account is to be **bonzo**, a line similar to the following should be added to the password file:

```
bonzo::1947:10:Mr. Chimp:/usr2/bonzo:/bin/sh
```

Note that the second field is blank in the example. This field, when filled, contains an encrypted version of the user's password. When the field is blank, anyone can log in simply by typing the user name; no password is required. It is not possible to create a password by making an entry in the **/etc/passwd** file: the **passwd**(1) command must be used by someone logged in either as the user in question, or as super-user. Since anyone can log in when a user has no password, it may be preferable to provide a password for the new user and let him know it so he can log in and change it using **passwd**(1) or **yppasswd**(1) to change it in the YP database.

After **Mr. Chimp** has a password, the entry for **bonzo** in the password file will resemble the following:

```
bonzo:3u0mRdrJ4tEVs:1947:10:Mr. Chimp:/usr2/bonzo:/bin/sh
```

Fields in the password file have the following meanings:

1. Login name — synonymous with user name.
2. Encrypted password. All users should be told how to add or change their passwords with the **passwd** and **yppasswd** commands. The system administrator can empty this field when a user has forgotten his or her password, thereby enabling login without a password until a new one is set. Note that an asterisk (*) in this field matches no password.
3. User ID. A number unique to this user. A system knows the user by ID number associated with login name; therefore, a login name must have the same user ID number on all password files of machines networked in a local domain. Failure to keep IDs unique will prevent users from moving files between directories on different machines, because the system will respond as if the directories are owned by two different users. In addition, file ownership may become confused when an NFS server exports a directory to an NFS client whose password file contains users with uids matching those of different users on the NFS server.
4. Group ID. This field may be used to group users who are working on similar projects. All system staff are in group "10" for historical reasons. In the example above, Mr. Chimp is in the system staff group. Normal users should not be placed in this group. Guidance on which group to put a new user in may be obtained from **group**(4) and the file **/etc/group**.
5. Information about the user — usually real name, phone number, etc.
6. The user's home directory — the directory the user logs in to.
7. Initial shell to use on login. If this field is blank, the default **/bin/sh** is used.

After the password file is updated and a password created for the new user, the Yellow Pages database must be updated by running **/etc/yp/make** for **/etc/passwd**:

```
cd /etc/yp
make passwd
```

Make a Home Directory

After adding a new entry to the password file, a home directory should be created for the new user to log in to. This will be the same as the directory given in the sixth field of the password file entry. In the **/usr2** directory, a directory should be made for the new user. Ownership should be changed to the user's login name and group to the user's group. The following is an example:

```
cd /usr2
mkdir bonzo
chown bonzo bonzo
chgrp 10 bonzo
```

Note that if the Yellow Pages databases for the password file have not yet been updated on the machine's Yellow Pages server, the following error message will result when **chown**(1) is attempted:

```
unknown user id: username
```

In this case, the following set of commands should be used:

```
cd /usr2
mkdir bonzo
chown <userid#> bonzo
chgrp 10 bonzo
```

The user ID number (from the password file entry) is used instead of login name to change the ownership of the user's home directory.

The New User's Environment

The environment in which the new user is placed at login may be defined in several ways. For example, he may be given a copy of the file **.profile** if he uses the Bourne shell. See the **sh**(1) pages in the *UNIX System V User's Reference Manual* for discussion of this file.

If the new user is a member of any groups on the site, he should be added to **/etc/group** as necessary (see **group**(4) and **groups**(1)). The changes must be made to the **/etc/group** file on the master YP server if the Yellow Pages is used.

7. NQS Installation

Procedure 7: NQS Installation

Overview of NQS Installation

Purpose	To instruct the system administrator in installing the Network Queuing System
Starting Conditions	Log in as super-user System V 3.1 Release INC 3.1 Release
Installation Requirements	Internet Address
Commands	**newprod**(1M)
Reference	"NQS Tutorial" in the System Administrator's Tutorials "NQS Tutorial" in the Programmer's & User's Tutorials

This chapter provides the instructions needed to install NQS on your CLIPPER workstation or InterServe processor.

Installation Requirements

To install NQS on your workstation or InterServe, certain conditions must exist. The following subsections discuss these conditions.

Software Requirements

To install NQS, the following software must be installed on your workstation or InterServe:

- System V 3.1 kernel (UNIXBOOT)
- System V 3.1 File System (SYSTEMV)
- Workstation Network Software 3.1 Release (INC)

Internet Address Requirement

Your node must have a valid Internet address before you can initialize NQS. The Internet address is explained in this section.

The Transmission Control Protocol/Internet (TCP/IP) protocols use the Internet address to identify a node on the network. An Internet address is a number with the following format, where *nnn* is between 1 and 223 in the first *nnn* set and between 1 and 254 in the following sets.

nnn.nnn.nnn.nnn

To determine whether your node has a current Internet address, key in the following at the system prompt:

netaddr

If an Internet address exists, it will display in the following format. (If an Internet does not exist, only the first line will display.)

```
00012345.00-00-00-00-00-00
Internet Address: nnn.nnn.nnn.nnn
Subnet Mask: 255.255.255.0
```

If your node does not have an Internet address, you will be prompted to assign one when you install NQS. Because you supply the Internet address, you should abide by your site's numbering scheme to prevent address duplication. Check with your system administrator before assigning an Internet address. (See the "Delivering NQS for the First Time" section.)

NOTE — The TCPIP product does not need to be installed before you can use the TCP/IP protocol.

Delivering NQS for the First Time

To install NQS on your node, follow the steps in the succeeding sections.

Step 1: Invoke newprod

Invoke **newprod**(1M) from the system super-user prompt. (See the "New Product Delivery" procedure.) Select the Network Queuing System (SSS0126) product. You must install NQS in the **/usr** file system.

If you do not have an Internet Address, you will be prompted for one. If you need to assign an Internet Address, proceed to step 2. If you already have an Internet address, skip to step 3.

Step 2: Assign an Internet Address

NOTE

If you are connecting to a Department of Defense (DoD) network, you must obtain your Internet address from the DoD. Consult your system administrator for instructions before you assign your Internet address. (See the "BSD Network Configuration Tutorial" for details about Internet addresses.)

If your workstation or InterServe does not have an Internet address, the system prompts for it now. (If your workstation or InterServe has an Internet address, no prompt will display and you may skip to step 3.)

Enter an address with the following format, where *nnn* is between 1 and 223 in the first *nnn* set and between 1 and 254 in the following sets. (You may let the system generate a temporary address for you.)

nnn.nnn.nnn.nnn

After you enter the Internet address, the installation procedure continues.

Step 3: Answer the Installation Prompts

To save disk and process space, a "client side only" option is available when NQS is downloaded. This option will prevent server-related executables from being downloaded and will keep the network daemon from starting. While this option saves space, remote nodes cannot use the remote **qstat**(1) command option to display requests on this node.

The installation script displays the following prompts:

```
Will this machine be a CLIENT only? [n]

Do you want the NQS logging process started by default? [y]
```

To save process space, the "no log process" option is available. This will prevent the NQS logging process from starting. If the question is answered "no," logging will still be directed to the file specified in the **qmgr**(1M) configuration. However, error messages will no longer appear on the console and the time will not appear on messages logged.

```
Default file system for spooled data file [/usr]
```

If you have more free space in another file system (such as **/usr2**) and would like your data files to spool to that file system, key in the file system at the prompt. Otherwise, press <RETURN> to accept the default.

The installation script will ask you the following question:

```
Do you want to map all remote users to user [rje] by default? [y]
```

If you want all users to have print and plot access to your node through the rje account, accept the default. If you require additional system security, answer "no" and read the information on the **hosts.nqs** file in the System Administrator's "NQS Tutorial."

```
Do you want to load the pconfig utility? [y]
```

We recommend that you use the **pconfig** utility to create your printing and plotting queues. Press return to accept the default and load **pconfig**. (See the *Network Queuing System (NQS) User's Guide* for more information on **pconfig**.) A message displays when the installation is complete.

Step 4: Initialize NQS

After you have installed NQS, reboot your workstation or InterServe to initialize it. You may initialize NQS manually if both of the following conditions are true:

- The INC, SYSTEMV, and UNIXBOOT products are present and initialized.
- An Internet address was assigned to your workstation or InterServe before NQS was installed.

To manually initialize NQS, key in the following at the super-user prompt:

```
/etc/init.d/nqs start
```

Delivering NQS Updates

After you install NQS for the first time, you will need to initialize NQS software updates by shutting down NQS and then restarting it or by rebooting your machine.

Shutting Down NQS

Any privileged user (or the super-user) can shut down NQS. (See the "NQS privileges" section in the System Administrator's "NQS Tutorial" for information on NQS privileges.) To shut down NQS, key in the following at the system prompt:

echo shutdown [*seconds*] | **qmgr**

All active NQS processes exit. The remaining processes are killed after the specified number of seconds. If seconds are not specified, the waiting period defaults to 20 seconds.

Using the Queue Manager to Shut Down NQS

You may also shut down NQS from the NQS Queue Manager (**qmgr**(1M)) utility. To shut down NQS in **qmgr**(1M), key in **qmgr** at the super-user prompt. Then, wait for the Mgr: prompt and key in the **shutdown** command as follows, where *grace_period* is the time (in seconds) that NQS will wait before shutting down.

qmgr
Mgr: **shutdown** *grace_period*

If you do not specify the grace period, NQS defaults to 20 seconds.

Restarting NQS

When NQS has been loaded and initialized on your workstation or server, it will normally start up automatically each time you boot. If you shut NQS down for any reason, restart it by keying in the following at the super-user prompt:

/etc/init.d/nqs start

Part 3: Programmer's and User's Tutorials

1. Technical Programming Tutorial

Technical Programming Tutorial

Introduction

This document is the Technical Programming Tutorial for CLIX System V. CLIX System V is a derivative of AT&T's UNIX System V for the Intergraph CLIPPER 32-bit CMOS microprocessor. This manual provides programming information for a software developer responsible for implementing applications software on a CLIX system.

This tutorial should be used with standard AT&T UNIX System V documentation. You should also have Intergraph CLIPPER documentation and other miscellaneous books and documents.

The reader should be familiar with UNIX System V. Certain well-known program names or acronyms (such as pcc) are used liberally without explanation. Knowledge of operating system and programming language porting issues is also desirable, but not required.

Overview of Chapters

Introduction
This chapter introduces the tutorial.

General Description
This chapter describes the CLIX System in very general terms. Original development goals and major decisions are discussed. It should aid the reader in understanding the approach used to develop CLIX System V.

Implementation Details
Low-level details, such as stack frames and executable file formats (COFF), are discussed. It is intended to be the definitive reference for implementation details and differences from UNIX System V.

Assembler
The CLIPPER assembler command invocation and input requirements are presented.

General Description

This chapter describes CLIX System V in general terms. Original development goals, new features, differences from UNIX System V Release 3.1, and major decisions are discussed.

CLIX System V Release 3.1 is a full-feature derivative of UNIX System V Release 3.1. It was developed to provide full UNIX System V compatibility while taking advantage of the CLIPPER architecture. CLIX System V was also carefully implemented to allow straightforward adaptation to specific CLIPPER-based systems.

The baseline version of UNIX System V Release 3.1 from AT&T is available only for the AT&T 3B2 computer. Throughout this manual, references to the 3B2 implementation are provided for comparison purposes.

Full-Feature Kernel Implementation

All UNIX System kernel features have been implemented and tested. These include features such as STREAMS and shared libraries. Also, features from previous releases, such as demand-paged virtual memory and interprocess communication, have been implemented and verified.

High-Quality Language Processing Tools

Intergraph contracted with Green Hills Software, Inc. to develop the CLIPPER C, FORTRAN, and Pascal compilers. The Green Hills compilers perform many standard optimizations (constant expression folding, operator strength reduction, loop invariant code motion, etc.) resulting in high-quality code generation. The UNIX System V "C" compiler, **pcc**(1), and FORTRAN compiler, **f77**(1), were not ported.

System V Interface Definition (SVID) Compatibility

AT&T has published a detailed functional description of most of the features (programs, libraries, system calls, etc.) supported in UNIX System V. It is called the *System V Interface Definition*, or SVID. Compatibility between the SVID and CLIX System V is the most important major issue. When the SVID is imprecise or incomplete, the 3B2 implementation of UNIX System V is used as the functional reference.

Some parts of the software were enhanced to take advantage of the CLIPPER architecture where big performance improvements can be realized. This has been done only when such modifications do not compromise SVID compatibility.

Implementation Details

This chapter discusses a broad range of CLIX System V Release 3.1 implementation details. Programming language calling sequence and other conventions, executable file formats, standard virtual memory addresses, and significant differences from the AT&T release are presented.

Data Types

CLIPPER memory is byte-addressed with 32-bit addresses. Bytes are ordered with the least significant byte of a multibyte value stored at the lowest address. Bits are numbered with bit zero as the least significant bit.

Character encoding is ASCII. Floating point is IEEE format (32 and 64 bits), with the least significant byte at the lowest address. Table 1-1 shows the data types implemented by the compilers along with their sizes and alignments.

Pascal Data Types

Packed fields of records are allocated starting at bit zero. Every packed field of a record must be fully contained in four or fewer bytes. Each record or array is aligned to the maximum alignment requirement of any of its components.

C Data Types

Bit fields are allocated starting at bit zero. Every bit field is fully contained in four or fewer bytes. Each structure, union, or array is aligned to the maximum alignment requirement of any of its components.

Table 1-1 CLIPPER Data Types

Type	**Length (bits)**	**Alignment (bytes)**
Pascal Data Types		
integer	32	4
real	64	8

Table 1-1 CLIPPER Data Types (continued)

Type	Length (bits)	Alignment (bytes)
C Data Types		
char	8	1
unsigned char	8	1
short	16	2
unsigned short	16	2
int	32	4
unsigned int	32	4
long	32	4
pointers	32	4
float	32	4
double	64	8
enum (default)	32	4
enum (-X6)	8,16,32	1,2,4
FORTRAN Data Types		
INTEGER (default)	32	4
INTEGER (-i2)	16	2
INTEGER*1	8	1
INTEGER*2	16	2
INTEGER*4	32	4
LOGICAL	32	4
LOGICAL*1	8	1
LOGICAL*2	16	2
LOGICAL*4	32	4
REAL	32	4
REAL*4	32	4
REAL*8	64	8
DOUBLE PRECISION	64	8
CHARACTER*1	8	1
CHARACTER**n*	8**n*	1
COMPLEX	64	4
COMPLEX*8	64	4
COMPLEX*16	128	8
DOUBLE COMPLEX	128	8

Calling Sequence

The discussion that follows describes the calling sequence used for subroutine linkage. The terms *caller* and *callee* are used. The *caller* is a routine that invokes another routine, specifically the code near the actual call. The *callee* is a routine invoked by another routine, specifically the code near the *callee's* entry and exit points.

```
int foo()
{               /* A */
        . . .
        bar();    /* B */
        . . .
}               /* A */

int bar()
{               /* A */
}               /* A */
```

Figure 1-1 *Callee/Caller* Example

Both **foo()** and **bar()** are *callees* near **A**, while **foo()** is a *caller* near **B**.

Registers

The register usages for CLIX System V are described as follows:

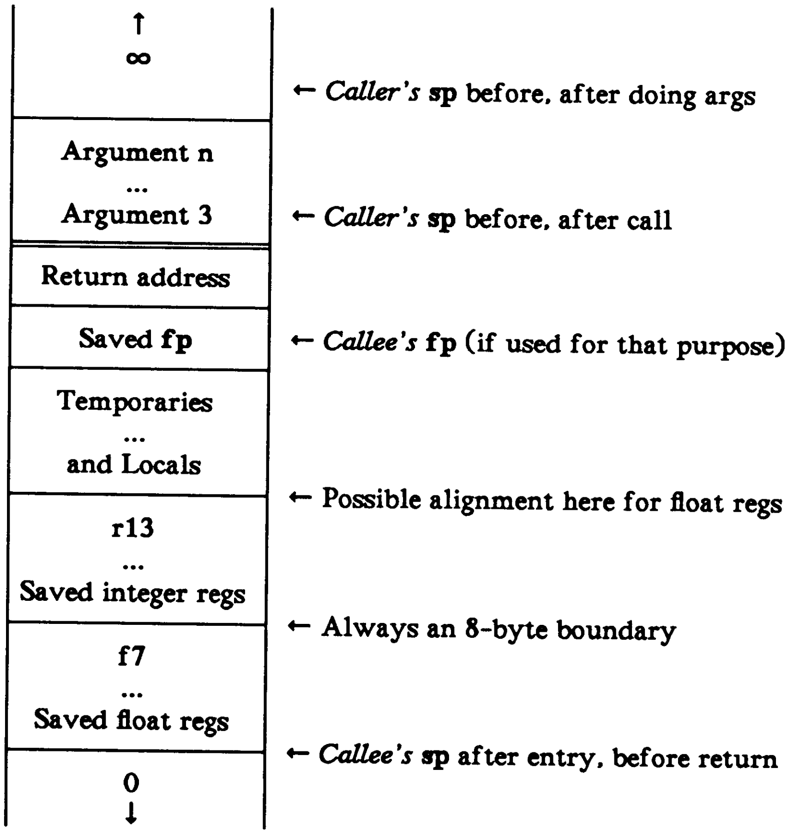
r15 **The stack pointer.** By hardware and software convention, the stack pointer is assigned to **r15**, or **sp**. The stack pointer contains the address of the top valid item on the stack. The stack grows from high to low addresses so that adding items to the stack decrements the pointer. For correct interrupt and signal handling, valid data should not be stored past the end of the stack.

r14	**Optional frame pointer.** In the CLIPPER calling sequence, the frame pointer, **r14** or **fp**, is optional. It is only used by the compilers in a complex routine or if the user directs a compiler to provide a frame pointer for debugging. If used, the frame pointer contains the address of a fixed reference point within the stack from which the arguments, locals, and temporaries may be indexed. By convention, **fp** contains the address of the saved copy of the *caller's* **fp**, providing the head of a linked list of all active stack frames. If **fp** is not used as a frame pointer, it may be used as a permanent register (see below).
r13-r6	**Permanent registers.** The *caller* assumes that the values in these registers remain the same before and after the call. The *callee* must save these registers before using them and must restore the original values prior to returning control to the *caller*. In "C," these registers are assigned to register variables and frequently used values.
r5-r0	**Temporary registers.** The *caller* must assume that the values in these registers were destroyed over a routine call. The *callee* does not have to save them. Registers **r0** and **r1** are used to pass integer and pointer arguments in "C" and the address (reference) of arguments in FORTRAN. Register **r0** is set by the *callee* to the routine's returned value if it is an integer or pointer type. NOTE: Once within the *callee*, the registers used to pass arguments may be used otherwise after the argument is offloaded to another register or temporary storage space.
f7-f4	**Permanent floating-point registers.** Saving requirements are the same as **r14-r6**.
f3-f0	**Temporary floating-point registers.** Saving requirements are the same as **r5-r0**. Floating-point registers **f0** and **f1** are used to pass floating-point arguments in "C." The floating-point register **f0** is set by the *callee* to the routine's returned value if it is a floating-point type.

Permanent registers may be saved with the **savewn** and **savedn** instructions and restored with the **restwn** and **restdn** instructions.

Stack Frame

Figure 1-2 shows the stack frame for the calling sequence. It demonstrates the worst case, that the *caller* is passing arguments on the stack and the *callee* uses permanent integer, floating-point registers and some local stack variables or temporaries. The double line shows the double-word boundary used as an alignment reference point.



↑
∞
← Caller's sp before, after doing args
Argument n
...
Argument 3
← Caller's sp before, after call
Return address
Saved fp
← Callee's fp (if used for that purpose)
Temporaries
...
and Locals
← Possible alignment here for float regs
r13
...
Saved integer regs
← Always an 8-byte boundary
f7
...
Saved float regs
← Callee's sp after entry, before return
0
↓

Figure 1-2 Stack Frame

The first two arguments are usually passed in registers. The first argument is normally passed in **r0** or **f0**, depending on its type. Similarly, the second argument is passed in **r1** or **f1**. Structure and union arguments of all sizes are always passed on the stack. If either the first or second argument (or both) is a structure or union, the corresponding argument register remains unused during the transfer of control from the *caller* to the *callee*.

The Caller

The compilers first determine the number and sizes of the arguments the *caller* is passing. In "C," all routine arguments are **int** or **double** expressions, pointers, or structures. The **int** and pointer arguments are aligned on 32-bit word boundaries; **double** arguments are aligned on 64-bit word boundaries. Structures are aligned on either 32-bit or 64-bit word boundaries depending on whether the structure has any **double** components. To support a variable number of arguments, the arguments are pushed on the stack in right-to-left order as they appear in the call (first argument last).

Working backward, and assuming that the last argument pushed is double-word aligned in memory, the compiler determines the stack offset of each argument that preserves its alignment. This eventually produces the required alignment of the first argument to be pushed. If this differs from the current alignment of **sp**, which the compiler knows by induction, a dummy word is pushed. (More likely, **sp** is decremented by 4.) Each argument is then pushed, preceded by an aligning dummy word if necessary. The *caller* pushes the *callee's* arguments on the stack in right to left order as they appear in the call. If the last two arguments (the left-most ones in the call) are not structures, they are put in their appropriate registers.

The *caller* then pushes its return address and transfers control to the *callee*. Upon return, the permanent registers should all be the same as they were before the call. The *caller* then removes the arguments, and any required alignment words, from the stack.

The Callee

The compiler, assuming that the *caller* has left the last-pushed argument double aligned, knows that the *callee's* **sp** is odd aligned on entry (to account for the saved return address). It saves any required permanent general registers by pushing the highest ones first. The **savew***n* instruction can be used for this purpose. The permanent floating-point registers, if used, must be saved as double precision values in case the *caller* left double precision values in them. Before saving permanent floating-point registers, **sp** must be aligned to a double-word boundary. An appropriate adjustment is made if it is not. The **saved***n* instruction can be used for storing these floating point register values.

Since the compiler knows how many and what type of registers these steps have saved, it can determine the alignment of **sp** at this point. Any stack locals or temporaries can then be properly aligned by pushing dummy words, just as the *caller* did with the arguments. The stack pointer (**sp**) is then left pointing to the last temporary with a known alignment.

On return, the *callee* removes its locals and temporaries by setting **sp** back to the last saved register. The *callee* then restores the saved registers and returns.

Alignment Maintenance

The operating system and runtime startup code align the stack for the first call to the main program, completing the induction in the previous section. The signal handling code in the kernel also aligns the stack before calling a user's signal handler.

If the address is ever taken of an argument that is passed in a register, that argument is permanently moved to a temporary variable at the beginning of the routine.

Profile Conventions

When the compilers are invoked with the **-p** option, they produce code that counts the number of times each routine is called. The user interface is as described in cc(1) and **prof**(1). This section describes the additional code produced by the compilers.

A routine compiled without the **-p** option normally produces an entry point that resembles Figure 1-3.

```
        .text
        .align  2
        .globl  _myrtn
_myrtn:
        first instruction
```

Figure 1-3 Routine Entry Point

When compiled with the -p option, the same routine resembles Figure 1-4.

```
        .data
.Ln:    .long 0
        .text
        .align  2
        .globl  _myrtn
_myrtn:
        loada   .Ln,r2
        call    sp,mcount
        first instruction
```

Figure 1-4 Profiled Routine Entry Point

The *n* in .L*n* is a number chosen by the compiler to produce a local symbol that is unique in the module. The first time **_myrtn** is called, the call to **mcount** sets .L*n* to point to an internal record containing the address of the routine and the number of times it was called. Thereafter, the call to **mcount** uses the pointer in .L*n* to update the calling count.

The profile calling sequence is a specialized version of the standard calling sequence. The address of .L*n* is passed in **r2** to preserve any arguments in **r0** or **r1**. Register **r2** is considered temporary by **mcount**. All other registers are preserved. Since **mcount** does not call any other routines, the usual requirement for 8-byte stack alignment is lifted.

The **mcount** routine can also be called by the **MARK** interface described in **prof**(4). The **MARK**() call is implemented as a series of **asm**() statements, as on the 3B2. Since the **asm**() code cannot know the compiler's register allocation, **r2** is explicitly saved over the call to **mcount**.

System Call Sequences

The system call numbers on the CLIX System are the same as the 3B2 implementation. The CLIPPER call supervisor instruction (**calls**) is used with the system call number as the call supervisor number. The arguments to the call are placed in registers. Upon return from the system call,the carry bit will be set if an error occurred and **r0** will contain the error number. The carry bit will be clear to indicate a successful system call, and **r0** and possibly **r1** will contain the results.

Regions, Segments, and Pages

UNIX System V Release 3.1 has the concepts of *regions* and *segments*. Segments are large, equal-sized divisions of a process's virtual address space. Regions are one or more contiguous segments. They are used for pageable executables and shared libraries. The region/segment concept was straightforward to implement on CLIPPER. Table 1-2 illustrates how the address spaces of the CLIPPER microprocessor and the 3B2 computer are organized.

Table 1-2 Organization of Segments

	CLIPPER	3B2
Address space size	4G byte	2G byte
Low address	0	0x80000000
High Address	0xffffffff	0xffffffff
Page Size	4096 byte	512 byte
Segment Size	4M byte	128K byte
Number of Segments	1024	16K

Common Object File Format (COFF)

CLIX System V object and image files conform to the Common Object File Format (COFF). See "Common Object File Format" in the *UNIX System V Programmer's Guide* for more details.

Headers and Magic Numbers

A COFF file begins with two headers: the file header and the **a.out** header. The **a.out** header is optional, and is only required in executable files. Normally, object (.o) files do not contain the **a.out** header. Each header contains a magic number. The file magic number in the file header identifies the machine for which the file was built. The **a.out** magic number in the **a.out** header identifies a particular type of executable image.

The file magic number supported in the current release of CLIX System V is CLIPPERMAGIC (0577) assigned by AT&T for the CLIPPER. Table 1-3 illustrates how the **a.out** magic numbers are supported.

Table 1-3 **A.out** Magic Numbers

Magic Number	Description
0407	The text region is not write-protected or sharable; the data region is contiguous with the text region.
0410	The text region is write-protected and sharable; the data region starts at the next segment boundary. The entire image is swapped in before execution begins.
0411	Separate instruction and data spaces. Currently reserved by Intergraph.
0413	The text region is write protected and sharable; the data region begins at the next segment boundary. Both are paged from the file system on demand.
0443	Target shared library.

An executable file with magic number 0413 has a noteworthy characteristic. The text region of a program appears in the executable file immediately after the COFF headers. There is no padding after the headers to the next page boundary. Instead, the first file page in which the text image begins is considered the first text page. The first address of the text region (the **.text** section) is bound by *ld*(1) just above the headers. Therefore, part of the executable file header will actually be in memory as part of the executable image.

The data region has a similar characteristic. The data region image begins immediately after the text image. Again, there is no padding to the next page boundary. So, the last page of the text image may have a small piece of the data image, and the first page of the data image may have a small piece of the text image.

In both cases, the extraneous bytes in the page are ignored.

Standard Virtual Addresses

Table 1-4 shows the standard virtual addresses for CLIPPER.

Table 1-4 Standard Virtual Addresses

Address	Description
0x00000000	Text
0x00400000 or next seg	Data, BSS, and Break
0xc0000000	Stack (grows down)
0xc0800000	First Shared Memory Segment
0xec000000 0xec400000 through 0xf1800000 0xf1c00000	private use lib 1 text private use lib 1 data . . . . private use lib 12 text private use lib 12 data
0xf2000000 0xf2400000 through 0xf5800000 0xf5c00000	generic to be defined lib 1 text generic to be defined lib 1 data . . . . generic to be defined lib 8 text generic to be defined lib 8 data
0xf6000000 0xf6400000	generic networking lib text generic networking lib data
0xf6800000 0xf6c00000	generic graphics lib text generic graphics lib data
0xf7000000 0xf7400000	generic screen lib text generic screen lib data
0xf7800000 0xf7c00000	generic user interface lib text generic user interface lib data
0xf8000000 0xf8400000	generic statistics lib text generic statistics lib data
0xf8800000 0xf8c00000	generic database lib text generic database lib data

Table 1-4 Standard Virtual Addresses (continued)

Address	Description
0xf9000000 through 0xfec00000	24 reserved segments
0xff000000 0xff400000	libnsl_s text libnsl_s data
0xff800000 0xffc00000	libc_s text libc_s data

Caching Modes

ld(1) has had options added that allow the user to specify the caching policies for the executable image. The new options are as follows:

-Ct*cm*	Set the cache mode for the text region to *cm*.
-Cd*cm*	Set the cache mode for the data region to *cm*.
-Cs*cm*	Set the cache mode for the stack region to *cm*.

Where *cm* is one of the following:

pw	Private, write through
sw	Shared, write through
cb	Private, copy back
nc	Noncached
df	Default for the region. Defined at kernel configuration time.

The cache mode only applies to the regions in the executable file produced by **ld**(1). The cache mode for regions from shared libraries are established by the shared library.

The last four bytes of the **a.out** header are the text caching mode, the data caching mode, the stack caching mode, and an undefined byte, respectively.

Debugging Tips

A few debugging tips are presented here. This will aid you in porting application programs to the CLIX system. This list is not comprehensive, but includes the most common problems.

Dereferenced NULL (0) Pointers

Null pointers are typically used to indicate undefined or erroneous results. For example, *fopen*(3S) returns NULL when a file could not be opened. However, some programs do not properly check pointers to ensure that they are not NULL (0) and may even attempt to dereference a NULL pointer. The NULL pointer should never be dereferenced. This kind of bug has plagued UNIX System programmers for years.

In the earlier VAX implementations, location 0 contained two 0 bytes; these were the entry mask for the VAX **calls** (call stack) instruction. However, since location 0 contains the beginning of the text region on CLIPPER, there is no assurance that dereferencing the NULL pointer will produce a 0. This is further frustrated because part of the COFF header is loaded with the text image.

The result is that uses of the NULL pointer as strings under earlier systems produced an empty string. However, under the CLIX System, you will get a few extraneous bytes.

We have found the most common offenses to be in calls to **strcat**() and **strcpy**() in **string**(3C). For these and other string routines, an empty string is represented by a valid pointer to a byte containing 0, not a NULL pointer.

Alignment

The CLIPPER C100 microprocessor does not generate data alignment faults (loading a word from an odd address). Instead, addresses are always aligned to the appropriate boundary by masking off low-order bits.

An example of code that might do this is shown in Figure 1-5.

```
char    buf[100];
short   *wp;

wp = (short *) &buf[1];/* cast of odd address */
*wp = 101;             /* use of improperly aligned short ptr */
```

Figure 1-5 Bad Address Pointer Alignment

In the example, **wp**, a pointer to a short, is set to the cast of an odd address of **buf**. The result is that **wp** will not be aligned on a 2-byte boundary. The result is undefined on the CLIPPER.

Look for suspicious casts of pointers. Also look for inconsistencies between formal and actual parameters. **lint**(1) can frequently detect these kinds of bugs.

Byte Order Dependencies

Bytes are ordered with the least significant byte of a multibyte value stored at the lowest address on the CLIPPER microprocessor. Some programs make specific assumptions about the byte ordering. This typically appears in structure or union declarations. Consider the declaration in Figure 1-6.

```
union {
        short   ashort;
        struct {
                char    hibyte, lobyte;
        } as2bytes;
};
```

Figure 1-6 Byte Order Dependencies

The example assumes that the bytes are arranged from high order to low order. Look for suspicious casts of pointers or structures or the use of a structure member name on a nonstructure variable as in Figure 1-7.

```
        short   i;

        i.as2bytes.hibyte = 0;
```

Figure 1-7 Suspicious Member Name

Varargs

One of the hardest problems to spot is improperly implemented variable argument, or VARARGS, routines. Arguments in this kind of routine are not used consistently. Most often, the number and type of the arguments are variable as in **printf**(3S). Sometimes, the number of arguments is not variable but the types change. Usually, the number and/or type of each argument is determined by other arguments.

VARARGS has two typical misuses. The first is not using the VARARGS convention. Beware of a function that assigns the address of one of its arguments to a pointer variable and then attempts pointer arithmetic of any kind (e.g., increment, decrement, and subscript). The second is improper use of the VARARGS convention. An example of this is the **scanf**(3S) routine in the original AT&T release. It assumed that the VARARGS conceptual pointer was implemented as a real pointer that could be explicitly incremented.

The Assembler

This chapter explains how to use the CLIPPER assembler. The CLIPPER assembler is derived from the System V Release 3.0 assembler for the AT&T 3B2. Command-line options, assembler directives, general input syntax, and object file format are substantially the same as the System V Release 3.0 common assembler. Instruction mnemonics and pseudo-ops are based on the *CLIPPER 32-bit Microprocessor User's Manual.*

Assembler Usage

The basic operation of the assembler is described in *as*(1) in the *CLIX Programmer's & User's Reference Manual.*

The **-a***list* option directs the assembler to generate extra alignment checking code for each load and store instruction. *List* can be set to either **f** for floating-point instruction alignment checks, **i** for integer alignment instruction checks, or both. (**-a** is equal to **-afi**.) The alignment checking code assumes that a valid, word-aligned stack addressed by the register **r15** exists. If an alignment error is detected, a "**calls $103**" instruction is executed with the **PC** left pointing to the bad instruction. This instruction normally causes a core dump, which can then be examined with a debugger like **adb**(1).

The -V option of the assembler has been enhanced to print an Intergraph version number in addition to AT&T's version number.

Lexical Conventions

Input to the assembler consists of lines with labels, directives, instructions, pseudo-ops, and comments. The detailed description for each of these assembler constructions uses the notation given in Table 1-5.

Table 1-5 Lexical Conventions

TERM	MEANING
digit	0-9
hexdigit	0-9, a-f
octdigit	0-7
letter	A-Z, a-z, _, .
char	Any ASCII character
ident	Identifier
iexpr	Integer expression
fexpr	Floating point expression
string	Double-quoted character string
bold	The literal text *bold*
[...]	... is optional
{...}	Zero or more occurrences of ...

Identifiers

Identifiers begin with a *letter* and continue with *letters* or *digits*. Identifiers may have any length but are unique only in the first 1023 characters. Character case is honored.

Examples:

```
ident   _main   .L23     long_nam
IdEnT   ___exit L00001   _really_very_long_name
```

Constants

Numeric constants are specified as shown in Table 1-6. The letters shown may be upper or lower case. A string may not include a double quote or a new line, nor may it exceed 30 characters.

Table 1-6 Numeric Constants

decimal integer	1-9 { *digit* }
hex integer	0x { *hexdigit* }
octal integer	0 { *octdigit* }
single floating	0f { *digit* } [. { *digit* }] [e[±] { *digit* }]
double floating	0d { *digit* } [. { *digit* }] [e[±] { *digit* }]
string	" { *char* } "

Labels

A label consists of an identifier followed by a colon (:).

Comments

Comments begin with a sharp character (#) and terminate with the end-of-line.

Statements

Statements consist of an optional series of labels, followed by an assembler directive, machine instruction, or pseudo-op, followed by an optional comment.

Expressions

The operators available for arithmetic expressions are multiply (*), divide (/), add (+), and unary and binary minus (−). The multiplication and division operators are higher in precedence than addition and negation operators. Expressions can be built up from constants, identifiers, and parentheses using these operators, but they must generally be absolute or relative to the origin of a segment. For example, if *_main* is a symbol in the .text segment, *_main−_main* is legal because it is absolute, but *_main+_main* is not.

Directives

The assembler accepts several directives to control data, segments, and symbols as listed in the following sections.

Section Directives

.text
Places all following code up to the next **.text**, **.data**, **.section**, or **.previous** directive into the text segment.

.data
Places all following code up to the next **.text**, **.data**, **.section**, or **.previous** directive into the data segment.

.previous
Places all following code up to the next **.text**, **.data**, **.section**, or **.previous** directive into the section that was in force immediately preceding the current one.

.section *ident*, *string*
Places all following code up to the next **.text**, **.data**, **.section**, or **.previous** directive into the section with the name *ident*. The *string* may contain one or more of the letters found in Table 1-7. These letters select the various section flags in the COFF section header. See "Common Object File Format" in the *UNIX Sys-*

tem V Programmer's Guide for more details.

Table 1-7 *String* Section Flags

Letter	Flag	Meaning
b	STYP_BSS	Uninitialized data
c	STYP_COPY	Copy section
d	STYP_DSECT	Dummy section
i	STYP_INFO	Comment section
l	STYP_LIB	Like STYP_INFO, but for **.lib** section
n	STYP_NOLOAD	No-load section
o	STYP_OVERLAY	Overlay section
w	STYP_DATA	Initialized data
x	STYP_TEXT	Executable text

Storage Directives

.align *iexpr*

Aligns the following text or data so that it begins on an *iexpr*-byte boundary. (*iexpr* must be between 1 and 1024, inclusive.) For the **.align** directive to work, the module's section containing the **.align** must begin on a byte boundary at least as constrained as the most constrained **.align** directive in the module. The linker, **ld**(1), provides control over this. By default, **as**(1) and **ld**(1) support *iexpr* set to 1, 2, 4, or 8. When branch optimization is enabled, **.align** directives in the **.text** section support only *iexpr* set to 1 or 2.

.space *iexpr*

Allocates *iexpr* bytes of storage initialized to zero. *Iexpr* must be greater than zero.

.byte [*iexpr1*:]*iexpr2*{,[*iexpr1*:]*iexpr2*}

If *iexpr1* is not specified, allocates a byte for each *iexpr2* with *iexpr2* as its value. If *iexpr1* is specified, allocates an *iexpr1*-bit field within a byte with *iexpr2* as its value. Subsequent fields will then fill the rest of the byte. Fields may not cross a byte boundary. The last partial byte, if any, is filled with zeros. Fields are allocated starting from bit 0, the least significant bit.

.word [*iexpr1*:]*iexpr2*{,[*iexpr1*:]*iexpr2*}
If *iexpr1* is not specified, allocates a 16-bit "word" (CLIPPER halfword) for each *iexpr2* with *iexpr2* as its value. If *iexpr1* is specified, allocates an *iexpr1*-bit field within a word with *iexpr2* as its value. Subsequent fields will then fill the rest of the word. Fields may not cross a word boundary. The last partial word, if any, is filled with zeros. Fields are allocated starting from bit 0, the least significant bit.

.long [*iexpr1*:]*iexpr2*{,[*iexpr1*:]*iexpr2*}
If *iexpr1* is not specified, allocates a 32-bit "long" (CLIPPER word) for each *iexpr2* with *iexpr2* as its value. If *iexpr1* is specified, allocates an *iexpr1*-bit field within a long with *iexpr2* as its value. Subsequent fields will then fill the rest of the long. Fields may not cross a long boundary. The last partial long, if any, is filled with zeros. Fields are allocated starting from bit 0, the least significant bit.

.float *fexpr* {,*fexpr*}
Allocates a 32-bit word for each single precision floating-point *fexpr* with *fexpr* as its value.

.double *fexpr* {,*fexpr*}
Allocates a 64-bit double word for each double-precision floating-point *fexpr* with *fexpr* as its value.

.comm *ident*,*iexpr*
Declares the symbol *ident* as an undefined external with value *iexpr*. If another module does not explicitly define *ident*, the linker allocates an *iexpr*-byte block in the **.bss** segment called *ident*. This is ordinary FORTRAN-style COMMON storage and is used by "C" for most global variables.

.bss *ident*,*iexpr1*,*iexpr2*
Allocates *iexpr1* bytes in the **.bss** area aligned on the next *iexpr2*-byte boundary with the name *ident*. Both *iexprs* must be positive and absolute, and *ident* must not already be defined.

Control Directives

.ident *string*
: The *string* is appended to the **.comment** section (created if necessary). The **.comment** section is preserved by **ld**(1), but not loaded by the kernel.

.version *string*
: The *string* is compared with the string "02.01," the AT&T version number assigned to the assembler. An error results if *string* is greater than the current assembler version number.

Symbol Directives

There are two kinds of symbol directives. The *basic* symbol directives are used in most modules even when the **-g** option is not given to **cc**(1). These are as follows:

.globl *ident*
: Declares *ident* as a global symbol. If the current module defines *ident*, *ident* becomes a global definition. Otherwise, it becomes an external request.

.set *ident*, *iexpr*
: Defines or redefines *ident* with the value *iexpr*. *Ident* must be non-null.

.file *string*
: Declares the original high-level language source file name to be *string*. The *string* must not exceed 14 characters. This directive is used in assembler error messages or by the debugger.

The *debugging* symbol directives are used only by high-level language debuggers. The directives in this section only apply when the compiler is asked to generate extra debugging information with the **-g** command-line option or when an assembly language programmer wishes to provide this information manually.

.ln *iexpr1* [, *iexpr2*]
: Declares *iexpr2* to be the address of the first instruction for statement number *iexpr1* in the current procedure. The program counter is used if *iexpr2* is not specified. By convention, *iexpr1* has the value 1 for the line containing the opening brace

of a procedure and increases by 1 for each source line. The *.ln* directives are only generated for lines that contain code; lines containing only declarations are not assigned line numbers. Line numbers given with *.ln* restart at 1 with each procedure so they do not completely specify the line number within the source file. The missing information, the absolute number of the line containing the opening brace of a function, is specified in the definition of the **.bf** symbol for each procedure. (See "Symbol Table Entry Directives" below and "Common Object File Format" in the *UNIX System V Programmer's Guide*.)

Symbol Table Entry Directives

Most of the debugging symbol information in the Common Object File Format (COFF) is stored in complex symbol table entries. Each entry is declared with a rigid sequence of directives that set each field in the entry and any necessary auxiliary entries. Each symbol table entry declaration begins with a **.def** directive and ends with a **.endef** directive. Between these directives, other directives are given. Each directive defines a single field in the symbol table entry structure. The **.def** directive is followed by the **.val** and **.scl** directives to define the value and the storage class of the symbol. Depending on the symbol's storage class and type, different directives may then follow before the final **.endef**.

For better readability, the compilers put all directives that define a single symbol on the same line and separate them by semicolons.

.def *ident*
: Begins the definition of the symbol *ident*.

.val *iexpr*
: Declares the value and section number of the current symbol by setting the *n_value* field of the symbol table entry to *iexpr* and *n_scnum* to the section number to which the symbol refers. For instance, the symbol *_main*, normally in the text segment, is a section 1 symbol. An absolute symbol value is specified as section -1.

.scl *iexpr*
: Declares the storage class of the current symbol by setting the *n_sclass* field of the symbol table entry to *iexpr*. This value also determines which, if any, of the following directives are applicable.

.type *iexpr*
Declares the type of the current symbol by setting the *n_type* field of the symbol table entry to *iexpr*.

.line *iexpr*
Declares the line number of the current symbol by setting the *x_lnno* field of the auxiliary symbol table entry to *iexpr*. For example, the line number associated with the **.bf** symbol is the number of the line containing the opening brace of the current procedure.

.size *iexpr*
Declares the size of the object to which the current symbol refers by setting the *x_size* field of the auxiliary symbol table entry to *iexpr*.

.tag *ident*
Declares the tag index of the current symbol by setting the *x_tagndx* field of the auxiliary symbol table entry to *ident*.

.dim *iexpr* { , *iexpr* }
Declares the dimensions of an array symbol by setting as many elements of the *x_dimen* field of the auxiliary symbol table entry as there are *iexpr*s. At most, four *iexpr*s are allowed.

.endef
Completes the declaration of the current symbol.

Machine Instructions

Table 1-8 defines the operand notation used to describe the assembler instruction mnemonics.

Table 1-8 Assembler Operand Notation

Operand Type and Size	
b	byte (8-bit integer)
h	halfword (16-bit integer)
w	word (32-bit integer)
l	longword (64-bit integer)
s	single floating (32-bit)
d	double floating (64-bit)
p	processor register (32-bit)
Operand Location and Syntax	
q	quick (*$value*)
i	immediate (*$value*)
b	byte (*$value*)
1,2	register (*r0-r15, f0-f7, psw, ssw, sswf*)
a	address (one of the addressing modes)
Addressing Modes	
n	absolute address
[r*x*](r*n*)	relative indexed
[r*x*](pc)	PC indexed
n(r*n*)	relative or relative + displacement
n(pc)	PC relative
expr	depends on relocatability

Each instruction operand is described by a two-letter code. The left letter of the operand code specifies the operand's type and size. The right letter of the code specifies the operand's field within the instruction and its location in the machine (immediate value, register, memory, etc.). The right letter also specifies the operand's syntax.

For example, the operand code **w1** indicates a word operand in the general register whose number is encoded in the *R1* field of the instruction. An example might be **r7**. The code **sa** indicates a single floating operand in the memory location whose address is given by one of the addressing modes shown in Table 1-8. An example might be **_coefficients(r3)**. Quick and immediate operand types are always **w** because these directly encoded values are always zero or sign-extended to a word by the hardware before use.

Note: **fp** and **sp** are synonyms for **r14** and **r15**, respectively.

Table 1-9 Instruction Formats

Op Code	Operands	Op Code	Operands
addd	*d1,d2*	divd	*d1,d2*
addi	*wi,w2*	divs	*s1,s2*
addq	*wq,w2*	divw	*w1,w2*
adds	*s1,s2*	divwu	*w1,w2*
addw	*w1,w2*	initc	
addwc	*w1,w2*	loada	*ba,w2*
andi	*wi,w2*	loadb	*ba,w2*
andw	*w1,w2*	loadbu	*ba,w2*
b*	*ha*	loadd	*da,d2*
bf*	*ha*	loadfs	*w1,d2*
call	*w2,ha*	loadh	*ha,w2*
callm	*bb*[,*w1*[,*w2*]]	loadhu	*ha,w2*
callmp	*bb*[,*w1*[,*w2*]]	loadi	*wi,w2*
calls	*bb*	loadq	*wq,w2*
cmpc		loads	*sa,s2*
cmpd	*d1,d2*	loadw	*wa,w2*
cmpi	*wi,w2*	modw	*w1,w2*
cmpq	*wq,w2*	modwu	*w1,w2*
cmps	*s1,s2*	movc	
cmpw	*w1,w2*	movd	*d1,d2*
cnvds	*d1,s2*	movdl	*d1,l2*
cnvdw	*d1,w2*	movld	*l1,d2*
cnvrdw	*d1,w2*	movpw	*p1,w2*
cnvrsw	*s1,w2*	movs	*s1,s2*
cnvsd	*s1,d2*	movsu	*w1,w2*
cnvsw	*s1,w2*	movsw	*s1,w2*
cnvtdw	*d1,w2*	movus	*w1,w2*
cnvtsw	*s1,w2*	movw	*w1,w2*
cnvwd	*w1,d2*	movwp	*w2,p1*
cnvws	*w1,s2*	movws	*w1,s2*

Table 1-9 Instruction Formats (continued)

Op Code	Operands	Op Code	Operands
muld	*d1,d2*	savew*n*	
muls	*s1,s2*	scalbd	*w1,d2*
mulw	*w1,w2*	scalbs	*w1,s2*
mulwu	*w1,w2*	shai	*wi,w2*
mulwux	*w1,l2*	shal	*w1,l2*
mulwx	*w1,l2*	shali	*wi,l2*
negd	*d1,d2*	shaw	*w1,w2*
negs	*s1,s2*	shli	*wi,w2*
negw	*w1,w2*	shll	*w1,l2*
noop	*bb*	shlli	*wi,l2*
notq	*wq,w2*	shlw	*w1,w2*
notw	*w1,w2*	storb	*w2,ba*
ori	*wi,w2*	stord	*d2,da*
orw	*w1,w2*	storh	*w2,ha*
popw	*w1,w2*	stors	*s2,sa*
pushw	*w2,w1*	storw	*w2,wa*
restd*n*		subd	*d1,d2*
restur	*w1*	subi	*wi,w2*
restw*n*		subq	*wq,w2*
ret	*w2*	subs	*s1,s2*
reti	*w1*	subw	*w1,w2*
roti	*wi,w2*	subwc	*w1,w2*
rotl	*w1,l2*	trapfn	
rotli	*wi,l2*	tsts	*wa,w2*
rotw	*w1,w2*	wait	
saved*n*		xori	*wi,w2*
saveur	*w1*	xorw	*w1,w2*

Pseudo-ops

For convenience, the assembler also accepts some additional "instruction" mnemonics called *pseudo-ops*. A pseudo-op accepts several types of operands and selects the appropriate machine instruction based on these types. Table 1-10 shows all pseudo-op argument combinations and the instructions they select.

Table 1-10 Pseudo-ops

Pseudo-op		Instruction	
	r1,r2	addw	*r1,r2*
addw	*wq,r2*	addq	*wq,r2*
	wi,r2	addi	*wi,r2*
	r1,r2	andw	*r1,r2*
andw	*wq,r2*	andi	*wq,r2*
	wi,r2	andi	*wi,r2*
	r1,r2	cmpw	*r1,r2*
cmpw	*wq,r2*	cmpq	*wq,r2*
	wi,r2	cmpi	*wi,r2*
movaw	*ba,w2*	loada	*ba,w2*
	d1,d2	movd	*d1,d2*
movd	*da,d2*	loadd	*da,d2*
	d2,da	stord	*d2,da*
	s1,s2	movs	*s1,s2*
movs	*sa,s2*	loads	*sa,s2*
	s2,sa	stors	*s2,sa*
movbw	*ba,w2*	loadb	*ba,w2*
movbwu	*ba,w2*	loadbu	*ba,w2*
movhw	*ha,w2*	loadh	*ha,w2*
movhwu	*ha,w2*	loadw	*ha,w2*
	w1,w2	movw	*w1,w2*
	wq,w2	loadq	*wq,w2*
movw	*wi,w2*	notq	~wi, w2
		loadi	*wi,w2*
	wa,w2	loadw	*wa,w2*
	w2,wa	storw	*wa,w2*
movwb	*w2,ba*	storb	*w2,ba*
movwh	*w2,ha*	storh	*w2,ha*
notw	*r1,r2*	notw	*r1,r2*
	wq,r2	notq	*wq,r2*

Table 1-10 Pseudo-ops (continued)

Pseudo-op		Instruction	
orw	*r1,r2*	orw	*r1,r2*
	wq,r2	ori	*wq,r2*
	wi,r2	ori	*wi,r2*
restw13		popw	*r13* **and**
		popw	*r14*
restw14		popw	*r14*
rotl	*w1,l2*	rotl	*w1,l2*
	wq,l2	rotli	*wq,l2*
	wi,l2	rotli	*wi,l2*
rotw	*w1,w2*	rotw	*w1,w2*
	wq,w2	roti	*wq,w2*
	wi,w2	roti	*wi,w2*
savew13		pushw	*r14* **and**
		pushw	*r13*
savew14		pushw	*r14*
shal	*w1,l2*	shal	*w1,l2*
	wq,l2	shali	*wq,l2*
	wi,l2	shali	*wi,l2*
shaw	*w1,w2*	shaw	*w1,w2*
	wq,w2	shai	*wq,w2*
	wi,w2	shai	*wi,w2*
shll	*w1,l2*	shll	*w1,l2*
	wq,l2	shlli	*wq,l2*
	wi,l2	shlli	*wi,l2*
shlw	*w1,w2*	shlw	*w1,w2*
	wq,w2	shli	*wq,w2*
	wi,w2	shli	*wi,w2*
subw	*r1,r2*	subw	*r1,r2*
	wq,r2	subq	*wq,r2*
	wi,r2	subi	*wi,r2*
xorw	*r1,r2*	xorw	*r1,r2*
	wq,r2	xori	*wq,r2*
	wi,r2	xori	*wi,r2*

2. PROC Debugging Tutorial

Chapter 2: PROC Debugging Tutorial

Introduction

The **/proc** file system is a simple and efficient means for inspecting and modifying running processes. Members of this file system are processes whose address space may be read and written using the standard file manipulation system calls **lseek**(2), **read**(2), and **write**(2). In addition, the **ioctl**(2) system call provides features such as stopping and starting a process, posting signals to a process, and monitoring system call use.

The familiarity of the file system interface coupled with the availability of features not found in the current process control mechanism, **ptrace**(2), make **/proc** the preferred implementation for current and future process debugging software.

Comparison of /proc and ptrace(2)

The **/proc** file system was designed to eliminate some problems associated with the current process control mechanism, **ptrace**(2). This section describes the problems with **ptrace**(2) and provides an overview of **/proc** debugging.

The **ptrace**(2) system call requires coordination between the process being traced and the process performing the tracing and is restricted to parent-child process relationships. The coordination between the two processes is an explicit request by the child to be traced and requires that the child process be in the stopped state every time the parent process issues a tracing command.

The tracing features of the **/proc** file system require no coordination between the process performing the trace and the process being traced. The only restriction on the two processes is that they must be owned by the same user ID. A trace command can be issued any time for a process opened under the **/proc** file system. The recipient acts on the command either immediately or during the next system call entry or exit. The flexibility of the **/proc** file interface enables tracing forked processes and processes that were created before the tracing process.

Using the **/proc** file debugging is more efficient than using **ptrace**(2). The **ptrace**(2) call transfers a maximum of one word of data per call to or from the traced process's address space while **/proc** will allow entire regions of a traced process's address space to be transferred in a single **read**(2) or **write**(2) system call.

The **/proc** file capabilities not found in **ptrace**(2) include the ability to read the kernel structure **proc**, stop a process on system call entry or exit, and read and write a process's register set in a single system call. All of these features are implemented through the **ioctl**(2) system call.

File System Differences

Although /**proc** appears as a normal file system to commands such as **mount**(1M) and **df**(1M), it differs from other file system types in two ways. First, each operating system can have only one /**proc** file system. Secondly, no device is associated with a /**proc** file system. A device is specified at mount time. However, it is specified only to satisfy the requirements of **mount**(1M). It is never read or written. The second difference implies that commands such as **fsck**(1M) and **fsdb**(1M) are unnecessary and undefined for /**proc** file systems.

File System Similarities

To use the /proc file feature of the kernel, the following command must be issued at some point before the first use. (An appropriate time would be during system initialization.)

mount -f PROC /dev/proc /proc

Once mounted, common file operations may be executed. For instance, a directory listing will resemble the following:

```
$ ls -ls /proc
total 10852
   0 -rw-------   1 root    root          0 Oct 10 13:57 00000
 145 -rw-------   1 root    root      72416 Oct 10 13:57 00001
   0 -rw-------   1 root    root          0 Oct 10 13:57 00002
   0 -rw-------   1 root    root          0 Oct 10 13:57 00003
   0 -rw-------   1 root    root          0 Oct 10 13:57 00004
 180 -rw-------   1 root    root      90520 Oct 10 13:57 00017
 454 -rw-------   1 terri   sys      229704 Oct 10 13:57 00018
 214 -rw-------   1 root    root     108008 Oct 10 13:57 00070
1018 -rw-------   1 root    root     516352 Oct 10 13:57 00073
1475 -rw-------   1 root    root     748360 Oct 10 13:57 00089
1475 -rw-------   1 root    root     748360 Oct 10 13:57 00090
1475 -rw-------   1 root    root     748360 Oct 10 13:57 00091
1009 -rw-------   1 root    root     511656 Oct 10 13:57 00114
1290 -rw-------   1 root    root     654656 Oct 10 13:57 00115
 158 -rw-------   1 root    root      78864 Oct 10 13:57 00118
1027 -rw-------   1 root    root     520832 Oct 10 13:57 00123
 180 -rw-------   1 root    root      90520 Oct 10 13:57 03933
 180 -rw-------   1 root    root      90520 Oct 10 13:57 06155
 366 -rw-------   1 terri   sys      185240 Oct 10 13:57 06441
 206 -rw-------   1 terri   sys      103560 Oct 10 13:57 06462
$
```

The display shows that the processes currently active on the system appear as files whose names correspond to their process identification numbers. The file sizes match the current size in bytes of the process's defined virtual address space; the owner and group are taken directly from the process;

and the protections allow the owner read and write access. The file dates reflect the time when the listing was generated and indicate that the information presented was valid at that time. (A **/proc** directory listing is a snapshot of a running system similar to that provided by **ps**(1).)

Other commands may be applied to the process files. For instance, **dd**(1) can be used to create a copy of a process's memory image. However, when attempting an operation like this, the user must be aware of possible discontinuities in the process's address space and map around them accordingly.

System Call Interface

Once the **/proc** file system is mounted, processes may be monitored and manipulated through the system calls **lseek**(2), **read**(2), **write**(2), and **ioctl**(2). Each call requires a file descriptor returned from a successful **open**(2). The **lseek**(2) system call positions the file pointer on a valid virtual address for **read**(2) and **write**(2). By default, a process's data and stack segments are read/write, while the text segment is typically read-only. For this reason, write attempts to the text segment may fail. However, in most cases it is possible to alter the read-only status of the text segment through the PIOEXCLU **ioctl**(2) command. (See the description below.)

The following **ioctl**(2) commands are supported for **/proc** files:

PIOCGETPR	Fetch the process's **proc** structure from the kernel process table.
PIOCSTOP	Send a SIGSTOP signal to the process and wait for it to enter the stopped state.
PIOCWSTOP	Wait for the process to enter the stopped state.
PIOCRUN	Make the process runnable after a stop.
PIOCSMASK	Specify a set of signals whose reception by the process causes the process to enter the stopped state. A mask of zeroes turns off the trace. Note that the trace state is retained even after the process that requested the trace closes the **/proc** file.
PIOCCSIG	Clear all pending signals to a process.
PIOCEXCLU	Mark the process's text segment as nonshared so that subsequent writes will succeed.
PIOCOPENT	Return a file descriptor that can be used to read the file containing the text and data segments of the process.
PIOCSTR	Set the trace flag in the process's PSW register, resulting in a trace trap after the process executes its next instruction.
PIOCRREGS	Read the process registers, including the general and floating registers and the PSW, SSW, and PC.

PIOCWREGS	Write the process registers, including the general and floating registers and the PSW, SSW, and PC.
PIOCSEXIT	Establish a system call for the process that, when exited, causes the process to stop.
PIOCSENTR	Establish a system call for the process that, when entered, causes the process to stop.

Refer to **proc**(7S) for a description of the types and formats of the arguments for these commands.

EXAMPLE

The following example employs many features described. The example is designed to monitor and log all **open**(2) system calls a user-specified program makes.

```
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/immu.h>
#include        <sys/param.h>
#include        <sys/region.h>
#include        <sys/proc.h>
#include        <sys/pioctl.h>
#include        <sys/errno.h>

#define OPEN_SYSCALL    5                  /* # of open system call */

void    print_open_args();

main(argc, argv)
int     argc;
char    **argv;
{
        int     pid;
        int     procfd;
        char    procname[80];
        int     syscall = OPEN_SYSCALL; /* ioctl needs address of syscall # */
        int     whystop, whatstop;      /* status info returned by PIOCWSTOP */

        if (argc < 2)  {
                fprintf(stderr, "usage:  %s a.out [arg1 ... argn]\n", argv[0]);
                exit(1);
```

continued

```
}
if ((pid = fork()) == -1)  {
        fprintf(stderr, "%s : fork error\n", argv[0]);
        exit(1);
}
/*
 * parent process will open child's proc entry and set up
 * "stop on entry" for open syscall
 */
if (pid)  {
        sprintf(procname, "/proc/%05d", pid);
        if ((procfd = open(procname, 0)) == -1)  {
                fprintf(stderr, "%s : cannot open %s\n",
                        argv[0], procname);
                exit(1);
        }
        if (ioctl(procfd, PIOCSENTR, &syscall) == -1)  {
                fprintf(stderr, "%s : PIOCSENTER error\n", argv[0]);
                exit(1);
        }
} else  {
        sleep(2);       /* give the parent a chance to set up */
        if (execv(argv[1], &argv[1]) == -1)  {
                fprintf(stderr, "%s : error exec-ing %s\n",
                        argv[1], argv[1]);
                _exit(1);
        }
}
/*
 * Parent will monitor child until child exits or error occurs
 */
for (;;)  {
        if (ioctl(procfd, PIOCRUN, 0) == -1)  {
                fprintf(stderr, "%s : run error\n", argv[0]);
                exit(1);
        }
        switch (whystop = ioctl(procfd, PIOCWSTOP, &whatstop))  {
        case SYSENTRY:
                if (whatstop != OPEN_SYSCALL)  {
                        fprintf(stderr,
                                "%s : stopped on wrong syscall\n",
                                argv[0]);
                        exit(1);
                }
```

continued

```
                        print_open_args(argv[0], procfd);
                        break;
                default:
                        if (kill(pid, 0) == -1 && errno == ESRCH)  {
                                fprintf(stderr, "%s : process exited\n",
                                        argv[0]);
                                exit(0);
                        }
                case SYSEXIT:
                case REQUESTED:
                case SIGNALLED:
                        fprintf(stderr,
                                "%s : unexpected PIOCWSTOP return (%d %d)\n",
                                argv[0], whystop, whatstop);
                        exit(1);
                }
        }
}


/*
 * This routine prints out the arguments to the open system call. On
 * the Clipper, arguments 1 and 2 will be in registers r0 and r1.
 */

void
print_open_args(progname, procfd)
char    *progname;
int     procfd;
{
        char    filename[80];
        int     open_flag;
        struct  {
                int     gen_regs[16];
                double  float_regs[8];
                int     cpu_regs[3];
        } regs;

        if (ioctl(procfd, PIOCRREGS, &regs) == -1)  {
                fprintf(stderr, "%s : error reading process registers\n",
                        progname);
                return;
        }
        /*
```

continued

```
          * seek to location of open's argument #1 and read it
          * NOTE: The seek will need to be broken into multiple seeks if
          *       open's arg #1 is at an address with high bit set !
          */
        if (lseek(procfd, regs.gen_regs[0], 0) == -1 ||
                        read(procfd, filename, 80) == -1)  {
                fprintf(stderr, "%s : error reading argument 0 of open\n",
                        progname);
                return;
        }
                /* open's argument #2 is an immediate value in register r1 */
        fprintf(stdout, "%s : open arguments - %s, %d\n",
                progname, filename, regs.gen_regs[1]);
}
```

Chapter 3: Network Programming Tutorial

Introduction

The "Network Programming Tutorial" provides a background essential to understanding the various network programming libraries associated with the Xerox Network Services Protocol (XNS). In this tutorial, *application, application program,* and *process* are used interchangeably. Also, *network* and *communications* are used interchangeably.

This tutorial also provides an overview of how the Intergraph network is started and how the local network setup can be customized by modifying the **/etc/incd.conf** configuration file.

This tutorial does not attempt to define all communication systems or architectures available. It assumes that an underlying communication system is present and does not attempt to describe how it works.

Definitions

This section introduces some basic terminology common to many networking systems.

Network Architecture

Networking systems, much like computer operating systems, are designed and constructed in functional layers. These layers create the *network architecture*. Each layer performs a specific set of functions and services. Together, the layers interact to provide total end-to-end network operation.

The network architecture created by most vendors conforms to the International Standards Organization's model for Open Systems Interconnection (ISO/OSI). Figure 3-1 includes the ISO/OSI model and a number of vendor architectures to show how they compare.

	ISO	DECnet	SNA	XNS	TCP/IP
7	File Transfer Virtual Terminal Job Management	DAP NCP SET HOST MSCP Mail LAT	3270 RJE 3770 Mail DISSOS	File Transfer Clearinghouse Printing Protocol	File Transfer Virtual Terminal Mail
6	Presentation	System	Function Management	Courier	System
5	Session	System	Data Flow Control	Courier	System
4	Transport	NSP	Transmission Control	Internet Transport Protocol ITP	Transmission Control Protocol TCP
3	Internetwork Protocol X.25 services	Routing	Path Control	Internet Datagram Protocol IDP	Internet Protocol IP
2	HDLC IEEE 802.3,4,5 X.25	HDLC, DDCMP IEEE 802.3,4,5 X.25	SDLC X.25	HDLC IEEE 802.3,4,5 X.25	HDLC IEEE 802.3,4,5 X.25
1	IEEE 802.3,4,5 RS232, V.35, X.21	IEEE 802.3,4,5 RS232, V.35, X.21	IBM® Specific RS232 V.35	IEEE 802.3,4,5 RS232, V.35, X.21	IEEE 802.3,4,5 RS232, V.35, X.21

Figure 3-1: Network Architecture - Functional Layers

Client/Server

At least two communicating processes, the *client* and the *server*, are in any communication system. The distinction between the two is not always important. The client process initiates communication and then communicates with the server process.

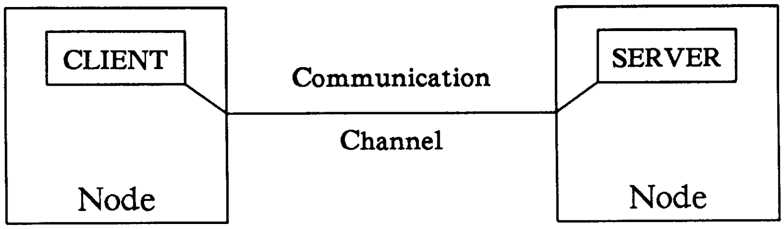


Figure 3-2: Client/Server Model

Connection/Connectionless

A client process can communicate with a server process by using either *connection-oriented* or *connectionless* methods. If connection-oriented methods are used, the client must create a connection, transfer data, and then close the connection. Connection-oriented data transfers are guaranteed. (Once the connection has been established, no data will be lost.) However, the process of establishing the connection constitutes a heavy system/network penalty. If connectionless methods are used, the client transmits the data to the server. This method has the least overhead, but does not guarantee the delivery of the data transmitted.

Full/Half Duplex

Any communication session between a client and a server can be either *full-duplex* or *half-duplex* (simplex). Either party can transmit any time (even at the same time) with a full-duplex conversation. Alternating communication, the client and server track which host is allowed to talk next in a half-duplex conversation.

Addressing

Any network has many ***addressable*** processes or entities. An address is an identifier (usually a number or ASCII string), specific to the network being used, that uniquely defines an entity in the network. On every Intergraph machine, addresses have the form [*net*.]*host*, where *net* represents the network address associated with an entire Local Area Network (LAN) and *host* represents the physical Ethernet address of the node within the LAN. An example of an address would be 13498.08-00-36-41-00-00. This address represents a node on LAN 13498 with a host address of 08-00-36-41-00-00. Within the Intergraph XNS network, one further identifier called *server number* is needed to identify a process/server within the node. This value is associated with image names in the file named **/usr/ip32/inc/server.dat**.

Intergraph Communication Environment

In the previous section, some concepts were defined without actual routines being mentioned. The remainder of this section is dedicated to describing networking modules and routines from a programmer's point of view.

The communication library provided with the Intergraph Network Core (INC) product contains a number of functionally-separate modules. These modules are FMU (File Management Utility), which provides a basic callable interface to transfer files; SNI (Simple Network Interface), which provides a general-purpose network interface; and CLH (Clearinghouse), which provides a network-wide clearinghouse function.

All calls return pointers to characters. If the operation was successful, the call will return 0. If the operation was unsuccessful, the call will return a pointer to an error message.

Figure 3-3 shows the modules and their relationships to each other.

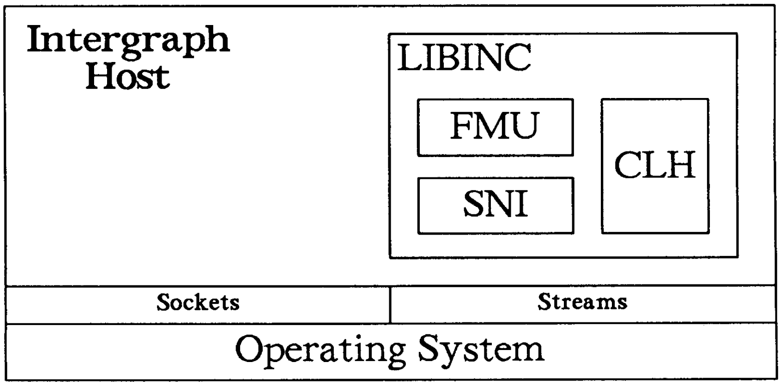


Figure 3-3: LIBINC Modules and Relationships

When linking a program to the **LIBINC** library, use the following syntax on the final link command:

cc [*option*] ... *file* ... **-linc -lnsl_s**

FMU - File Management Utility

The File Management Utility (FMU) interface allows files to be copied from a remote to a local system or from a local to a remote system. For example, to copy a file stored on a remote Intergraph VMS or CLIX host, the FMU subroutines can be used.

The **fmu**(1) application program uses the FMU subroutines to perform its various functions, such as the **send** command. For example, to transfer a copy of the program **dog.c** to another system and call the transferred file **cat.c**, use the following FMU command:

> **fmu** *host.username.password* **send dog.c cat.c**

The FMU subroutine corresponding to this command is as follows:

```
fmu_send("dog.c", "cat.c")
```

The **fmu**(1) **send** command and **fmu_send**(3N) subroutine have similar formats. The FMU calls currently supported by Intergraph follow. For more information on these calls, see section (3N) of the *CLIX Programmer's & User's Reference Manual.*

Call	Function
fmu_connect(node)	Connects to the Intergraph FMU server.
fmu_receive(srcfile, dstfile)	Receives a file from a remote system.
fmu_send(srcfile, dstfile)	Sends a file to a remote system.
fmu_rcmd(command)	Executes a remote command.
fmu_set(mode)	Sets a mode within FMU.
fmu_disconnect()	Disconnects from the server.

SNI - Simple Network Interface

The Simple Network Interface (SNI) subroutines allow the implemention of a simple client/server pair. Once a client and server are implemented, programs can be run on a remote system to communicate with the remote node. In other words, if Intergraph does not supply a service needed, SNI allows service to be implemented. If desired, more than one client/server connection can be opened at a time.

Currently, SNI is implemented on Xerox Network System (XNS) networks only. SNI is general enough to fit the needs of most application programmers. For instance, SNI is the underlying service used in Intergraph's File Management Utility (FMU) program.

A list of the SNI calls follows:

Call	Function
sni_accept(sd)	Accepts a connection from a client.
sni_connect(sd, node, sernum, server)	Initiates a connection to a server, and begins execution of the program.
sni_rxw(sd,buffer, len,timeout)	Receives transmitted data to a buffer.
sni_txw(sd,buffer, len, timeout)	Transmits data across a connection.
sni_close(sd)	Disconnects a connection.

When a client/server pair is run using SNI, the client calls **sni_connect**(3N) to initiate the server program. The server would then call **sni_accept**(3N) to accept the connection. Once the connection is established, the client and server can transmit and receive data across the connection using **sni_rxw**(3N) and **sni_txw**(3N). When the tasks are completed, both the client and server should call **sni_close**(3N) to end the connection.

The Server.dat File

server.dat(4) is a text file containing information about servers on the local node. The *sernum* and *server* arguments to **sni_connect**(3N) are used to access the proper server entry in **server.dat**(4). This file is in the **/usr/ip32/inc** directory.

The client initiates the link using the **sni_connect**(3N) call. The **xns_listener**(1M) on the same node as the server picks up the request and looks up information on the server using that node's **server.dat**(4) file. Next, the **xns_listener**(1M) starts the server, and the server can accept the link to the client using the **sni_accept**(3N) call. Once the connection is established, the client and server can communicate with each other across the network.

If the **xns_listener**(1M) cannot start the server, the **xns_listener**(1M) tells the client and the **sni_connect**(3N) call returns an error. This could happen if the server is not listed in that node's **server.dat**(4) file or if the client does not have sufficient privileges to use the server.

Four possible fields can be filled in the **server.dat**(4) file for each server. The fields are formatted in **server.dat**(4) as follows where exclamation points are the required delimiters:

![*server_no*]**!**[*flag*...]**!**[*server_file*]**!**[*default_username*]**!**

server_no is the number assigned to the server in **server.dat**(4). The numbers in the range of 1000-32767 may be used. Numbers in the range of 0-999 are reserved by Intergraph. A *server_no* is passed to the **sni_connect**(3N) call using the *sernum* argument.

The *flag* field in **server.dat**(4) is used with the *default_username* field to control access to the server and the context under which it is run. A list of the parameters specified in the *flag* field follows.

U	User name is required.
P	Password is required.
D	*default_username*, rather than the login account, is used.
N	Null passwords are not allowed.

The **U**, **P**, and **N** parameters refer to the argument *node* passed to the **sni_connect**(3N) call. If **U** is present, the user name needs to be passed in the *node* argument. If **P** is present, the password must also be included. If **N** is present, a null password is not allowed. These parameters help control access to the servers.

The **D** parameter pertains to the context under which a particular server is used. If **D** is present, use the context set by the *default_username* field instead of the defaults provided by the login account.

Flag	User name	Password	Context
!!	needed	needed	user name
!U!	needed	-	user name
!P!	needed	needed	user name
!D!	-	-	default
!UP!	needed	needed	user name
!UD!	needed	-	default
!PD!	needed	needed	default
!UPD!	needed	needed	default
!UPN!	needed	needed	-

The *server_filename* is the name of the server program. This entry should be a fully-specified file name. The information included in **server.dat**(4) helps determine what must be done to run a client/server pair.

If a client/server pair is being debugged and a server 0 entry exists in **server.dat**(4), this entry can be used to specify the server to be run. When the **sni_connect**(3N) call is used, *sernum* is set to 0. *Server* points to the name of the server desired. Server 0 is convenient for debugging a server program for which a **server.dat**(4) entry is not desired until it runs properly.

A new entry can be added to **server.dat**(4) after a server is completed or if server 0 is not enabled. When **sni_connect**(3N) is used in a program, *sernum* equals the server number of the entry and *server* is a null pointer.

Once a server has been entered in **server.dat**(4), the server can be used with a client program as long as the flags set for that server allow the user to do so.

CLH - Clearinghouse

The Intergraph clearinghouse is a network-wide distributed database. It is used mainly to bind names and network addresses so that users do not need to remember or type in addresses. It can also be used to access any information that needs to be available to the network. The following CLH call looks up the object property value:

clh_vbyop(*object*, *property*, *value*, *size*)

Organization of the Clearinghouse

If the clearinghouse has been initialized on a node, it can be found in the subdirectory **/usr/lib**. As the figure below illustrates, the **nodes** directory has three subdirectories: **local**, **owned**, and **heard**. These subdirectories store information about the nodes and individuals on the network, such as their associated network addresses. The files in these subdirectories are **clearinghouse objects**.

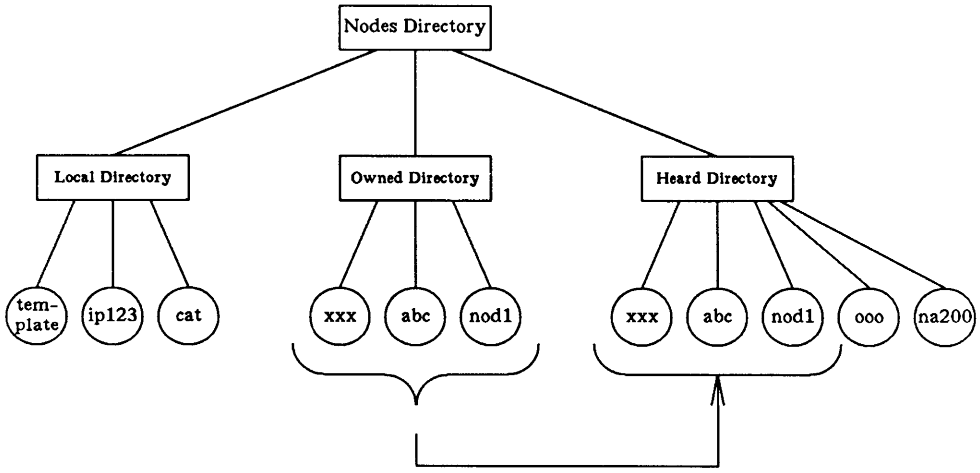


Figure 3-4: Organization of the Clearinghouse

After the clearinghouse is installed on a node, the local directory will contain an object named **template**. Template is a model for all clearinghouse entries in the owned directory. (When an object is created, it initially contains the contents of template.)

As the name implies, the local directory can also contain information that is known by the local node only. For example, suppose a node on the network was called **nod1** and had a clearinghouse entry of **nod1** containing information about that node. To name this clearinghouse entry **nod1** but include different information, place that entry in the local directory by using the **clh**(1) program to first delete the heard entry **nod1** and then add a **nod1** entry.

Only the local node knows the local entry and the information it contains. In addition, when the local node's clearinghouse searches for the **nod1** entry, it looks in its local directory first so it will find the local **nod1** clearinghouse entry instead of the **nod1** entry in the heard directory.

The owned directory contains the well-known node name entry. This entry lists the node's network address. If a new hardware address is assigned to the node, this entry is automatically updated to contain the new address. When the clearinghouse software is first installed on a system, the well-known node name is automatically entered in the owned directory.

All information in the owned directory including the well-known node name is propagated to the the local area network. Entries can be added to the owned directory using **clh**(1). **clh**(1) copies the template from the local directory to begin the new entry. Information about users on the local node or different names for the node (aliases) can be placed in the owned directory.

Only the well-known node name entry is updated when the hardware address changes. However, any entries in the local and owned directories that point to the well-known node name in an owned directory for a network address are indirectly updated when the well-known node name's address is updated. Any entries that do not point to the well-known node name for an address must be updated manually.

The heard directory contains all object entries known to the local area network. This includes the owned directory of every node but does not include the entries in the local directory. This directory contains information about nodes or individuals on a network even when all network facilities are not working properly. Each node's clearinghouse propagates one object in its owned directory once an hour, thus ensuring that the heard directories are kept up-to-date.

Each clearinghouse object is a text file stored in at least one of the node's subdirectories. Once the information about a node or individual is added to the clearinghouse, that node is represented by an object name and its associated properties. The name of the file is the object name. The file contents list the object properties and property values. An object name can have up to 14 characters and contain only lowercase alphanumeric characters and underscores. The clearinghouse does not accept any other characters (including blanks) in object names.

A list of one or more properties and property values is in each object's file. Only one property and property value is allowed per line. Each line looks like the following where the colon is the required delimiter:

*property***:***value*

The following lists well-known properties:

address	Specify a combination of a network and host address stored as a hexadecimal character string. The following shows a sample XNS address: 13498.08-00-36-28-8A-00
xns_address	Equivalent to **address**.
nodename	Specify a name representing a node, which the clearinghouse can use to look up a network address. (Maximum size is 14 characters.)
alias	Equivalent to **nodename**.
scope	Specify the LANs that this object should be propagated to. (The property is always propagated to the local LAN.)
namex_note	Specified when **namex**(1M) modifies the object.
namex_host	Specified when **namex**(1M) creates the object.
tcp_address	Specify an Internet address stored as a decimal character string. The following shows a sample Internet address: 129.135.200.100
netmap_info	Specify a formatted line containing information about the system and its components. This line is added or updated automatically at system startup.
owner	Specify the hexadecimal network address of the node from which the object was broadcast. This property is added automatically when the object is received from the network and therefore should only appear in heard objects.
SM_info	Specified by **sendmail**(1) installation. This information is specific to **sendmail**(1).

The well-known node name entry in the owned directory is created when the clearinghouse is installed on a node. The well-known node name may be changed on the Operating Systems Parameters Utility Page (accessed from the blue screen).

WARNING

The information stored in the well-known node name file can be changed, but do not change the address property value since this value is automatically updated by the clearinghouse to reflect the local node's hardware address.

Network Configuration Through incd and ifconfig

An entry in **/etc/inittab** invokes the **/etc/incd.start** shell script, which, in turn, invokes **incd**(1M) at boot time to configure the specified network protocols on a CLIX system. **incd**(1M) uses a configuration file, **/etc/incd.conf**, to determine which network protocols are to be initialized on the system.

Configuring the incd.conf File

/etc/incd.conf specifies the protocols to be started on an interface device. The network set-up can be customized by modifying the **/etc/incd.conf** configuration file. For example, if only the XNS protocols are desired on the Ethernet network interface, modify the **/etc/incd.conf** to appear as follows:

```
        et0 xns
```

If only the Department of Defense (DoD) Internet Protocol (IP) suite is desired on the Ethernet interface, **/etc/incd.conf** should appear as follows:

```
        et0 dod arp trlr udp tcp
```

In the above example, **dod** specifies the DoD Internet Protocol. This alone configures access to the Ethernet interface through IP.

The **arp** option specifies the Address Resolution Protocol, which is essential in mapping Internet addresses to physical Ethernet addresses.

The **trlr** option specifies the acceptance of trailer encapsulated Internet Protocol packets on this interface.

The **udp** option specifies the User Datagram Protocol, the connectionless service of the DoD Internet Protocol suite used by the Network File System (NFS), **rwho**(1), and the trivial file transfer protocol (**tftp**(1)).

The **tcp** option specifies the Transmission Control Protocol, the connection-oriented service of the DoD Internet Protocol suite. Utilities that use **tcp** include remote login (**rlogin**(1), **telnet**(1)) and the file transfer protocol (**ftp**(1)).

If communications are made directly through the Internet Protocol and the higher-level User Datagram Protocol and Transmission Control Protocol are not needed, only **dod** and **arp** need to be in the configuration file, as shown below:

```
et0 dod arp
```

If both XNS and the full DoD Internet Protocol suite are desired, **/etc/incd.conf** file should appear as follows:

```
et0 xns dod arp trlr udp tcp
```

NOTE The DoD Internet Protocol suite will not be configured for a given node unless there is an entry in the **/etc/hosts** file that corresponds to the local node name. Even if the protocols are specified in the /etc/incd.conf file, there must be an entry for the local node in the /etc/hosts file for the DoD Internet Protocol suite to be configured on that node.

Using ifconfig to Configure Network Interface Parameters

ifconfig(1M) configures the network interface parameters for the DoD Internet Protocol suite. It is also run at boot time on CLIX systems through the **/etc/incd.start** shell script. The parameters that can be configured through **ifconfig**(1M) are as follows:

- Internet address
- Subnetwork mask
- Broadcast address
- Marking an interface up (or down)
- Routing metric
- Authority to reply to ICMP Address Mask Requests

The default invocation of **ifconfig**(1M) in **/etc/incd.start** is as follows:

```
/etc/ifconfig et0 inet "$UNAME" up
```

et0 is the name of the Ethernet interface device. **inet** indicates that this setup is for the Internet address family. (Specifying **inet** is optional because the Internet address family is the only one supported by **ifconfig**(1M)). The node name of the system will be substituted for "$UNAME". **ifconfig**(1M) will look for this node name in the **/etc/hosts** file to get the corresponding Internet address that will be used as the local Internet address for this interface. **up** indicates to mark the interface "up" so that it will begin receiving Internet Protocol packets from the network.

When the interface is marked up, **ip**(7S) will send an Internet Control Message Protocol (ICMP) Address Mask Request out on the network. Other nodes on the network that are authorized to answer ICMP Address Mask Requests will reply with ICMP Address Mask Replies containing their subnetwork mask. The subnetwork mask will be set up for this interface using the first valid ICMP Address Mask Reply received.

Once the subnetwork mask is set for an interface, it cannot be modified; however, it is possible to manually specify a subnetwork mask on the **ifconfig** command line in the **/etc/incd.start** shellscript to be used at boot time. An example of specifying the subnetwork mask 255.255.255.0 follows:

```
/etc/ifconfig et0 inet "$UNAME" netmask 255.255.255.0 up
```

The address for an interface cannot be modified once it has been set; however, a particular address on the **ifconfig**(1M) command line can be specified in **/etc/incd.start** for use at boot time. An example of specifying the Internet address 129.135.200.1 follows:

```
/etc/ifconfig et0 inet 129.135.200.1 up
```

NOTE There must be an entry in the **/etc/hosts** file for the node name of the local host or **incd**(1M) will not configure the DoD Internet Protocols, thus rendering **ifconfig**(1M) useless.

The broadcast address will automatically be established for an interface based on the Internet address of the interface and the subnetwork mask. The broadcast address can be modified by the super-user at any time through **ifconfig**(1M). An example setting the broadcast address to 129.135.200.255 for the **et0** interface follows:

```
        /etc/ifconfig et0 inet broadcast 129.135.200.255
```

The interface accepting Internet packets from the network can be toggled at any time by the super-user. The **et0** interface is set to **up** by default in the **/etc/incd.start** shellscript.

To stop receiving the Internet Protocol traffic from the network, the following command should be issued:

```
        /etc/ifconfig et0 inet down
```

The routing metric determines the number of hops an Internet Protocol datagram would take before reaching its destination. The default is 1. This can be changed at any time by the super-user. An example of changing the routing metric to 3 follows:

```
        /etc/ifconfig et0 metric 3
```

By default, this implementation does not reply to ICMP Address Mask Requests. To use this interface as an authoritative agent for issuing ICMP Address Mask Replies, the following command should be issued by the super-user:

```
        /etc/ifconfig et0 maskrep
```

For more information on using **incd**(1M) and **ifconfig**(1M), refer to section 1M in the *CLIX System Administrator's Reference Manual.*

4. BSD Porting Tutorial

Chapter 4: BSD Porting Tutorial

Introduction

This tutorial describes porting programs from a 4.3 Berkeley Software Distribution (BSD) environment to a CLIX environment. The enhancements made to support many BSD routines will be discussed. In addition, include file differences and general examples of how to use standard CLIX routines to imitate BSD routines that are not provided will be covered.

This tutorial does not cover all possible inconsistencies encountered when going from a 4.3 BSD system to a CLIX system. However, enough information should be provided for the programmer to find a workaround for any differences.

This is not a completely compatible 4.3 BSD system. However, we have attempted to provide as much compatibility as possible on a per-system-call or library-routine basis. Error numbers returned from system calls may differ slightly. See section (2B) in the *CLIX Programmer's & User's Reference Manual* for exact errors to be returned from each system call.

The implemented system calls are listed on the following page. Any known inconsistencies are discussed in the remainder of the section. For a list of implemented library routines, see section (3B) of the *CLIX Programmer's & User's Reference Manual.* We are not aware of any discrepancies in the library routines. Therefore, these routines will not be discussed in this tutorial. Several routines under BSD and CLIX differ only in name. For example, the BSD **getwd()** and standard UNIX **getcwd(3C)** routines both return the current working directory. A list of such routines is not provided because of the difficulty in supplying a complete list.

All routines added to CLIX to provide BSD compatibility may be accessed with the BSD library, **libbsd**.

System Call Compatibility

The 4.3 BSD system calls provided under CLIX are as follows:

```
accept()              /* accept a connection on a socket */
bind()                /* bind a name to a socket */
connect()             /* initiate a connection on a socket */
ftruncate()           /* truncate a file to a given length */
getdtablesize()       /* get descriptor table size */
gethostid()           /* get identifier of current host */
gethostname()         /* get name of current host */
getpeername()         /* get name of connected peer */
getpgrp()             /* get process group */
getsockname()         /* get name of socket */
getsockopt()          /* get options on a socket */
gettimeofday()        /* get current date and time */
killpg()              /* send signal to a process group */
listen()              /* listen for connections on a socket */
lstat()               /* get status of a file */
readlink()            /* read value of a symbolic link */
readv()               /* read input from scattered locations */
recv()                /* receive a message from a socket */
recvfrom()            /* receive a message from a socket */
recvmsg()             /* receive a message from a socket */
rename()              /* change the name of a file */
select()              /* synchronous I/O multiplexing */
send()                /* send a message from a socket */
sendmsg()             /* send a message from a socket */
sendto()              /* send a message from a socket */
sethostid()           /* set identifier of current host */
sethostname()         /* set name of current host */
setpgrp()             /* set process group */
setsockopt()          /* set options on a socket */
shutdown()            /* shut down part of a full-duplex connection */
socket()              /* create an endpoint for communication */
socketpair()          /* create a pair of connected sockets */
symlink()             /* make a symbolic link to a file */
vfork()               /* spawn a new process in a virtual way */
wait3()               /* wait for a process to terminate */
writev()              /* write input to scattered locations */
```

Most of these BSD system calls are implemented as system calls under CLIX. However, a few were implemented as library routines using standard System V.3 primitives. These are still documented in section (2B) of the *CLIX Programmer's & User's Reference Manual* because they represent system calls under BSD.

Any known descrepancies between the BSD and CLIX implementations, aside from error codes, will be discussed in the remainder of this section. For possible errors returned in the global variable *errno*, see the manual page in section (2B) of the *CLIX Programmer's & User's Reference Manual* for the specific system call.

File Truncation

ftruncate(2B) is provided to allow files to be truncated to a given length. The **truncate**() routine, which takes a file name, as opposed to **ftruncate**(2B), which takes a file descriptor, is not provided. The **truncate**() routine may be simulated by opening the file for writing with the **open**(2) system call and using the descriptor returned as the argument to **ftruncate**(2B) as shown below.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/fcntl.h>
...
extern int errno;
...
main()
{
        int fd;
        off_t length;
        ...

        /*
         * The call to open() and ftruncate() replace
         * a call to truncate().
         */

        if ((fd = open("foobar", O_RDWR, 0)) == -1) {
                perror("open");
                exit(errno);
```

continued

```
        }

        /*
         * Truncate file to length bytes, in this
         * case, 1024 bytes
         */

        length = 1024;
        if (ftruncate(fd, length) == -1) {
                perror("ftruncate");
                exit(errno);
        }
        ...
}
```

Process Groups

Two BSD system calls, **getpgrp**() and **setpgrp**(), are provided under different names. They were renamed to **getpgrp2**(2B) and **setpgrp2**(2B) to avoid clashes with the standard System V.3 **getpgrp**(2) and **setpgrp**(2) system calls.

Virtual Fork

vfork(2B) is a BSD addition that allows a new process to be spawned without copying the entire address space of the calling process. Under CLIX, a process's stack region is the only portion of the address space copied.

Interprocess Communication

BSD sockets are implemented in the Internet and UNIX domains. Stream and datagram abstractions are provided in the Internet domain. The UNIX domain supports only the stream abstraction.

The **getsockopt**(2B) and **setsockopt**(2B) system calls support only a subset of the options that 4.3 BSD offers. SO_SNDBUF and SO_RCVBUF are not supported. The operating system sets the buffer sizes. They range from 4 to 4096 bytes with several sizes in between. The system chooses the size for a request according to the smallest buffer available to fit a request. If a request exceeds 4096 bytes, several buffers may be linked by the system to obtain a buffer large enough for the request. SO_ERROR is also not supported. Errors will be returned when they occur or on the first use after the error's occurrence of the affected socket. SO_DEBUG and SO_DONTROUTE are always off. Attempting to set them will result in an error. SO_KEEPALIVE, SO_LINGER and SO_BROADCAST are always on. Attempting to clear them will result in an error.

The queue limit (as set by **listen**(2B)) for incoming connections on a socket is currently zero. Attempting to set it higher will not result in an error. The system will silently set it to zero.

I/O

The **select**(2B) call supported on files, sockets, pseudo terminals, and tty's in BSD is only supported on sockets and pseudo terminals under CLIX. **readv**(2B) and **writev**(2B), which perform scatter-gather I/O for files and sockets under BSD, only work on sockets under CLIX. When performing scatter-gather I/O for files, you will need to perform a **read**(2) or **write**(2) for each scatter-gather location. (See figures 4-1 and 4-2.) All forms of the **recv**(2B) system call and the **send**(2B) system call support only the MSG_OOB flag. Any other flags will result in an error.

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/uio.h>
...
extern int errno;
...
main()
{
        char buf1[10], buf2[10];
        int fd, cc;
        struct iovec iov[2];
        ...

        /*
         * Set up scatter-gather locations
         */

        iov[0].iov_base = buf1;
        iov[0].iov_len = 10;
        iov[1].iov_base = buf2;
        iov[1].iov_len = 10;

        /*
         * Perform scatter-gather.
         * The fd argument was obtained from a previous
         * open(), dup(), etc. system call.
         */

        if ((cc = readv(fd, &iov[0], 2)) == -1) {
                perror("readv");
                exit(errno);
        }

        ...

}
```

Figure 4-1:

```
#include <sys/errno.h>
...
extern int errno;
...
main()
{
        char buf1[10], buf2[10];
        int fd, cc;
        ...
        /*
         * Perform 2 reads to accomplish what the
         * readv() call in figure 4-1 does.
         * The fd argument was obtained from a previous
         * open(), dup(), etc. system call.
         */
        if ((cc = read(fd, buf1, 10)) == -1) {
                perror("readv");
                exit(errno);
        }
        if ((cc = read(fd, buf2, 10)) == -1) {
                perror("readv");
                exit(errno);
        }
        ...
}
```

Figure 4-2:

Programs that cannot block while performing I/O have two methods for blocking under BSD. One method is to use an **ioctl(2)** call with the command FIONBIO set. The other is to call **fcntl(2)** with the F_SETFL command and a flag of FNDELAY. CLIX only supports the **fcntl(2)** call. Therefore, nonblocking I/O must be set with an **fcntl(2)** instead of an **ioctl(2)**. See the following figure.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
...
extern int errno;
...
main()
{
        int fd;
        ...
        /*
         * The fd argument was obtained from a
         * previous open(), dup(), etc. system call.
         */
        if (fcntl(fd, F_SETFL, FNDELAY) == -1) {
                perror("fcntl");
                exit(errno);
        }
        ...
}
```

General

gettimeofday(2B) and **rename**(2B) were implemented as library routines using standard System V.3 primitives. As a result, the granularity needed to provide accurate microseconds for **gettimeofday**(2B) does not exist and always has a value of 0. The **link**(2B) and **unlink**(2B) system calls were used to implement the **rename**(2B) call. Since several system calls are needed to implement **rename**(2B), signals that would not be received during the **rename**() call under BSD may be received during the **rename**(2B) call.

wait3(2B) is implemented as a library routine using the **wait2**(2I) system call. This prevents the third argument to **wait3**(2B) from being used.

Include File Compatibility

The following procedure was performed to achieve compatibility between BSD and CLIX include files. When no CLIX include file similar to a BSD include file existed, the entire BSD include file was added to the CLIX system. However, when a similar file was present, parts of its BSD counterpart were added to the existing CLIX include file. Compatibility was not achieved in two areas. First, when going from a BSD program that includes **<fcntl.h>**, **<sys/file.h>** will usually be needed. This need depends on **<sys/types.h>**. Secondly, **<dir.h>** should be replaced with **<dirent.h>**.

Signal Compatibility

BSD and CLIX support a **signal(2)** system call. However, they differ in the actions performed. When porting from a BSD environment to CLIX, you should change all **signal(2)** calls to **sigset(2)** calls. **sigset(2)** is a standard System V.3 system call that acts as the BSD **signal(2)** call does.

The BSD sigmask macro is equivalent to the CLIX sigbit macro defined in **<signal.h>**. The BSD **sigsetmask()** and **sigblock()** system calls may be emulated by the System V.3 **sigrelse(2)** and **sighold(2)** calls, respectively. The BSD calls act on a mask of signals. Whereas, the System V.3 calls act on an individual signal. (See figures 4-3 and 4-4.) Therefore, multiple calls may be needed under CLIX to perform the actions of a single call under BSD.

```
#include <signal.h>
...
main()
{
        int mask;
        ...
        mask = signask(SIGALRM);
        mask = signask(SIGIO);
        sigblock(mask);
        ...
        sigsetmask(mask);
        ...
}
```

Figure 4-3: BSD Signals

```
#include <signal.h>
#include <sys/errno.h>
...
extern int errno;
...
main()
{
        ...
        /*
         * Call sighold for each signal number that was
         * set in the mask in Figure 4-3
         */
        if (sighold(SIGALRM) == -1) {
                perror("sighold");
                exit(errno);
        }
        if (sighold(SIGIO) == -1) {
                perror("sighold");
                exit(errno);
        }
        ...
        /*
         * Call sigrelse for each signal number that was
         * set in the mask in Figure 4-3
         */
        if (sigrelse(SIGALRM) == -1) {
                perror("sigrelease");
                exit(errno);
        }
        if (sigrelse(SIGIO) == -1) {
                perror("sigrelease");
                exit(errno);
        }
        ...
}
```

Figure 4-4: CLIX Signals

5. Introductory Socket Tutorial

Chapter 5: Introductory Socket Tutorial

Introduction

CLIX offers several choices for interprocess communication. To aid the programmer in developing programs composed of cooperating processes, the different choices are discussed and a series of example programs presented. These programs demonstrate in a simple way the use of *pipes*, *socketpairs*, and *sockets* and *datagram* and *stream* communication. This document intends to present a few simple example programs, not to describe the networking system in full.

Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions, and end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking*. In UNIX these lines can never rejoin. A call to the system routine **fork**(2) causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by **fork**(2). The parent will receive a process ID for the child; the child will receive a zero. Calls to **fork**(2), therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets. (Sockets are communication objects that will be discussed below.) Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, **stdin, stdout** and **stderr**. These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal, the child's input will be taken from the same terminal. (Devices are also treated as files in UNIX.) Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

Pipes

Most UNIX users know that they can pipe the output of a program **prog1** to the input of another, **prog2**, by typing the command **prog1 | prog2**. This is called *piping* the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple such as **prog1**, the shell forks a process, which executes the program, **prog1**, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, **prog1 | prog2**, the shell creates two processes connected by a pipe. One process runs the program, **prog1**; the other runs **prog2**. The pipe is an I/O mechanism with two ends, or sockets. Data that is written to one socket can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without the program text being changed. It is in this way that the shell is able to set up pipes. Before executing **prog1**, the process can close whatever is at **stdout** and replace it with one end of a pipe. Similarly, the process that will execute **prog2** can substitute the opposite end of the pipe for **stdin**.

Consider a program that creates a pipe for communication between its child and itself (Figure 5-1). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 5-1, the parent process calls the system routine **pipe**(2). This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. **pipe**(2) is passed an array in which it places the index numbers of the sockets it created. The two ends are not equal. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling **fork**(2). Figure 5-2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

```
#include <stdio.h>
#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks.  The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device.  I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */
main()
{
        int sockets[2], child;

        /* Create a pipe */

        if (pipe(sockets) < 0) {
                perror("opening stream socket pair");
                exit(10);
        }
        if ((child = fork()) == -1)
                perror("fork");
        else if (child) {
                char buf[1024];
                /* This is still the parent.  It reads the child's message. */
                close(sockets[1]);
                if (read(sockets[0], buf, 1024) < 0)
                        perror("reading message");
                printf("-->%s\n", buf);
                close(sockets[0]);
        } else {

                /* This is the child.  It writes a message to its parent. */
                close(sockets[0]);
                if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                        perror("writing message");
                close(sockets[1]);
        }
}
```

Figure 5-1: Use of a pipe

A pipe is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or child to parent. Using the same pipe for communicating both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one to use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. Unused descriptors are not required to be closed, but it is good practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When readers request a certain number of bytes from this stream, they are given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to **write**(2) or from several calls to **write**(2).

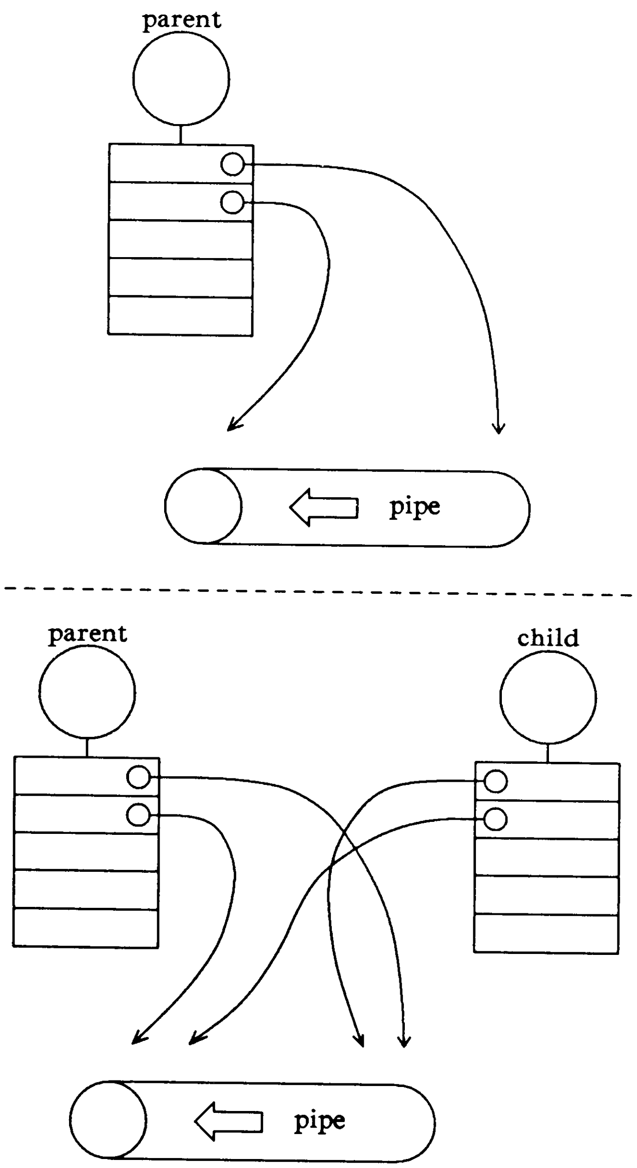


Figure 5-2. Parent and Child Sharing a Pipe

Socketpairs

Sockets provide a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. You may obtain a pair of connected sockets for two-way stream communication by calling the routine **socketpair**(2B). The program in Figure 5-3 calls **socketpair**(2B) to create such a connection. The program uses the link for communication in both directions. Since socketpairs are extensions of pipes, their use resembles that of pipes. Figure 5-4 illustrates the result of a fork following a call to **socketpair**(2B).

socketpair(2B) takes a specification of a domain, a style of communication, a protocol and an array in which to return the 2 socket descriptors opened as arguments. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows sockets on the same machine to communicate.

Note that the header files **<sys/socket.h>** and **<sys/types.h>** are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in **<sys/socket.h>**, which, in turn, requires the file **<sys/types.h>** for some of its definitions.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes,
 * however, socketpairs are two-way communications objects. Therefore I can
 * send messages in both directions.
 */

main()
```

continued

```
{
        int sockets[2], child;
        char buf[1024];


        if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
                perror("opening stream socket pair");
                exit(1);
        }

        if ((child = fork()) == -1)
                perror("fork");
        else if (child) {       /* This is the parent. */
                close(sockets[0]);
                if (read(sockets[1], buf, 1024, 0) < 0)
                        perror("reading stream message");
                printf("-->%s\n", buf);
                if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
                        perror("writing stream message");
                close(sockets[1]);
        } else {                /* This is the child. */
                close(sockets[1]);
                if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                        perror("writing stream message");
                if (read(sockets[0], buf, 1024, 0) < 0)
                        perror("reading stream message");
                printf("-->%s\n", buf);
                close(sockets[0]);
        }
}
```

Figure 5-3: Use of a Socketpair

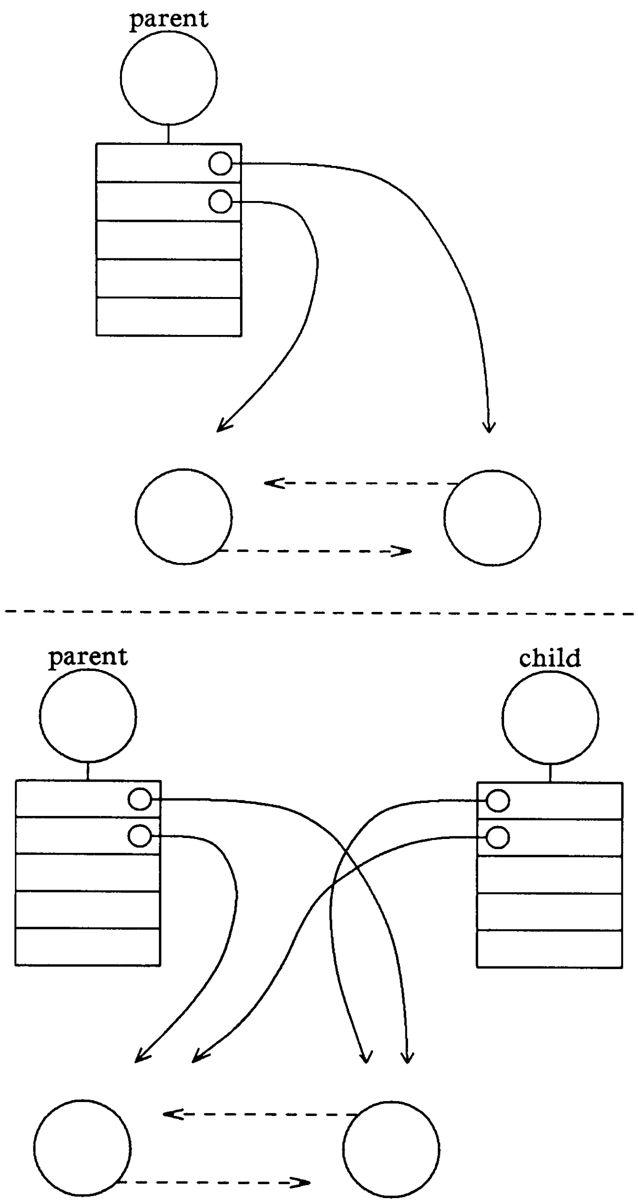


Figure 5-4: Parent and Child Sharing a Socketpair

Domains and Protocols

Pipes and socketpairs are a simple solution for a parent and child or child processes to communicate. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs can be used in this situation, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. With sockets you can create individual sockets, give them names, and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain*. Several domains for sockets exist. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the socket by giving the proper path name. UNIX domain names, therefore, allow any two processes that work in the same file system to communicate. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols TCP/IP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow machines to communicate.

Communication follows some particular "style." Currently, communication is either through a *stream* or by *datagram*. Stream communication has several implications. Communication occurs across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to **write**(2) only part of the data from a single call, if not enough room exists for the entire message or if not all data from a large message is transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if you try to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read. Thus, message boundaries are preserved. Datagrams in the UNIX domain are not supported under CLIX.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain found when using datagrams must be weighed against the increased complexity of the program, which is now concerned with lost or out-of-order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequently using the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats, and conventions that regulate the transfer of data between participants in the communication. In general, each socket type (stream, datagram, etc.) has one protocol within each domain. The code that implements a protocol tracks the names bound to sockets, sets up connections, and transfers data between sockets, perhaps sending the data across a network. Several protocols, differing only in low-level details, can implement the same style of communication within a particular domain. Although the protocol to be used can be selected, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

When a socket is created, the domain, style, and protocol of a socket may be specified. For example, in Figure 5-5 the call to **socket**(2B) causes a datagram socket with the default protocol in the Internet domain to be created.

Datagrams in the Internet Domain

The examples in Figures 5-5 and 5-6 are examples of datagram sockets in the Internet domain. The structure of Internet domain addresses is defined in the file **<netinet/in.h>**. Internet addresses specify a host address (a 32-bit number) and a delivery slot or port on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered in the file system and, therefore, they do not need to be unlinked after the socket is closed. When a message must be sent between machines, it is sent to the protocol routine on the destination machine, which interprets the address to determine the socket the message should be delivered to. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple <*protocol, local machine address, local port, remote machine address, remote port*>. An association may be transient when using datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/* In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *      short   sin_family;
 *      u_short sin_port;
 *      struct in_addr sin_addr;
 *      char    sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{       int sock, length;
        struct sockaddr_in name;
        char buf[1024];
        /* Create socket from which to read. */
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
```

continued

```
                perror("opening datagram socket");
                exit(1);
        }
        /* Create name with wildcards. */
        name.sin_family = AF_INET;
        name.sin_addr.s_addr = INADDR_ANY;
        name.sin_port = 0;
        if (bind(sock, &name, sizeof(name))) {
                perror("binding datagram socket");
                exit(1);
        }
        /* Find assigned port value and print it out. */
        length = sizeof(name);
        if (getsockname(sock, &name, &length)) {
                perror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(name.sin_port));
        /* Read from the socket */
        if (read(sock, buf, 1024) < 0)
                perror("receiving datagram packet");
        printf("-->%s\n", buf);
        close(sock);
}
```

Figure 5-5: Reading Internet Domain Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is:
 *  dgramsend hostname portnumber
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in name;
        struct hostent *hp, *gethostbyname();

        /* Create socket on which to send. */

        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("opening datagram socket");
                exit(1);
        }

        /*
         * Construct name, with no wildcards, of the socket to send to.
         * Gethostbyname() returns a structure including the network address
         * of the specified host.  The port number is taken from the command
         * line.
         */

        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host0, argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
        name.sin_family = AF_INET;
        name.sin_port = htons(atoi(argv[2]));
```

continued

```
        /* Send message. */

        if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
                perror("sending datagram message");
        close(sock);
}
```

Figure 5-6: Sending an Internet Domain Datagram

sockets; the association actually exists during a **send**() operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one. It can also be the wildcard value INADDR_ANY. The wildcard value is used in the program in Figure 5-5. If a machine has several network addresses, the messages sent to any of the addresses should likely be deliverable to a socket. This will be the case if the wildcard value is chosen. Even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from "anywhere," but one cannot send a message "anywhere." The program in Figure 5-6 is given the destination host name as a command-line argument. To determine a network address to which it can send the message, it looks up the host address by the call to **gethostbyname**(2B). The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be considered the number of a mailbox into which the protocol places messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit **bind**(2B) call is made with a port number of 0 or when a **connect**(2B) or **send**(2B) is performed on an unbound socket. Port numbers are not automatically reported back to the user. After calling **bind**(2B), asking for port 0, you may call **getsockname**(2B) to discover what port was actually assigned. The routine **getsockname**(2B) will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing the port number returned by **getsockname**(2B) may result in a misinterpretation. To print the number, you must use the routine **ntohs**(3B) (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines (such as 68000-based machines) this is a null operation. On others (such as the CLIPPER) this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called **htons**(2B); similar routines exist for long integers. For further information, refer to **byteorder**(3B) of the *CLIX Programmer's & User's Reference Manual.*

Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 5-7 and 5-8 show two programs that create such a connection. The program in 7 is relatively simple. To initiate a connection, this program creates a stream socket and then calls **connect**(2B), specifying the address to which it wishes the socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, the operating system sends a SIGPIPE signal to the process. Unless explicit action is taken to handle the signal (see **signal**(2) or **sigset**(2)), the process will terminate and the shell will print the message "broken pipe."

Forming a connection is asymmetrical; one process (such as the program in Figure 5-7) requests a connection with a particular socket; the other process accepts connection requests. Before a connection can be accepted, a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 5-9. Process 2 created a socket and bound a port number to it. Process 1 created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps, to several other potential communicants as well. If several possible communicants exist, this socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is:  streamwrite hostname portnumber
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();
        char buf[1024];

        /* Create socket */

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }

        /* Connect socket using name specified by command line. */

        server.sin_family = AF_INET;
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host0, argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_port = htons(atoi(argv[2]));

        if (connect(sock, &server, sizeof(server)) < 0) {
                perror("connecting stream socket");
                exit(1);
        }
```

continued

```
        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
        close(sock);
}
```

Figure 5-7: Initiating an Internet Domain Stream Connection

The program in Figure 5-8 is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case, it prints the socket number.) The program then calls **listen**(2B) for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. **listen**(2B) marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of **listen**(2B); the maximum length is limited by the system. Once the listen call is complete, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 5-9 shows the result of Process 1 connecting with the named socket of Process 2 and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The **accept**(2B) call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        int i;

        /* Create socket */

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }

        /* Name socket using wildcards */

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
                perror("binding stream socket");
                exit(1);
        }

        /* Find out assigned port number and print it out */

        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
```

continued

```
                perror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(server.sin_port));

        /* Start accepting connections */

        listen(sock, 5);
        do {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("accept");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                                perror("reading stream message");
                        i = 0;
                        if (rval == 0)
                                printf("Ending connection\n");
                        else
                                printf("-->%s\n", buf);
                } while (rval != 0);
                close(msgsock);
        } while (TRUE);

        /*
         * Since this program has an infinite loop, the socket "sock" is
         * never explicitly closed.  However, all sockets will be closed
         * automatically when a process is killed or terminates normally.
         */
}
```

Figure 5-8: Accepting an Internet Domain Stream Connection

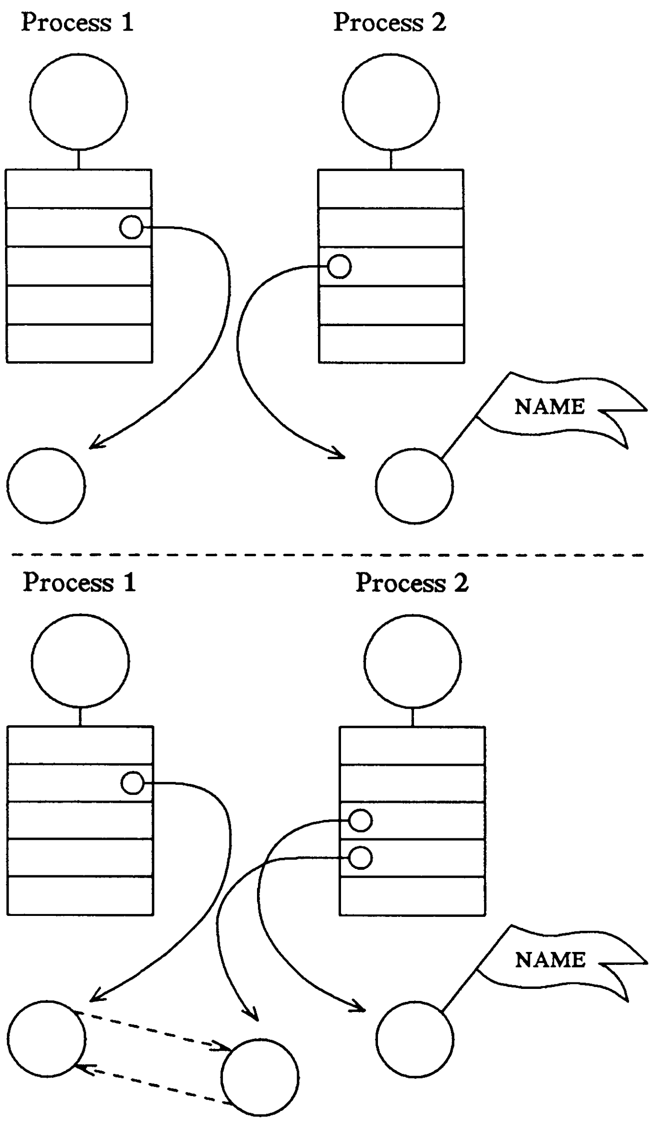


Figure 5-9: Establishing a Stream Connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        fd_set ready;
        struct timeval to;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
                perror("binding stream socket");
                exit(1);
        }
        /* Find out assigned port number and print it out */
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                perror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(server.sin_port));

        /* Start accepting connections */
```

continued

```
        listen(sock, 5);
        do {
                FD_ZERO(&ready);
                FD_SET(sock, &ready);
                to.tv_sec = 5;
                if (select(sock + 1, &ready, 0, 0, &to) < 0) {
                        perror("select");
                        continue;
                }
                if (FD_ISSET(sock, &ready)) {
                        msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
                        if (msgsock == -1)
                                perror("accept");
                        else do {
                                bzero(buf, sizeof(buf));
                                if ((rval = read(msgsock, buf, 1024)) < 0)
                                        perror("reading stream message");
                                else if (rval == 0)
                                        printf("Ending connection\n");
                                else
                                        printf("-->%s\n", buf);
                        } while (rval > 0);
                        close(msgsock);
                } else
                        printf("Do something else\n");
        } while (TRUE);
}
```

Figure 5-10: Using **select**(2B) to Check for Pending Connections

The program in Figure 5-10 is a slight variation on the server in Figure 5-8. It avoids blocking when there are no pending connection requests by calling **select**(2B) to check for pending requests before calling **accept**(2B). This strategy is useful when connections may be received on more than one socket or when data may arrive on other connected sockets before another connection request.

The programs in Figures 5-11 and 5-12 show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this type of program in the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system differ. Some differences exist, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this paper.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is:
 *                      ustreamwrite pathname
 */
main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_un server;
        char buf[1024];

        /* Create socket */
        sock = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Connect socket using name specified by command line. */
        server.sun_family = AF_UNIX;
        strcpy(server.sun_path, argv[1]);

        if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
                close(sock);
                perror("connecting stream socket");
                exit(1);
```

continued

```
        }
        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
}
```

Figure 5-11: Initiating a UNIX Domain Stream Connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it.  When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
main()
{
        int sock, msgsock, rval;
        struct sockaddr_un server;
        char buf[1024];

        /* Create socket */
        sock = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
        /* Name socket using file system name */
```

continued

```
        server.sun_family = AF_UNIX;
        strcpy(server.sun_path, NAME);
        if (bind(sock, &server, sizeof(struct sockaddr_un))) {
                perror("binding stream socket");
                exit(1);
        }
        printf("Socket has name %s\n", server.sun_path);
        /* Start accepting connections */
        listen(sock, 5);
        for (;;) {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("accept");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                                perror("reading stream message");
                        else if (rval == 0)
                                printf("Ending connection\n");
                        else
                                printf("-->%s\n", buf);
                } while (rval > 0);
                close(msgsock);
        }
        /*
         * The following statements are not executed, because they follow an
         * infinite loop.  However, most ordinary programs will not run
         * forever.  In the UNIX domain it is necessary to tell the file
         * system that one is through using NAME.  In most programs one uses
         * the call unlink() as below. Since the user will have to kill this
         * program, it will be necessary to remove the name by a command from
         * the shell.
         */
        close(sock);
        unlink(NAME);
}
```

Figure 5-12: Accepting a UNIX Domain Stream Connection

Reads, Writes, Recvs, etc.

UNIX 4.3 BSD has several system calls for reading and writing information. The simplest calls are **read**(2) and **write**(2). **write**(2) takes the index of a descriptor, a pointer to a buffer containing the data and the size of the data as arguments. The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket (as described in the Connections Section) or a datagram socket for which a **connect**(2B) call has provided a default destination (see **connect**(2B)). **read**(2) also takes a descriptor that indicates either a file or a socket. **write**(2) requires a connected socket since no destination is specified in the parameters of the system call. **read**(2) can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on **read**(2) and **write**(2) that allow the source and destination of the input and output to use several separate buffers, which are only available on sockets. These are **readv**(2B) and **writev**(2B), for read and write *vector*.

It is sometimes necessary to send high-priority data over a connection that may have unread low-priority data at the other end. For example, a user interface process may be interpreting commands and sending them to another process through a stream connection. The user interface may have filled the stream with, as yet, unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high-priority data wait to be processed after the low-priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal if this signal has been enabled (see **signal**(2) or **sigset**(2)). See the "Advanced Socket Tutorial" for a more complete description of the OOB mechanism. A pair of calls similar to **read**(2) and **write**(2) allow options, including sending and receiving OOB information; these are **send**(2B) and **recv**(2B). When not using these options, these calls have the same functions as **read**(2) and **write**(2). These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status.

```
/*
 * The variable descriptor may be the descriptor of a file or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;
/*
 * The variable ''sock'' must be the descriptor of a socket.
 * Flags may be MSG_OOB.
 * An iovec can include several source buffers.
 */
cc = readv(sock, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = writev(sock, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 5-13: Varieties of Read and Write Commands

To send datagrams, you must be allowed to specify the destination. The call **sendto**(2B) takes a destination address as an argument and is therefore used for sending datagrams. The call **recvfrom**(2B) is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, you may use **read**(2) or **recv**(2B).

Finally, a pair of calls allow the sending and receiving of messages from multiple buffers when the address of the recipient must be specified. These are **sendmsg**(2B) and **recvmsg**(2B). These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

The various options for reading and writing are shown in Figure 5-13 with their parameters. The parameters for each system call reflect the different functions of the different calls. In the examples given in this paper, the calls **read**(2) and **write**(2) are used when possible.

Choices

This paper presented examples of some of the forms of communication supported by Intergraph's port of 4.3 BSD sockets. These were presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

Pipes have the advantage of portability in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagram and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as only 64 entries are usually available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that, for delivering a short message, the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often make you decide to choose streams.

What to Do Next

Many examples presented here can be models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in *The UNIX Programming Environment*. Further documentation on Intergraph's port of 4.3 BSD socket mechanisms can be found in the *Advanced Socket Tutorial*. More detailed information about particular calls and protocols is provided in sections (2), (3), and (4) of the *CLIX Programmer's & User's Reference Manual*. In particular, the following manual pages are relevant:

Creating and naming sockets	socket(2B), bind(2B)
Establishing connections	listen(2B), accept(2B), connect(2B)
Transferring data	read(2), write(2B), send(2B), recv(2B)
Addresses	inet(7B)
Protocols	tcp(7B), udp(7B).

References

B.W. Kernighan & R. Pike, 1984, *The UNIX Programming Environment*. Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978, *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall.

The "Advanced Socket Tutorial."

CLIX Programmer's & User's Reference Manual.

6. Advanced Socket Tutorial

Chapter 6: Advanced Socket Tutorial

Introduction

This document provides a high-level description of the 4.3 BSD Interprocess Communication (IPC) facilities available under CLIX. It is designed to complement the manual pages for the IPC primitives by giving examples of their use. The remainder of this document is organized into four sections. The "Basics" section introduces the IPC-related system calls and the basic model of communication. The "Network Library Routines" section describes some of the supporting library routines users may find helpful in constructing distributed applications. The "Client/Server Model" section is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. The "Advanced Topics" section delves into advanced topics that sophisticated users are likely to encounter when using the IPC facilities.

Basics

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain, sockets are named with UNIX path names; for example, a socket may be named **/dev/foo**. Sockets normally exchange data only with sockets in the same domain. (It may be possible to cross domain boundaries, but only if some translation process is performed.) The 4.3 BSD IPC facilities support three separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes that communicate using the DARPA standard communication protocols; and the NS domain, which is used by processes that communicate using the Xerox standard communication protocols. (See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS) 028112 for more information.) The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions possible when operating in the Internet (or NS) domain.

Socket Types

Sockets are typed according to the communication properties visible to a user. Processes are presumed to communicate only among sockets of the same type, although nothing prevents communication among sockets of different types should the underlying communication protocols support this.

Two types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes. (In the UNIX domain, in fact, the semantics are identical.

A *datagram* socket supports bidirectional flow of data that is not guaranteed to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which they were sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet-switched networks such as the Ethernet.

Other socket types not currently supported under the CLIX implementation are *raw* and *sequenced packet* sockets.

Socket Creation

To create a socket, use the **socket**(2B) system call as follows:

```
s = socket(domain, type, protocol);
```

This call requests the system to create a socket in the specified *domain* and of the specified *type*. A particular *protocol* may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from protocols that compose the communication domain and that may be used to support the requested socket type. The user is returned a descriptor that may be used in later system calls that operate on sockets. The domain is specified as one of the manifest constants defined in the **<sys/socket.h>** file. For the UNIX domain, the constant is AF_UNIX; for the Internet domain it is AF_INET; and for the NS domain it is AF_NS. (The manifest constants are named AF_*whatever* as they indicate the address format used in interpreting names.) The socket types are also defined in this file and either SOCK_STREAM or SOCK_DGRAM must be specified. To create a stream socket in the UNIX domain, use the following call:

```
s = socket(AF_UNIX, SOCK_STREAM, 0);
```

To create a datagram socket in the Internet domain, use the following call:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The above call would create a datagram socket with the User Datatgram Protocol (UDP) protocol providing the underlying communication support.

The default protocol (used when the *protocol* argument to the **socket**(2B) call is 0) should be useful in most situations. However, it is possible to specify a protocol other than the default; this will be covered in the "Advanced Topics" section.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT) or a request for a type of socket that is not supported for the given domain (ESOCKTNOSUPPORT).

Binding Local Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet and NS domains, an association is composed of local and foreign addresses and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names. (The phrase "foreign path name" means a path name created by a foreign process, not a path name on a foreign system.) In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local path name, foreign path name> tuples. The path names may not refer to files already existing on the system.

The **bind**(2B) system call allows a process to specify half of an association, <local address, local port> (or <local path name>), while the **connect**(2B) and **accept**(2B) primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the **connect**(2B) and **send**(2B) calls will automatically bind an appropriate address if they are used with an unbound socket. The process of binding names to NS sockets is similar in most ways to that of binding names to Internet sockets.

The **bind**(2B) system call is used as follows:

```
    bind(s, name, namelen);
```

The bound name is a variable-length byte string interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain. (This is one of the properties that compose the *domain*.) As mentioned, in the Internet domain, names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. To bind the name **/tmp/foo** to a UNIX domain socket, use the following code:

```
#include <sys/un.h>
 . . .
struct sockaddr_un addr;
 . . .
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
        sizeof (addr.sun_family));
```

In determining the size of a UNIX domain address null bytes are not counted, which is why **strlen**(3C) is used. In the current implementation of UNIX domain IPC, the file name referred to in **addr.sun_path** is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where **addr.sun_path** is to reside, and this file should be deleted by the caller when it is no longer needed.

Binding an Internet address is more complicated. The actual call is similar,

```
#include <sys/types.h>
#include <netinet/in.h>
 . . .
struct sockaddr_in sin;
 . . .
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in the "Network Library Routines" section when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS host names. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
 . . .
struct sockaddr_ns sns;
 . . .
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Discussion of what to place in a "struct sockaddr_ns" will be deferred to the "Network Library Routines" Section.

Connection Establishment

Connection establishment is usually asymmetric, with one process a client and the other a **server**. The server, when willing to offer its advertised services, ***binds*** a socket to a well-known address associated with the service and then passively ***listens*** on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the **connect**(2B) call is used to initiate a connection. Using the UNIX domain, this might appear as,

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server, strlen(server.sun_path) +
        sizeof (server.sun_family));
```

in the Internet domain,

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

and in the NS domain,

```
struct sockaddr_ns server;
 . . .
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where **server** in the previous example would contain either the UNIX path name, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful. (Any name automatically bound by the system, however, remains.) Otherwise, the socket is associated with the server and data transfer may begin.

For the server to receive a client's connection, it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
        listen(s, 1);
```

The second parameter to the **listen**(2B) call specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages that compose the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up. As it is now, it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of one pending connection on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog and then ignoring all connection requests.

With a socket marked as listening, a server may **accept(2B)** a connection:

```
struct sockaddr_in from;
 . . .
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a **struct sockaddr_un**, and for the NS domain, *from* would be declared as a **struct sockaddr_ns**, but nothing different would need to be done as far as *fromlen* is concerned. In the examples that follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, and then it is modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

accept(2B) normally blocks. That is, **accept(2B)** will not return until a connection is available or the system call is interrupted by a signal to the process. Further, a process cannot indicate it will accept connections from only a specific individual(s). The user process must consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket or wants to avoid blocking on the accept call, there are alternatives; these will be considered in the "Advanced Topics" section.

Data Transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As you might expect, in this case, the normal **read**(2) and **write**(2) system calls are usable.

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to **read**(2) and **write**(2), the new calls **send**(2B) and **recv**(2B) may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While **send**(2B) and **recv**(2B) are virtually identical to **read**(2B) and **write**(2B), the extra *flags* argument is important. The flags, defined in **<sys/socket.h>**, may be specified as a nonzero value if the following is required:

MSG_OOB	send/receive out-of-band data

Out-of-band data is specific to stream sockets, and we will not immediately consider it.

Discarding Sockets

Once a socket is no longer of interest, it may be discarded by applying a **close**(2) to the following descriptor:

```
close(s);
```

If data is associated with a socket that promises reliable delivery (such as a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should there not be a need for any pending data, a **shutdown**(2B) may be performed on the socket before closing it. This call has the form:

```
shutdown(s, how);
```

How is 0 if the user is no longer interested in receiving data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

Connectionless Sockets

To this point we have been concerned mostly with sockets that follow a connection-oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the **bind**(2B) operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the following **sendto**(2B) primitive is used:

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When an unreliable datagram interface is used, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that cannot be delivered (for instance, when a network is unreachable), the call will return -1 and the global variable *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the **recvfrom**(2B) primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the **connect**(2B) call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at a time; a second connect will change the destination address, and a connect to a null address will disconnect. **connect**(2B) requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end-to-end connection). **accept**(2B) and **listen**(2B) are not used with datagram sockets.

While a datagram socket is connected, errors from recent **send**(2B) calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket. A **select**(2B) for reading or writing will return true when an error indication is received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in the "Advanced Topics" section.

Input/Output Multiplexing

One last facility often used in developing applications is the ability to multiplex I/O requests among multiple sockets and/or files. This is done using the **select**(2B) call:

```
#include <sys/time.h>
#include <sys/types.h>
  ...

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
  ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

select(2B) takes as arguments pointers to three sets. One set is for the set of file descriptors for which the caller wishes to be able to read data on, one is for descriptors to which data is to be written, and one for which exceptional conditions are pending; out-of-band data is the only exceptional condition currently implemented by the socket. If the user is not interested in certain conditions (such as, read, write, or exceptions), the corresponding argument to **select**(2B) should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array is be long enough to hold one bit for each of FD_SETSIZE file descriptors.

The macros FD_SET(*fd*, *&mask*) and FD_CLR(*fd*, *&mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the **select**(2B) call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely. (To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.) **select**(2B) normally returns the number of file descriptors selected; if the **select**(2B) call returns due to the timeout expiring, the value 0 is returned. If the **select**(2B) terminates because of an error or interruption, a -1 is returned with the error number in *errno* and with the file descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the FD_ISSET(*fd*, *&mask*) macro, which returns a nonzero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an **accept**(2B) call, **select**(2B) can be used, followed by a FD_ISSET(*fd*, *&mask*) macro to check for read readiness on the appropriate socket. If FD_ISSET returns a nonzero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, **s1** and **s2**, as it is available from each and with a one-second timeout, the following code might be used:

```
#include <sys/time.h>
#include <sys/types.h>
  . . .
fd_set read_template;
struct timeval wait;
  . . .
        for (;;) {
                wait.tv_sec = 1;                /* one second */
                wait.tv_usec = 0;

                FD_ZERO(&read_template);
                FD_SET(s1, &read_template);
                FD_SET(s2, &read_template);

                nb = select(FD_SETSIZE, &read_template,
                        (fd_set *) 0, (fd_set *) 0, &wait);
                if (nb <= 0) {
                        An error occurred during the select, or
                        the select timed out.
                }

                if (FD_ISSET(s1, &read_template)) {
                        Socket #1 is ready to be read from.
                }

                if (FD_ISSET(s2, &read_template)) {
                        Socket #2 is ready to be read from.
                }
        }
```

select(2B) provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible by using the SIGIO and SIGURG signals described in the "Advanced Topics" section.

Network Library Routines

The discussion in the "basics" section indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. In this section we will consider several routines provided to manipulate network addresses. While the networking facilities support both the DARPA standard Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name intended for human consumption; for example, "the *login server* on host monet." This name and the name of the peer host, must then be translated into network *addresses* that are not necessarily understandable to a user. Finally, the address must then be used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary among network architectures. For instance, it is desirable for a network to not require hosts to be named so that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host when a client host wishes to communicate. This ability to have hosts named in a location-independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file **<netdb.h>**

must be included when using any of these routines.

Host Names

An Internet host name to address mapping is represented by the **hostent** structure:

```
struct hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type (e.g., AF_INET) */
        int     h_length;       /* length of address */
        char    **h_addr_list; /* list of addrs, null terminated */
};

#define h_addr  h_addr_list[0]  /* first addr, network byte order */
```

The routine **gethostbyname**(3B) takes an Internet host name and returns a **hostent** structure, while the routine **gethostbyaddr**(3B) maps Internet host addresses to a **hostent** structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null-terminated list of variable-length addresses. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility and is defined to be the first address in the list of addresses in the **hostent** structure.

The database for these calls is provided by the file **/etc/hosts** (see **hosts**(4)). **gethostbyname**(3B) will return only one address but all listed aliases will be included.

Unlike Internet names, there are no standard routines to map NS names and addresses.

An NS host address is represented by the following:

```
union ns_host {
        u_char  c_host[6];
        u_short s_host[3];
};

union ns_net {
        u_char  c_net[4];
        u_short s_net[2];
};

struct ns_addr {
        union ns_net    x_net;
        union ns_host   x_host;
        u_short x_port;
};
```

The following code fragment inserts a known NS address in an *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
u_long netnum;
struct sockaddr_ns dst;
 ...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a ''union ns_net'' at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
```

continued

```
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

Network Names

For host names, routines for mapping network names to numbers and back are provided. These routines return a **netent** structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
 */
struct netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net address type */
        int     n_net;          /* network no., host byte order */
};
```

The routines **getnetbyname**(3B), **getnetbynumber**(3B), and **getnetent**(3B) are the network counterparts to the host routines described above. The routines extract their information from **/etc/networks**.

Protocol Names

For protocols, which are defined in **/etc/protocols**, the **protoent** structure defines the protocol-name mapping used with the routines **getprotobyname**(3B), **getprotobynumber**(3B), and **getprotoent**(3B):

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

In the NS domain, no protocol database exists; see the "Advanced Topics" section for more information.

Service Names

Information regarding services is more complicated. A service is expected to reside at a specific *port* and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher-level library routines must be bypassed or extended. Services available are contained in the file **/etc/services**. A service mapping is described by the **servent** structure:

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port; /* port number, network byte order */
        char    *s_proto;       /* protocol to use */
};
```

The routine **getservbyname**(3B) maps service names to a **servent** structure by specifying a service name and, optionally, a qualifying protocol. Thus the call

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol. The call

```
sp = getservbyname("telnet", "tcp");
```

returns only the telnet server that uses the TCP protocol. The routines **getservbyport**(3B) and **getservent**(3B) are also provided. The **getservbyport**(3B) routine has an interface similar to that provided by **getservbyname**(3B); an optional protocol name may be specified to qualify lookups.

Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network-independent way. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets, some network dependency will always be in a program. For example, the normal code included in client programs, such as the remote login program, has the form shown in Figure 6-1. (This example will be considered in more detail in the "Client/Server Model" section.)

If we wanted to make the remote login program independent from the Internet protocols and addressing scheme, we would be forced to add a layer of routines that masked the network-dependent aspects from the mainstream login code. For the current facilities available in the system, this does not appear to be worthwhile.

Aside from the address-related database routines, several other routines are available in the run-time library that are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 6-1 summarizes the routines for manipulating variable-length byte strings and handling byte swapping of network addresses and values.

Table 6-1 C Run-Time Routines

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit host quantity to network byte order
htons(val)	convert 16-bit host quantity to network byte order
ntohl(val)	convert 32-bit network quantity to host byte order
ntohs(val)	convert 16-bit network quantity to host byte order

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. Host byte ordering is different than network byte ordering for the CLIPPER architecture. Consequently, programs are required to byte swap quantities. The library routines that return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed, the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines where unneeded, these routines are defined as null macros.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

 . . .

main(argc, argv)
int argc;
char *argv[];
{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;

        . . .

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr,
                        "rlogin: tcp/login: unknown service\n");
                exit(1);
        }

        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr,
                        "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }

        bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr,(char *)&server.sin_addr, hp->h_length);

        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;

        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }

        . . .
```

continued

```
        /* Connect does the bind() for us */

        if (connect(s, (char *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }

        . . .
}
```

Figure 6-1: Remote Login Client Code

Client/Server Model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies asymmetry in establishing communication between the client and server that has been examined in the "Basics" section. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well-known set of conventions before service may be rendered (and accepted). This set of conventions composes a protocol that must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a *client process* and a *server process*. We will first consider the properties of server processes, ant then of client processes.

A server process normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such time, the server process "wakes up" and services the client, performing the appropriate actions the client requests of it.

Alternative schemes that use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers, this scheme has been implemented via **inetd**(1M), the so called "Internet super-server." **inetd**(1M) listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which **inetd**(1M) is listening, **inetd**(1M) executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as **inetd**(1M) played any part in the connection. **inetd**(1M) will be described in more detail in the "Advanced Topics" section.

Servers

Most servers are accessed at well-known Internet addresses or UNIX domain names. For example, the remote login server's main loop of the form shown in Figure 6-2.

```
main(argc, argv)
        int argc;
        char *argv[];
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr,
                        "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        . . .

#ifndef DEBUG
        /* Disassociate server from controlling terminal */
        . . .
#endif

        sin.sin_port = sp->s_port;      /* Restricted port */
        . . .
        f = socket(AF_INET, SOCK_STREAM, 0);
        . . .
        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
                . . .
        }

        . . .

        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
```

continued

```
                                        fprintf(stderr,
                                                "rlogind: accept: %m\n");
                                continue;
                        }

                        if (fork() == 0) {
                                close(f);
                                doit(g, &from);
                        }
                        close(g);
                }
        }
```

Figure 6-2: Remote Login Server

The first step the server takes is to look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the **getservbyname**(3B) call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Step two is to disassociate the server from the controlling terminal of its invoker:

```
setpgrp();
```

This step is important as the server will not likely want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself, it can no longer send reports of errors to a terminal, and must log errors another way.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The **bind**(2B) call is required to ensure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user ID of root. This concept of a "restricted port number" is covered in the "Advanced Topics" section.

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        /*write error message to log file */
                continue;
        }
        if (fork() == 0) {      /* Child */
                close(f);
                doit(g, &from);
        }
        close(g);               /* Parent */
}
```

An **accept**(2B) call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in the next section). Therefore, the return value from **accept(2B)** is checked to ensure a connection has actually been established, and an error may be logged to the log file.

With a connection in hand, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the **accept(2B)** is closed in the parent. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

Clients

The client side of the remote login service was shown earlier in Figure 6-1. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a **gethostbyname**(3B) call:

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start the remote login protocol. The address buffer is cleared, and then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that **connect**(2B) implicitly performs a **bind**(2B) call, since **s** is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
 . . .

if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One in particular is the **rwho**(1) service that provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may learn the current status of a machine with the **ruptime**(1) program. The output generated is illustrated in Figure 6-3.

```
arpa      up    9:45,       5 users, load  1.15,  1.39,  1.31
cad       up    2+12:04,    8 users, load  4.67,  5.13,  4.59
calder    up    10:10,      0 users, load  0.27,  0.15,  0.14
dali      up    2+06:28,    9 users, load  1.04,  1.20,  1.65
degas     up    25+09:48,   0 users, load  1.49,  1.43,  1.41
ear       up    5+00:05,    0 users, load  1.51,  1.54,  1.56
ernie     down  0:24
esvax     down  17:04
ingres    down  0:26
kim       up    3+09:16,    8 users, load  2.03,  2.46,  3.11
matisse   up    3+06:18,    0 users, load  0.03,  0.03,  0.05
medea     up    3+09:39,    2 users, load  0.35,  0.37,  0.50
merlin    down  19+15:37
miro      up    1+07:20,    7 users, load  4.59,  3.28,  2.12
monet     up    1+00:43,    2 users, load  0.22,  0.09,  0.07
oz        down  16:09
statvax   up    2+15:57,    3 users, load  1.52,  1.81,  1.86
ucbvax    up    9:34,       2 users, load  6.08,  5.16,  3.28
```

Figure 6-3: *ruptime* Output

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured in Figure 6-4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to ensure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

```
main()
{
        . . .
        sp = getservbyname("who", "udp");
        . . .
        s = socket(AF_INET, SOCK_DGRAM, 0);

        sin.sin_port = sp->s_port;
        sin.sin_addr.s_addr = INADDR_ANY;
        sin.sin_family = AF_INET;
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        . . .
        sigset(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod),
                        0, (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                fprintf(stderr, "rwhod: recvfrom: %s\n",
                                        sys_errlist[errno]);
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port\n",
                                ntohs(from.sin_port));
                        continue;
                }
                . . .
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr,
                                "rwhod: malformed host name from %x\n",
                                from.sin_addr);
                        continue;
```

continued

```
                }
                (void) sprintf(path, "%s/whod.%s",
                        RWHODIR, wd.wd_hostname);
                whod = open(path,
                        O_WRONLY | O_CREAT, 0644);

                . . .

                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 6-4: *rwho* Server

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematic, however.

Status information must be broadcast on the local network. For networks that do not support the notion of broadcast, another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will not know what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a network are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts that are not (possibly) directly neighbors. If the server is able to support networks that provide a broadcast capability, as well as those that do not, networks with an arbitrary topology may share status information. (One must, however, be concerned about "loops." That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful exchange of information.)

Advanced Topics

A number of facilities have yet to be discussed. For most users of IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features that we consider in this section.

Out-of-Band Data

The stream socket abstraction includes the notion of *out-of-band* data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently from normal data. The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any time. For communications protocols that support only in-band signaling (the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all intervening data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process ID to be informed by the SIGURG signal via the appropriate **fcntl**(2) call, as described below for SIGIO. If multiple sockets may have out-of-band data awaiting delivery, a **select**(2B) call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out of band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushes any pending output from the remote process(es), all data up to the mark in the data stream is discarded.

To send an out-of-band message, the MSG_OOB flag is supplied to a **send**(2B) or **sendto**(2B) call, while to receive out-of-band data MSG_OOB should be indicated when performing a **recvfrom**(2B) or **recv**(2B) call.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a **recv**(2B) is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, in-band data may be in the input buffer so that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data so that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (such as, **telnet**(1)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see **setsockopt**(2B) for usage. With this option, the position of urgent data (the *mark*) is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Receiving multiple urgent indications causes the mark to move, but no out-of-band data is lost.

Nonblocking Sockets

It is occasionally convenient to use sockets that do not block; that is, I/O requests that cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the **socket**(2B) call, it may be marked as nonblocking by **fcntl**(2B) as follows:

```
#include <fcntl.h>
#include <file.h>
 ...
int     s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
 ...
```

When performing nonblocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as nonblocking. In particular, **accept**(2B), **connect**(2B), **send**(2B), **recv**(2B), **read**(2), and **write**(2) can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a **send**(2B) cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

Interrupt Driven Socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket has data waiting to be read. Using the SIGIO facility requires three steps. First, the process must set up a SIGIO signal handler by using the **signal**(2) or **sigset**(2) calls. Second, it must set the process ID or process group ID to receive notification of pending input to its own process id, or the process group ID of its process group. (Note that the default process group of a socket is group zero.) This is accomplished by using an **fcntl**(2) call. Third, it must enable asynchronous notification of pending I/O requests with another **fcntl**(2) call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket s is given in Figure 6-5. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
#include <file.h>
 ...
int     io_handler();
 ...
sigset(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

Figure 6-5: Use of Asynchronous Notification of I/O requests

Signals and Process Groups

Due to the existence of the SIGURG and SIGIO signals, each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but may be redefined at a later time with the F_SETOWN **fcntl**(2), such as was done in the code above for SIGIO. To set the socket's process ID for signals, positive arguments should be given to the **fcntl**(2) call. To set the socket's process group for signals, negative arguments should be passed to **fcntl**(2). Note that the process number indicates either the associated process ID or the associated process group; it is impossible to specify both at the same time. A similar **fcntl(2)**, F_GETOWN, is available for determining the current process number of a socket.

Another signal that is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed states. Normally servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 6-2 may be augmented as shown in Figure 6-6.

```
int reaper();
 . . .

        sigset(SIGCHLD, reaper);
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *)&from, &len,);
                if (g < 0) {
                        if (errno != EINTR)
                                /* write error message to log file */
                        continue;
                }
                . . .
        }
         . . .

#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Figure 6-6: Use of the SIGCHLD Signal

If the parent server process fails to reap its children, a large number of "zombie" processes may be created.

Pseudo Terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal.* A pseudo-terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side is processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal controls the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. This abstraction preserves terminal semantics over a network connection; that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt on the remote machine that flushes terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under CLIX, the name of the slave side of a pseudo-terminal is of the form **/dev/tty***xy*, where *x* is a single letter starting at 'p' and continuing to 'r'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is **/dev/pty***xy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened and set to the proper terminal modes if necessary.

The process then **forks**; the child closes the master side of the pseudo-terminal and **execs** the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code using of pseudo-terminals is given in Figure 6-7; this code assumes that a connection on a socket **s** exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

```
gotpty = 0;

for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;

        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] =
                        "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}

if (!gotpty) {
        /* write error to log file */
        exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);     /* now slave side */
if (slave < 0) {
        /* write error to log file */
        exit(1);
}

ioctl(slave, TCGETA, &b);       /* Set slave tty modes */
/* set any modes */
ioctl(slave, TCSETA, &b);

i = fork();
```

continued

```
if (i < 0) {
        /* write error to log file */
        exit(1);
} else if (i) {                    /* Parent */
        close(slave);
        . . .
} else {                           /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);

        . . .

}
```

Figure 6-7: Creation and Use of a Pseudo Terminal

Selecting Specific Protocols

If the third argument to the **socket**(2B) call is 0, **socket**(2B) will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in the "Network Library Routines" section, such as **getprotobyname**(3B):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
  . . .
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket **s** using a stream based connection, but with protocol type "newtcp" instead of the default "tcp."

Address Binding

Binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the **bind**(2B) system call, a process may specify half of an association, the <local address, local port> part, while the **connect**(2B) and **accept**(2B) primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps, the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain, the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address." For example, to bind a specific port number to a socket but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 . . .
struct sockaddr_in sin;
 . . .
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests addressed to 128.32.0.4 or 10.0.0.78. If a server process wished to only allow hosts on a given network to connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero). In this case, the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
        . . .
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that Internet ports below IPPORT_RESERVED (1024) (for the XNS domain, 0 through 3000) are reserved for privileged users (the super-user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for nonprivileged servers. The second is that the port number is not currently bound to some other socket. To find a free Internet port number in the privileged range, the **rresvport**(3B) library routine may be used as follows to return a stream socket in with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
...
s = rresvport(&lport);
if (s < 0) {
        if (errno == EAGAIN)
                fprintf(stderr, "socket: all ports in use\n");
        else
                perror("rresvport: socket");
        ...
}
```

The restriction on allocating ports was done to allow processes executing in a "secure" environment to perform authentication based on the originating address and port number. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the **accept**(2B) call or from the **getpeername**(2B) call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two-step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
 . . .
int     on = 1;
 . . .
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses which are already in use may be bound . This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

Broadcasting and Determining Network Configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, a datagram socket should be created:

```
        s = socket(AF_INET, SOCK_DGRAM, 0);
```

or

```
        s = socket(AF_NS, SOCK_DGRAM, 0);
```

and a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = sp->s_port;
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);

/* insert net number */
sns.sns_addr.x_net = *(union ns_net *) &netnum;
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in **<netinet/in.h>**. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected.

With the appropriate broadcast or destination address, the **sendto**(2B) call may be used:

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
```

A **sendto**(2B) may be done for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing.

Received broadcast messages contain the sender's address and port, as datagram sockets are bound before a message is allowed to go out.

Socket Options

It is possible to set and get a number of options on sockets via the **setsockopt**(2B) and **getsockopt**(2B) system calls. The general forms of the calls are as follows:

```
        setsockopt(s, level, optname, optval, optlen);
```

and

```
        getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level," indicated by the symbolic constant SOL_SOCKET, defined in **<sys/socket.h>**. The actual option is specified in *optname* and is a symbolic constant also defined in **<sys/socket.h>**. *Optval* and *optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For **getsockopt**(2B), *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (such as, stream or datagram) of an existing socket; programs under **inetd**(1M) (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the **getsockopt**(2B) call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
        ...
}
```

After the **getsockopt(2B)** call, *type* will be set to the value of the socket type, as defined in **<sys/socket.h>**. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

Inetd

One of the daemons provided is **inetd(1M)**, the so-called "Internet super-server." **inetd(1M)** is invoked at boot time, and determines from the file **/etc/inetd.conf** the servers for which it is to listen. Once this information has been read and a pristine environment created, **inetd(1M)** proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

inetd(1M) then performs a **select(2B)** on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. **inetd(1M)** then performs an **accept(2B)** on the socket in question, **fork(2)**s, **dup(2)**s the new socket to file descriptors 0 and 1 (**stdin** and **stdout**), closes other open file descriptors, and **exec(2)**s the appropriate server.

Servers using **inetd(1M)** are considerably simplified, as **inetd(1M)** takes care of the majority of the IPC work required in establishing a connection. The server invoked by **inetd(1M)** expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as **read(2)**, **write(2)**, **send(2B)**, or **recv(2B)**. Indeed, servers may use buffered I/O as provided by the "stdio" conventions, as long as as they remember to use **fflush**(3S) when appropriate.

One call that may be of interest to individuals writing servers under **inetd**(1M) is the **getpeername**(2B) call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" ("128.32.0.4") of a client connected to a server under **inetd**(1M), the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
        fprintf(stderr, "getpeername: %m\n");
        exit(1);
} else
        fprintf(stderr, "Connection from %s\n",
                inet_ntoa(name.sin_addr));
 ...
```

While the **getpeername**(2B) call is especially useful when writing programs to run with **inetd**(1M), it can be used under other circumstances. Be warned, however, that **getpeername**(2B) will fail on UNIX domain sockets.

Chapter 7: NQS Tutorial

Displaying Status Information

You can display a variety of status information about NQS queues, devices, and configuration. There are two ways to display status information in NQS: by using the show commands available through **qmgr**(1M) or using the **qstat**(1) and **qdev**(1)commands from the CLIX command line. Any user can obtain status information; no special privileges are necessary.

This chapter is divided as follows:

Using the Queue Manager (qmgr) to Display Queue Status Information

The **qmgr**(1M) utility provides nine show commands that allow you to display various levels of status information about queues and the requests in them. The show commands are detailed in the following sections.

NOTE The show commands must be executed from the Mgr: prompt. The Mgr: prompt indicates that you are in the qmgr(1M) environment. To access the Mgr: prompt, key in qmgr at the system prompt as follows:

```
$ qmgr
Mgr:
```

The show all Command

The **show all** command displays all NQS configuration information including status information about all NQS queues, queue complexes, devices, managers, forms, parameters, and defaults and limits.

Key in the **show all** command at the Mgr: prompt as follows:

```
Mgr: show all
```

The parameters set for your NQS configuration display. The output will be similar to the following:

```
Queues:

  sysbatch@lga;  type=BATCH;  [ENABLED, INACTIVE]; pri=16
    0 exit; 0 run; 0 stage; 0 queued; 0 wait; 0 hold; 0 arrive;

  syspipe@lga;  type=PIPE;  [ENABLED, INACTIVE]; pri=16
    0 depart; 0 route; 0 queued; 0 wait; 0 hold; 0 arrive;

  sysprint@lga;  type=BATCH;  [ENABLED, INACTIVE]; pri=16
    0 run; 0 queued; 0 wait; 0 hold; 0 arrive;

Complexes:

  batchcom@lga
  Run_limit = 1
  Queues = [sysbatch];

Devices:

  epson@lga
  Fullname: /dev/tty00
  Server: /usr/lib/nqs/devserver
  Forms: sysform
  Status = [ENABLED, INACTIVE]

Managers:

  root:m
  laura:m

Forms:

  sysform
  shortform
  longform

NQS Operating Paramaters:

  Debug level = 1
  Default batch_request priority = 31
  Default batch_request queue = sysbatch
  Default destination_retry time = 72 hours
  Default destination_retry wait = 5 minutes
  Default device_request priority = 31
```

continued

```
  No default print forms
  Default print queue = NONE
  (Pipe queue request) Lifetime = 168 hours
  Log_file = /usr/lib/nqs/logfile
  Mail account = root
  Maximum number of print copies = 2
  Maximum failed devices open retry limit = 2
  Maximum print file size = 1000000 bytes
  Netdaemon = /usr/lib/nqs/netdaemon
  Netclient = /usr/lib/nqs/netclient
  Netserver = /usr/libnqs/netserver
  (Failed device) Open_wait time = 5 seconds
  NQS daemon is not locked in memory
  Next available sequence number = 4
  Batch request shell choice strategy = FREE

Limits supported:

  Per-process permanent file size limit (-lf)
  Nice value (-ln)
```

Parameter Class Descriptions

- The Queues are the NQS queues that exist on the local node.
- The Complexes are all queue complexes (queue groups) that exist on the local node. Queue complexes are queues joined for the purpose of limiting the number of jobs that the group will execute simultaneously.
- The Devices are the NQS devices that exist on the local node.
- The Managers are the users that have NQS manager and operator privileges on the local node.

- The NQS Operating Parameters are the existing defaults and limits set on the local node.
- The Limits Supported are the limit defaults that have been set for NQS.

The show complex Command

The **show complex** command displays status information for one or all NQS queue complexes. A queue complex is a set of one or more queues grouped together to limit the number of jobs that the queue group can service at a given time. Queue complexes are discussed in the "Configuring Queues From the Command Line" section in the System Administrator's "NQS Tutorial."

Key in the **show complex** command at the Mgr: prompt as follows. If you do not specify a complex name, information on all complexes displays. (If no queue complexes have been created, none will display.)

Mgr: **show complex** [*complex_name*]

Information similar to the following displays:

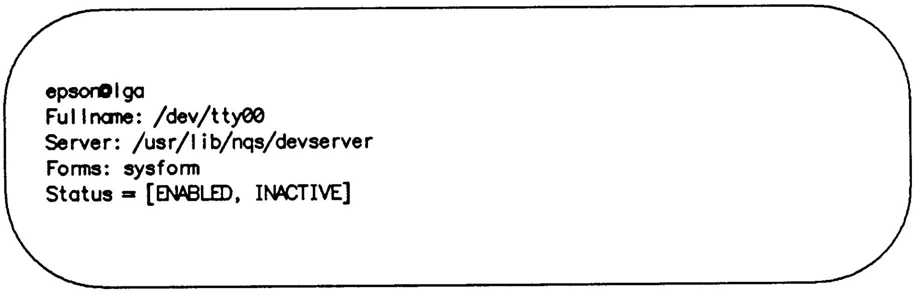
```
batch@olga
Run_limit = 1
Queues = {sysbatch};
```

The show device Command

The **show device** command displays general status information for one or all NQS devices. NQS devices are the interfaces to printers and plotters that produce the hardcopy output requested by NQS. See the "Manipulating Devices From the Command Line" section in the System Administrator's "NQS Tutorial"for information about devices. Key in the **show device** command at the Mgr: prompt as follows. If you do not specify a device name, information for all NQS devices defined on the local node displays. (If no devices have been created, none will display.)

Mgr: **show device** [*device_name*]

Information similar to the following displays:



```
epson@lga
Fullname: /dev/tty00
Server: /usr/lib/nqs/devserver
Forms: sysform
Status = [ENABLED, INACTIVE]
```

The show forms Command

The **show forms** command displays the forms in the device forms list. Forms are discussed in the "NQS Concepts" section of the System Administrator's "NQS Tutorial".

Key in the **show forms** command at the Mgr: prompt as follows:

Mgr: **show forms**

NQS displays the forms you have defined as follows:

```
sysform
shortform
longform
```

The show limits_supported Command

The **show limits_supported** command displays the set of request and process limits. You can set limits for individual batch queues from within qmgr(1M). Setting limits is discussed in the "Configuring Queues From the Command Line" section in the System Administrator's "NQS Tutorial."

Key in the **show limits_supported** command at the Mgr: prompt as follows:

Mgr: **show limits_supported**

Information similar to the following displays. (The limits shown below are the system defaults.)

```
Per-process permanent file size limit (-lf)
Nice value (-ln)
```

The show long queue Command

The **show long queue** command displays detailed information about all queues or a specific queue.

Key in the **show long queue** command at the manager prompt as follows. If you do not specify a queue name, information about all local queues displays. (If no queues have been created, then none display.)

Mgr: **show long queue** [*queue_name*]

Information similar to the following displays:

```
sysbatch@lga ;  type=BATCH; [ENABLED, INACTIVE]; pri=16
0 exit;  0 run;  0 stage;  0 queued;  0 wait;  0 hold;  0 arrive;
Cumulative system space time = 0.00 seconds
Cumulative user space time = 0.00 seconds
Unrestricted access
Per-process permanent file size limit = 1 megabytes <DEFAULT>
Per-process execution nice value = 0 <DEFAULT>

syspipe@lga ;  type=PIPE; [ENABLED, INACTIVE]; pri=16
0 depart; 0 route; 0 queued;  0 wait;  0 hold;  0 arrive;
Cumulative system space time = 1.91 seconds
Cumulative user space time = .85 seconds
Unrestricted access
Queue server: /usr/lib/nqs/pipeclient
Destset = {epson@britt};
```

The show managers Command

The **show managers** command displays the accounts included in the NQS manager list. This list includes users with manager and operator privileges. Privileges are discussed in the "NQS Privileges" section of the System Administrator's "NQS Tutorial."

Key in the **show managers** command at the manager prompt as follows:

Mgr: **show managers**

Information similar to the following displays. (The :m extension indicates a manager; the :o extension indicates an operator.)

```
root:m
laura:m
diane:o
```

The show parameters Command

The **show parameters** command displays the current values for NQS operating parameters. (The "Configuring Queues From the Command Line" section in the System Administrator's "NQS Tutorial" discusses how to change parameter values.)

Key in the **show parameters** command at the Mgr: prompt as follows:

```
Mgr: show parameters
```

Information similar to the following displays:

```
Debug level = 1
Default batch_request priority = 31
Default batch_request queue = sysbatch
Default destination_retry time = 72 hours
Default destination_retry wait = 5 minutes
Default device_request priority = 31
No default print forms
Default print queue = NONE
(Pipe queue request) Lifetime = 168 hours
Log_file = /usr/lib/nqs/logfile
Mail account = root
Maximum number of print copies = 2
Maximum failed devices open retry limit = 2
Maximum print file size = 1000000 bytes
Netdaemon = /usr/lib/nqs/netdaemon
Netclient = /usr/lib/nqs/netclient
Netserver = /usr/libnqs/netserver
(Failed device) Open_wait time = 5 seconds
NQS daemon is not locked in memory
Next available sequence number = 4
Batch request shell choice strategy = FREE
```

The show queue Command

The **show queue** command displays general queue status information. Key in the **show queue command** at the manager prompt as follows:

Mgr: **show queue**

Information similar to the following displays:

```
sysbatch@lga;  type=BATCH;  [ENABLED, INACTIVE]; pri=16
  0 exit; 0 run; 0 stage; 0 queued; 0 wait; 0 hold; 0 arrive;

syspipe@lga;  type=PIPE;  [ENABLED, INACTIVE]; pri=16
  0 depart; 0 route; 0 queued; 0 wait; 0 hold; 0 arrive;

sysprint@lga;  type=DEVICE;  [ENABLED, INACTIVE]; pri=16
  0 run; 0 queued; 0 wait; 0 hold; 0 arrive;

sysplot@bob;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
  0 run; 3 queued; 2 wait; 0 hold; 0 arrive;

  Request Name  Request ID Usr      Pri     State     Size

<3 jobs queued>
1:   JOB1          154.bob   bob      31      WAITING   766
2:   JOB2          155.bob   bob      31      WAITING   531
```

Using the qstat Command to Display Queue Status Information

The **qstat**(1) command displays the status of one or more queues and the requests in those queues. The **qstat**(1) command provides access to the same type of information as the **show queue** command provides access to.

To display queue status information, key in the **qstat**(1) command as follows:

> **qstat** [**-a**] [**-l**] [**-m**] [**-u** *username*] [**-x**] [**-b**] [**-d**]
> [**-p**] [**-r**] [*queue_name* ...] [*queue@host* ..]

Keying in **qstat** with no options displays information about jobs that you own in all local queues. You may also display information about individual queues on multiple nodes. For example, the following command line displays information about the local queue **sysbatch**, the queue **sysprint** at node **red**, and all queues at node **blue**:

> **qstat sysbatch sysprint@red @blue**

The **qstat**(1) options are listed in the following table and described in detail in the following sections.

Option	Description
-a	Display all requests
-l	Display requests in long format
-m	Display requests in medium-length format
-u *username*	Display only requests belonging to the specified user
-x	Display the queue header in extended format
-b	Display only batch queues
-d	Display only device queues
-p	Display only pipe queues
-r	Recursively display pipe queue destinations

Displaying Requests Using -a

The **-a** option displays a line of information about your requests in the specified queue or all queues. The information provided about the requests includes the request name, request ID, originating user, the request's intraqueue priority, the request's execution state, and its size in bytes (or its process group for batch queues). Requests that were not submitted under your user name (for example, screen copies submitted by root through the workstation pull-down menu) are listed in angle brackets with no job information.

For example, keying in the following command displays information about all requests in the queue **sysbatch.**

qstat -a sysbatch

Information similar to the following displays:

```
    sysbatch@lga;  type=BATCH;  [ENABLED, ACTIVE]; pri=16
     0 exit; 1 run; 0 stage; 0 queued; 1 wait; 0 hold; 0 arrive;

    Request Name  Request ID Usr      Pri      State      Pgrp

1:    STDIN         154.lga   sue      31       RUNNING    411
2:    MYJOB         155.lga   jim      31       WAITING    421
```

- The Request Name field contains the name that NQS assigned to the request.
- The Request ID field contains the identification number that NQS assigned to the request.
- The Usr field contains the name of the originating user.
- The Pri field contains the request's privilege level.
- The State field contains the request's execution state.
- The Size field contains the request's size in bytes.

Displaying Requests Using -l

The -l option displays all information available about the requests in the queue. For example, keying in the following displays all information about the requests in the queue **syspipe:**

qstat -l syspipe

Information similar to the following displays:

```
syspipe@lga;  type=PIPE;  [ENABLED, ACTIVE]; pri=16
  0 depart; 0 route; 0 queued; 1 wait; 0 hold; 0 arrive;

Request  1: Name=STDIN  Id=155.lga
  Owner=fred  Priority=31  WAITING Sat Mar 24 00:00:00 CDT 1989
Creafred at Fri Mar 23 14:00:17 CDT 1989
Mail = [END]
Mail address=fred@lga
Owner user name at originating machine=fred
Per-proc permanent file size limit=[1MB, 1MB]
  <DEFAULT>
Per-proc execution nice priority=0 <DEFAULT>
Standard-error access mode=SPOOL
Standard-error name = lga:/usr/fred/STDIN.e155
Standard-output access mode = SPOOL
Standard-output name = lga:/usr/fred/STDIN.o155
Shell = DEFAULT
Umask = 2
```

Displaying Limited Information Using -m

The **-m** option displays a subset of the available information about jobs in the queues. For example, keying in the following command displays a subset of information about the queues **sysprint** and **syspipe**:

qstat -m sysprint syspipe

Information similar to the following displays:

```
sysprint@lga;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
  1 run; 0 queued; 0 wait; 0 hold; 0 arrive;

Request  1: Name=STDIN  Id=155.lga
  Owner-fred  Priority=31  RUNNING  Sat May 24 00:00:00 CDT 1989

syspipe@lga;  type=PIPE;  [ENABLED, ACTIVE]; pri=16
  0 depart; 0 route; 0 queued; 1 wait; 0 hold; 0 arrive;

Request  1: Name=myprint  Id=25.lga
  Owner-fred  Priority=31  WAITING  Sat May 24 00.00.00 CDT 1989
```

Displaying a User's Requests Using -u

The **-u** option allows you to display a specific user's requests in the queue. For example, keying in the following command displays the default information for **bob**'s requests in queue **sysprint** on node **bob**:

qstat @bob -u bob sysprint

Information similar to the following displays:

```
  sysprint@bob;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
    0 exit; 1 run; 0 stage; 0 queued; 2 wait; 0 hold; 0 arrive;

   Request Name  Request ID Usr      Pri      State      Size

<1 job running>
1:    JOB1          154.bob   bob      31       WAITING    766
2:    JOB2          155.bob   bob      31       WAITING    531
```

Displaying the Queue Header Using -x

The -x option displays queue header information in an extended format. Use this option to find out detailed information about the queue rather than about the requests in it. For example, keying in the following command displays information about queue **sysprint**:

```
$ qstat -x sysprint

  sysprint@bob;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
    0 run; 0 queued; 2 wait; 0 hold; 0 arrive;
  Run_limit = 1;
  Cumulative system space time = 0.60
  Cumulative user space time = 0.25 seconds
  Unrestricted access
  Devset = {sysprint};
```

Displaying Batch Queues Using -b

The -b option displays batch queues. For example, keying in the following displays requests in all batch queues:

qstat -b

Displaying Device Queues Using -d

The -d option displays device queues. For example, keying in the following displays requests in all device queues:

qstat -d

Displaying Pipe Queues Using -p

The **-p** option displays pipe queues. For example, keying in the following displays requests in all pipe queues:

```
qstat -p
```

Recursively Displaying Pipe Queues Using -r

The **-r** option allows the user to recursively follow pipe queue destinations. That is, if pipe queue **a** sends jobs to pipe queue **b**, which sends jobs to device queue **c**, the **-r** option will display queues **a**, **b**, and **c**.

For example, keying in the following displays all device queues on the local machine and all device queues on the end of local pipe queues. This form of the **qstat**(1) command will not display intervening pipe queues.

```
qstat -d -r
```

The results of the **-r** option can become long if no other options are used because all queues from all local and remote devices will be displayed. Therefore, it is best to use the **-r** option with one of the following options: **-b**, **-d**, or **-p**.

Using the qdev Command to Display Device Information

The **qdev**(1) command is used to display information about devices on local and remote nodes. Key in the **qdev**(1) command as follows:

qdev [*device_name device_name* ...] [*device_name*] [@*host* ...]

If you do not specify a *device_name*, information about all local devices displays. For example, the **qdev**(1) command may return information similar to the following:

```
epson@mynode
Fullname: /dev/tty00
Server: /usr/lib/nqs/devserver /usr/lib/nqs/config_files/epson
Forms: sysform
Status = [ENABLED, INACTIVE];

il2217@mynode
Fullname: /dev/cop
Server: /usr/lib/nqs/devserver /usr/lib/nqs/config_files/il2217
Forms: land10
Status = [ENABLED, ACTIVE];
```

You can also display the status of remote devices on multiple nodes. However, you must have access to these nodes. For example, the following command line displays the status of the **ilp811** device on the local node and the **il2217** device on node **mac**.

```
qdev ilp811 il2217@mac
```

Submitting and Manipulating NQS Requests

NQS provides these commands to submit and delete print, plot, and batch requests:

- **qpr**(1) submits print and plot requests.
- **qsub**(1) submits batch requests.
- **qdel**(1) deletes any NQS request from a queue.

This section contains the following major sections:

Submitting Print Requests

Print requests are spooled to an output device such as a printer or plotter. Devices may be connected to your node or to a node at a remote location on the network. Use the **qpr**(1) command to submit print requests.

Key in the **qpr**(1) command at the system prompt as follows:

qpr -q *queue_name* [**-a** *date_time*] [**-d** *name=value*] [**-e** *tag=filename*]
    [**-f** *form_name*] [**-l** "*log_message*"] [**-mb**] [**-me**] [**-mu** *username*]
    [**-n** *copies*] [**-o** "*filter_options*"] [**-p** *priority*] [**-r** *request_name*] [**-R**]
    [**-s**] [**-t** *input_type*] [**-x**] [**-z**] *request_files*

The **qpr**(1) options are listed in the following table and discussed in the following sections. If no request files are specified, the standard input (keyboard) is used.

Option	Description
-a *date_time*	Executes the request after the specified time.

-d *name=value*	Defines an environment variable to be placed in the devserver environment.
-e *tag=filename*	Associates the tag with the file name and places it on the server node.
-f *form*	Prints the request on the stated form.
-l	Logs the specified message in the accounting file if an accounting file is defined in the devicecap file.
-mb	Sends mail when the request begins executing.
-me	Sends mail when the request finishes executing.
-mu *username*	Sends mail about the request to the specified user.
-n *copies*	Prints the specified number of copies.
-o *"filter_options"*	Places the specified options at the end of the filter's argument list. Enclose the options in double quotes ("").
-p *priority*	Specifies the intraqueue request priority.
-q *queue*	Submits the request to the specified queue.
-r *request_name*	Assigns the specified name to the request.
-R	Specifies that original files will be deleted after a request is spooled or transported.
-s	Specifies that the file will be symbolically linked to the spool directory instead of being copied to it.
-t *input_type*	Specifies the input type being sent.
-x	Copies the complete environment to the server node.
-z	Submits the request silently.

Specifying a Queue Using -q

The **-q** parameter is the only required parameter besides the *request_file*. The **-q** parameter specifies the queue that the request will be submitted to. Using the **-q** parameter followed by the name of a pipe queue, you can submit requests to another workstation or server. Using the **-q** parameter followed by the name of a device queue, you can submit requests to a device connected to your machine. (You may set up a default queue. See the "Manipulating Devices from the Command Line" section in the System Administrator's "NQS Tutorial.")

For example, the following command line submits a request to the local pipe queue, **syspipe. Syspipe** then routes the request to its destination queue on a remote node.

```
qpr -q syspipe print_file
```

Delaying Execution Using -a

The -**a** option specifies a date and time after which NQS will execute a request. Use the -**a** option if you wish to delay request execution. The following examples illustrate acceptable formats for the -**a** option:

```
qpr -a "July, 4, 1989 9:00-EST" print_file

qpr -a "04-Jul-1989 9am, EST" print_file

qpr -a "Wednesday, 09:00:00 EST" print_file

qpr -a "9am tues." print_file
```

Defining Environment Variables Using -d

The -**d** option allows you to define environment variables that will be placed in the server's environment with the request.

Example:

Including the following on the **qpr**(1) command line sets the environment variable MYNAME to **tom.**

```
qpr -d MYNAME=tom print_file
```

Exporting a Tagged File Name Using -e

The -**e** option allows you to associate a tag with a filename and export that filename to the server by placing the tag in the server's environment. The server can then access the tagged file by looking at the tag's environment variable.

Example:

Including the following on the **qpr**(1) command line sends the tag REND_ATT to the server. This is useful for sending attribute information with an InterPlot plot request. (See your *InterPlot User's Guide.*)

```
qpr -e REND_ATT=/usr/tim/myattr.ra print_file
```

Specifying a Form Using -f

The **-f** option specifies a form that the request will print on. If no form is specified, the default form will be used. If no default form exists, the request will print on any form in the device. (The "Understanding Forms" section of the System Administrator's "NQS Tutorial" discusses forms.)

Example:

Include the following on the **qpr**(1) command line specifies that the print file will be printed on a form called **longform.**

```
qpr -f longform print_file
```

Logging a Message Using -l

The **-l** option allows you to place a log message in the accounting file if an accounting file is defined in your devicecap file. (See the "Devicecap File Format" section of the System Administrator's "NQS Tutorial.") Log messages are useful for tracking information about submitted jobs. For example, the following command places the message "Part Number 3600" in your accounting file:

```
qpr -l "Part Number 3600" print_file
```

Notifying the Request Originator Using -mb, -me, and -mu

NQS can send mail to the request originator (or another user) when a request begins or finishes printing. (Mail is automatically sent to the request originator if the request fails.)

NOTE If you are the super-user when you submit a request, NQS will send your mail to the root account.

- The **-mb** (mail begin) option tells NQS to send mail to the request originator when the request begins printing. Use the **-mb** option as follows:

  qpr -mb *print_file*

- The **-me** (mail end) option tells NQS to send mail to the request originator when the request finishes printing. Use the **-me** option as follows:

  qpr -me *print_file*

- The **-mu** (mail user) option tells NQS to send mail to a user other than the originator. NQS will send mail when the request finishes printing unless you also specify **-mb**. Use the **-mu** option as shown in the examples below

- The following command sends mail to user **joe** when the request finishes printing:

  qpr -mu joe *print_file*

- The following command sends mail to user **sue** when the request begins printing:

  qpr -mb -mu sue *print_file*

Selecting the Number of Copies Using -n

The **-n** option specifies how many request copies will print. The following command will print two copies of the print file. (NQS sets a default maximum number of copies to two. To change the default, see the "Setting Defaults" section in the System Administrator's "NQS Tutorial."

qpr -n 2 *print_file*

Appending Filter Arguments Using -o

The -o option appends the specified arguments to the filter specification before it executes. This option is useful for the application programmer who has developed custom filter files. Using the -o option, you can add to or modify the filter specification without needing to modify the devicecap file.

For example, the following line may appear in the devicecap file:

```
epson:of=/usr/lib/myfilter -l66 -w132:
```

You can append another option specific to your filter by submitting the request with the -o option. For example, if you have defined **-f** as a footer (in your application) and want the footer to print for this job, include the -o option in your command line as follows:

qpr -o "-f" *print_file*

The filter specification for this job becomes

```
/usr/lib/myfilter -l66 -w132 -f
```

Setting Scheduling Priorities Using -p

The **-p** option defines an intraqueue priority value (the relative order of requests in a queue). The priority value is a number from 0 to 63. Requests with higher scheduling priority values are positioned first in the queue. If the **-p** option is not used, the default priority value for the system is used. (See the "Setting Defaults" section in the System Administrator's "NQS Tutorial.")

NOTE

The scheduling priority does not determine the request's execution priority; it determines only the relative ordering of requests in a queue.

The following command submits a request with an intraqueue priority of **20**:

qpr -p 20 *print_file*

Specifying a Request Name Using -r

The **-r** parameter specifies the name to be assigned to a request file. For example, the following command specifies that the request will be named **barb**:

qsub -r barb *print_file*

The name of the request (**barb** in this example) is the name that will appear in all status displays and compose part of the output and error log file names. (Names beginning with a number will be prefixed with the letter "r." For example, the name 210 would become r210.)

Deleting Files After Spool or Transport Using -R

The **-R** option specifies that original files be deleted after a request is spooled or transported. This option is similar to the **-r** option of **lpr**(1).

If a request is actually spooled, all the files associated with the request, including the ancillary files, are deleted if the user has privilege to delete them. If the user does not have delete privilege, a message is displayed to that effect.

If the request also specifies the **-s** option to symbolically link the files into the spool directory, the files are not deleted until the request has printed or has been transported to another machine. In this case, the user does not actually know if the file was successfully deleted; however, a message is printed to the console if the user does not have the privilege to delete the request.

If a request is deleted before it has been printed or transported, the files will also be deleted.

Specifying a Symbolic Link Using -s

The **-s** option allows you to symbolically link your print file to the NQS spool directory instead of copying it there. This is useful when large print files are too large to fit in the spool directory or are spooling slowly.

Use the **-s** option as follows:

qpr **-s** *print_file*

CAUTION Do not modify or delete a print file until it is finished printing. The file and its symbolic link are one; any changes made to the original file affect the symbolic link. Do not move a symbolically linked file, as the symbolic link is only a pointer to the file and moving the file will disable the link.

Specifying the Input Type Using -t

The **-t** option specifies the input type being sent to the queue. The input type may be any input type that the device that will service the request supports. When printing to an ASCII printer such as an Epson or Printronix, the input type will be ascii or text. For input types associated with individual metafile interpreters, see your *InterPlot User's Guide*.

Copying the Environment to the Server Using -x

The **-x** option behaves the way the **-d** option does; it allows you to define environment variables that will be placed in the server's environment with the request. However, using the **-x** option copies the entire environment to the server machine.

The automatically exported variables are saved as QPR_HOST, QPR_REQNAME, QPR_REQID, and QPR_QUEUE when the request is submitted. If you do not use the **-x** option, no other environment variables will be exported for the device request from the originating host.

For example, the following command will export all environment variables from the originating user's machine to be used for printing the file:

qpr **-x** *print_file*

Submitting a Request Silently Using -z

The **-z** option allows you to submit a request without a message similar to the following displaying on your screen:

```
Request 24.red submitted to queue sysbatch
```

For example, the following command will silently submit the request to the default queue:

```
qpr -z print_file
```

qpr Examples

The following sections illustrate possible printing scenarios and the **qpr**(1) command lines that you would use to submit requests in these situations.

Example 1. Submitting an ASCII Print File

In this example, user **Anna** wants to submit an ASCII file named **anna.txt** from her node, **anna**, to a dot-matrix printer attached to a server node. The server node is named **red** and is located across the hall from her office. The following conditions exist:

- The server's administrator used **pconfig** to create a device queue named **dot_mat** to the dot-matrix printer. He has set the default data type for queue **dot_mat** to ASCII.
- All users have access to the **rje** account on node **red**.
- Anna used **pconfig** to create a pipe queue named **to_red** on her client node. She has set the destination queue to **dot_mat** on node **red**.
- Both the device and pipe queues are enabled and running, and the printer is online.

To send her file to be printed at server **red**, Anna follows these steps:

1. Anna keys in the following command line:

   ```
   qpr -q to_red ann.txt
   ```

   After several seconds, the following message displays:

   ```
   Request 18.anna submitted to queue: to_red.
   ```

2. Anna checks her local pipe queue, **to_red**, to ensure that the request is queued by keying in the following:

   ```
   qstat to_red
   ```

The following queue information displays:

```
 to_red@anna;  type=PIPE;  [ENABLED, ACTIVE]; pri=16
   0 depart; 1 route; 0 queued; 0 wait; 0 hold; 0 arrive;

  Request Name  Request ID Usr      Pri     State     PGRP

1: anna.txt      18.anna    anna     31      Routing
```

3. Anna waits several seconds and then checks the server node to ensure that her job arrived. She keys in the following:

   qstat @red

   The following queue information displays:

```
dot_mat@red;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
   1 run; 0 queued; 0 wait; 0 hold; 0 arrive;

  Request Name  Request ID Usr      Pri     State     SIZE

1: anna.txt      18.anna    rje      31      Running   687
```

4. Anna sees that her request is running and retrieves it from the dot-matrix printer at server **red**.

Example 2. Submitting a Plot Request

In this example, user **Bill** wants to submit a postscript file named **rasfile** from his node, **bill**, to an ILP2217 laser printer attached to a server node. He wants to print three copies and receive mail when the request has finished executing. The following conditions exist:

- The server's administrator used **pconfig** to create a device queue named **il2217** to the ILP2217 printer. He has set the default data type for queue **il2217** to igds. (Because the default is not script (the postscript data type), Bill will need to specify the data type in the **qpr**(1) command line.)
- The IP_Script metafile interpreter is installed on the server node, **blue**.
- All users have access to the **rje** account on node **blue**.
- Bill used **pconfig** to create a pipe queue named **to_2217** on his client node. He has set the destination queue to **il2217** on node **blue**.
- Both the device and pipe queues are enabled and running, and the printer is online.

To send his file to be plotted on server **blue**, Bill follows these steps:

1. Bill keys in the following command line:

   qpr -q -t script to_il2217 -me -n 3 rasfile

   After several seconds, the following message displays:

   ```
   Request 20.bill submitted to queue: to_il2217
   ```

2. Bill checks his pipe queue, **to_il2217**, to ensure that the request is queued by keying in the following:

   qstat to_2217

   The following queue information displays:

```
to_2217@bill;  type=PIPE;  [ENABLED, ACTIVE]; pri=16
  0 depart; 1 run; 1 route; 0 queued; 0 wait; 0 hold; 0 arrive;

  Request Name  Request ID Usr      Pri     State     PGRP

1: rasfile       18.bill    bill     31      Routing
```

3. Bill waits several seconds and then checks the server node to ensure that his job arrived. He keys in the following:

   qstat @blue

   The following queue information displays:

```
IL2217@blue;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
  1 run; 1 queued; 0 wait; 0 hold; 0 arrive;

  Request Name  Request ID Usr      Pri     State     SIZE

1  rasfile       18.bill    rje      31      Running   1221
```

4. Bill sees that his request is running and retrieves it from the ILP2217 at server **blue**.

Submitting Batch Requests

A batch request is used to execute a file containing command procedures. An NQS batch request consists of a file containing commands and options that control request execution.

When a batch request executes, NQS logs in as the requesting user. Then, it retrieves input from the file containing the commands to be executed. The output and error messages normally directed to a terminal screen are redirected to predefined standard output and standard error files. NQS returns these output files to the original user's current directory (when the job was submitted) after the request completes execution.

Use the **qsub**(1) command to submit a batch request to NQS. Key in the **qsub**(1) command as follows:

> **qsub** [**-a** *date_time*] [**-e** *machine*: [[/] *path/*] *filename*] [**-eo**]
> [**-ke**] [**-ko**] [**-lf** *file_size_limit*] [**-ln** *nice_value_limit*] [**-mb**]
> [**-me**] [**-mu** *username*] [**-nr**] [**-o** *machine*: [[/] *path/*] *filename*]
> [**-p** *priority*] [**-q** *queue*] [**-r** *request_name*] [**-re**] [**-ro**] [**-s** *shell*] [**-x**]
> [**-z**] *script_file*

The script file contains the file to be executed. The **qsub**(1) options are listed in the following table. Note the similarities between the **qsub**(1) and **qpr**(1) options.

Option	**Description**
-a *date_time*	Executes the request after the specified time.
-e *machine*: [[/] *path/*] *filename*	Directs the standard error (**stderr**) output to the specified destination.
-eo	Directs the **stderr** output to the standard output (**stdout**) destination.
-ke	Keeps the stderr output on the executing machine.
-ko	Keeps the stdout output on the executing machine.
-lf *file_size_limit*	Establishes the per-process permanent file size limits.
-ln *nice_value_limit*	Establishes per-process nice value (execution priority).
-mb	Sends mail when the request begins executing.

-me	Sends mail when the request finishes executing.
-mu *username*	Sends mail about the request to the specified user.
-nr	Specifies that the batch request is not restartable.
-o	Directs the stdout output to the stated destination.
-p *priority*	Establishes the intraqueue request priority.
-q *queue*	Submits the request to the specified queue.
-r *request*	Assigns the specified name to the request.
-re	Accesses the stderr output file from the remote machine.
-ro	Accesses the stdout output file from the remote machine.
-s *shell*	Specifies the shell that will interpret the batch request script.
-x	Exports all environment variables with the request.
-z	Submits the request silently.

Delaying Execution Using -a

The -a option specifies a date and time after which NQS will execute a request. Use the **-a** option if you wish to delay request execution. The following examples illustrate acceptable formats for the **-a** option:

qsub -a "July, 4, 1989 9:00-EST" *script_file*

qsub -a "04-Jul-1989 9am, EST" *script_file*

qsub -a "Wednesday, 09:00:00 EST" *script_file*

qsub -a "9am tues." *script_file*

Placing Output and Error Log Files Using -e, -o, -eo, -ke, -ko, -re, and -ro

The output and error log file parameters specify where the output and error log files will be placed after they are created. The following paragraphs detail each parameter:

- The **-e** and **-o** parameters specify where the standard output and standard error log files will be placed when the batch job executes. If **-e** and **-o** are not specified, the output and error files return to your current directory.

  For example, the following command line routes the standard error output log file to the file **../output/logfile** on the machine **red.** (If the path name supplied is a relative path name, NQS places the file in your home directory.)

  ```
  qsub -e red:output/logfiles script_file
  ```

- The **-eo** parameter tells NQS to direct the error output to the standard output log file. This causes the output log and error log files to merge to one file. For example, the following command tells NQS to direct the standard error output from the request to the standard output log file:

  ```
  qsub -eo script_file
  ```

- The **-ke** (keep error) and **-ko** (keep output) parameters tell NQS to retain the standard output and standard error output log files on the executing machine. For example, the following command tells NQS to retain the standard output log file on the destination machine for the queue **syspipe**:

  ```
  qsub -ko -q syspipe script_file
  ```

NOTE

If the queue you specify in the command line is on your node, -ke and -ko parameters have no meaning; the standard output log files will be placed on your node by default.

- The **-re** and **-ro** parameters force the standard output and standard error output log files to be sent to their destination directories as the output is generated. (NQS normally retains the standard output log files internally until the batch job finishes.) For example, the following command tells NQS to send the error log output file to its destination as it is generated:

  qsub -re *script_file*

NOTE **If the destination for the output log files is a remote node, you cannot use the -re and -ro parameters.**

Limiting the File Size Using -lf

The **-lf** option specifies the maximum size of a file that may be created during the request's execution. For example, the following command limits the size of output files from the request file to 1 MB:

qsub -lf 1mb *script_file*

NOTE **NQS enforces the file size limit internally using the ulimit command. Since 1 MB equals 2048 blocks, NQS uses the following command to enforce a 1 MB file size limit when a request spawns:**

ulimit 2048

Because 2048 blocks is less than the default file size limit imposed when you log in to your machine, your error log file would contain the following error. (This error would not interfere with the request's execution.)

```
ksh [31]: ulimit: bad number
```

Setting the Execution Priority Using -ln

The **ln** parameter sets the request's nice value (execution priority). The execution priority is a number from -20 to 20. Requests with larger execution priority values will receive the lowest priority and consume the least amount of CPU resources. If the **-ln** parameter is not used, the system defaults to 0. A priority of 0 is the highest allowed for a request submitted by a user without super-user privileges.

The following command submits a request with an execution priority of **-10**:

```
qsub -ln -10 script_file
```

Notifying the Request Originator Using -mb, -me, and -mu

NQS can send mail to the request originator (or another user) when a request begins or finishes executing. (Mail is automatically sent to the request originator if the request fails.)

NOTE: If you are the super-user when you submit a request, NQS will send your mail to the root account.

- The **-mb** (mail begin) option tells NQS to send mail to the request originator when the request begins executing. Use the **-mb** option as follows:

  ```
  qsub -mb script_file
  ```

- The **-me** (mail end) option tells NQS to send mail to the request originator when the request finishes executing. Use the **-me** command as follows:

  ```
  qsub -me script_file
  ```

- The **-mu** (mail user) option tells NQS to send mail to a user other than the originator. NQS will send mail when the request finishes printing unless you also specify **-mb**. Use the **-mu** option as shown in the examples below:

- The following command sends mail to user **joe** when the request finishes executing:

```
qsub -mu joe script_file
```

- The following command sends mail to user **sue** when the request begins executing:

```
qsub -mb -mu sue script_file
```

Submitting a Nonrestartable Request Using -nr

The **-nr** option allows you to submit a request that will not restart if the system fails or NQS shuts down. If you do not use the **-nr** option, NQS will attempt to restart the request after recovering from a system failure. Use the **-nr** option as follows:

```
qsub -nr script_file
```

Setting Scheduling Priorities Using -p

The **-p** option defines an intraqueue priority value (the relative order of requests within a queue). The priority value is a number from 0 to 63. Requests with higher scheduling priority values will be positioned first in the queue. If the **-p** option is not used, the default priority value for the queue is used.

NOTE The scheduling priority does not determine the request's execution priority; it determines only the relative ordering of requests in a queue.

The following command submits a request with an intraqueue priority of 20:

```
qsub -p 20 script_file
```

Specifying a Queue Using -q

The **-q** parameter specifies the queue that the request will be submitted to. Using the **-q** parameter to specify a pipe queue, you can submit requests to another node. (You may set up a default queue. See the "Manipulating Devices from the Command Line" section in the System Administrator's "NQS Tutorial.")

For example, the following command line submits a request to the local pipe queue, **syspipe**. **Syspipe** then routes the request to its destination queue on a remote node.

qsub -q syspipe *script_file*

Specifying a Request Name Using -r

The **-r** parameter specifies the name to be assigned to a request file. For example, the following command specifies that the request will be named **barb**:

qsub -r barb *script_file*

The name of the request file (**barb** in this example) is the name that will appear in all status displays and compose part of the output and error log file names. (Names beginning with a number will be prefixed with the letter "r." For example, the name 210 would become r210.)

Defining an Execution Shell Using -s

The **-s** option defines the execution shell that will service the request. If the **-s** option is not used, the system default shell is used.

The execution shell options are the following:

- Bourne shell (sh)
- Korn shell (ksh)
- C shell (csh)

You must be at the machine that the request file currently resides on. You must key in the full path name of the shell that will execute the request. For example, the following command executes the request file using the Korn shell:

```
qsub -s /bin/ksh script_file
```

Exporting Environment Variables Using -x

The -x option allows you to export environment variables with your request so long as the variable names do not conflict with the automatically exported variables, HOME, SHELL, PATH, LOGNAME, MAIL, and TZ.

The automatically exported variables are saved as QSUB_HOME, QSUB_SHELL, QSUB_PATH, QSUB_LOGNAME, QSUB_USER, QSUB_MAIL, and QSUB_TZ when the batch request is spawned. If you do not use the **-x** option, no other environment variables will be exported for the batch request from the originating host.

For example, the following command will export all environment variables from the originating user's machine to be used for executing the request file:

```
qsub -x script_file
```

Embedding Commands in the Request File

All **qsub**(1) options can be embedded in the batch request file to be executed with other commands in the file. The following example illustrates a request file named **echo.bat** with embedded commands. This request will be placed in the queue **sysbatch**, will execute after 11:30 PM Central Daylight Time, and will send mail messages to the originating user when the request begins and finishes executing.

```
$ cat echo.bat

# Comment
# @$-q sysbatch
# @$-a "11:30PM CDT"
# @$-mb -me
# @$
echo This is my batch job
date
```

As shown in the previous example, the **echo.bat** file is broken out as follows:

- Optional parameters are specified in comment lines. The first line in the file is a standard comment.
- The @$ sequence tells NQS that one or more optional parameters will follow. The following three lines are optional parameters that NQS will recognize and execute. (The - symbol must immediately follow the @$ symbols.)
- The @$ sequence by itself indicates the end of a series of optional parameters.

The request file shown above could be submitted to a batch queue named **sysbatch** as follows:

qsub -q sysbatch batch

Checking Resource Limits and Shell Strategy

You can display the default per-process file size limit (set with the **-lf** option), the default per-process nice value limit (set with the **-ln** option), and the default shell strategy (set with the **-s** option) on the local machine by keying in the following:

qlimit

Information similar to the following will appear:

```
Per-process permanent file size limit (-lf)
Nice value (-ln)

Shell strategy=FREE
```

Sample Batch Requests

The following is an example of the simplest type of batch request. The request is explained in the following sample session.

```
$ qsub
echo This is my first batch job.  The time is
date
<Ctrl-D>
Request 29.lga submitted to queue: sysbatch.
$ ls
STDIN.e29
STDIN.o29
$ cat STDIN.o29
This is my first batch job.  The time is
Thu Sep 22 14:20:24 CDT 1988
$ cat STDIN.e29
stty: Not a typewriter
ksh [31]: ulimit: bad number
```

When you enter **qsub** without a script file name, the batch command list is taken from standard input (the keyboard). The request in the example above displays an introduction and then the time and date.

In the example above, NQS assigns a request identification number of 29 to the request and places the request in the queue **sysbatch**. (Request identification numbers are sequential numbers assigned to all NQS requests.) After the request is submitted, NQS schedules and executes it.

A few seconds after the request is submitted, the output and error logs appear in the current directory. (If you do not have write permission in the current directory, the output and error logs are placed in your home directory.)

The Output Log File

The output log file contains the results of commands in the request. In the previous example, the request prints an introduction (This is my first batch job. The time is) and then the time and date (Thu Sep 22 14:20:24 CDT 1988).

Since the original request input came from the keyboard (STanDard INput), NQS names the output log file **STDIN.o29** by default. If the input had come from a request file named **time**, the output file would have been named **time.029**.

The Error Log File

The error log file contains the errors resulting from the request. In the example above, stty and ulimit problems occurred during request execution. The stty error resulted when the default setup **/etc/profile** attempted to set up a terminal device, an invalid procedure in a batch environment. The ulimit error resulted when the default profile attempted to set a permanent file size limit higher than the NQS default limit of 2048 blocks (1 MB).

NOTE

The stty and ulimit errors are common and do not affect the results of the request.

Since the original request input came from the keyboard (STanDard INput), NQS names the error log file **STDIN.e29** by default. If the input had come from a request file named **time**, the error log file would have been named **time.e29**.

Manipulating Requests

There are a number of ways that you can manipulate requests once after they have been submitted to a queue. The following sections describe how to manipulate requests.

NOTE In most cases, you must have NQS operator privileges to manipulate requests. Exceptions are noted in the command description.

Deleting a Request

After a request has been submitted, you can delete it using the **qdel(1)** command. To delete a request, at least one of the following conditions must exist:

- You must own the request.
- You must have NQS operator privileges.
- You must be logged in as the super-user.

To delete a request, key in the **qdel(1)** command as follows:

qdel [**-k**] [*signal*] [**-u** *username*] *request_id* [@*host*]. . .

- The **-k** option sends a kill signal to a process that is already running a request in a queue. You do not need to use the **-k** option if the request is waiting in the queue.
- The *signal* option sends a numeric kill signal (such as the CLIX -9) to a process that is already running a request in a queue.
- The **-u** option allows you to specify the user name of the request owner (if you have NQS operator privileges). You must use the **-u** option if you do not own the request you are deleting.
- The *request_id* is the request identification number obtained using the **qstat(1)** command or the **qmgr(1M) show queue** command.

- Use the @*host* option if the job resides on a remote host. (However, you must have access to these nodes.)

For example, to delete a request from the queue named **il2217** on node **bob**, follow these steps. (You must be at node **bob** to delete the request.)

1. Show the queue as follows to obtain the request identification number.

   qstat il2217@bob

```
   il2217@bob;  type=DEVICE;  [ENABLED, ACTIVE]; pri=16
   1 run; 0 queued; 1 wait; 0 hold; 0 arrive;

  Request Name  Request ID Usr      Pri      State      Size

1:   JOB1          154.bob   bob      31       RUNNING    766
2:   JOB2          155.bob   bob      31       WAITING    531
```

2. Delete JOB2 by keying in the following:

   qdel 155

3. Execute the **qstat(1)** command again as follows to confirm that JOB 1 has been deleted.

   qstat il2217

```
    i12217@bob; type=DEVICE; [ENABLED, ACTIVE]; pri=16
    1 run; 0 queued; 0 wait; 0 hold; 0 arrive;

   Request Name  Request ID Usr      Pri      State     Size

1:   JOB1          154.bob   bob      31      RUNNING   766
```

Modifying a Request

You can modify several request parameters after a job has reached its destination queue, so long as the job has not begun executing. (However, a changed priority will not take effect unless the job returns to a WAIT state.) You must have operator privileges to modify a request. Currently, the only request parameter that may be changed is the scheduling priority.

To modify a request, obtain the request ID using the **qstat**(1) or **show queue** command. Then, key in the command as follows:

> Mgr: **modify request** *request_id* [**priority=***priority_value*]

Holding a Request

The **hold** command is used to place a request in an operator hold state. You must have operator privileges to put a request on operator hold.

To place a request on operator hold, obtain the request ID using the **qstat**(1) or **show queue** command. Then, key in the **hold** command as follows:

> Mgr: **hold** *request_id*

Unholding a Request

The **unhold** command is used to remove a request from the operator hold state. The request will return to the queued or waiting state.

To remove a request from the operator hold state, obtain the request ID using the **qstat(1)** or **show queue** command. Then, key in the **unhold** command as follows:

Mgr: **unhold** *request_id*

Releasing a Request

The **release** command releases a request from any held or waiting state. To release a request, obtain the request ID using the **qstat(1)** or **show queue** command. Then, key in the **release** command as follows:

Mgr: **release** *request_id*

Purging Requests from a Queue

The **purge queue** command deletes all requests that are not running from the specified queue. Purged requests cannot be recovered.

Key in the **purge queue** command as follows:

Mgr: **purge queue** *queue_name*

Aborting Requests in a Queue

The **abort queue** command sends a kill signal (SIGTERM) to each process for each request running in the specified queue when the **abort queue** command was received.

Key in the **abort queue** command as follows. (If you do not specify a grace period, the grace period defaults to 20 seconds. Aborted requests cannot be recovered.

Mgr: **abort queue** *queue_name* [*grace_period*]

Troubleshooting

If you have problems operating NQS, you may be able to fix the problem yourself. Please refer to this section or the "Troubleshooting" section in the *Network Queuing System (NQS) User's Guide* to see if you can diagnose and solve the problem. If the problem persists, call Intergraph support.

NQS returns mail saying "Unknown input type."

Cause:	NQS did not recognize the input type specified in the **qpr**(1) command line.
Solution:	Ensure that you have installed the appropriate metafile interpreter and that you have correctly keyed in the input type.

The error, "Print file limits exceeded" or "Print file exceeds maximum quote limits" displays.

Cause:	Your request exceeded the maximum print file size.
Solution:	Use the NQS **qmgr**(1M) utility to increase your maximum print file size. (See the "Setting Defaults" section in the System Administrator's "NQS Tutorial.")

You did not receive mail from your job.

Cause:	You may have been logged in as the root user when you submitted your job.
Solution:	Log in the console window as root to read your mail.
Cause:	You did not specify **-me** or **-mb** when submitting your job.
Solution:	If you want to receive mail when a job begins or finishes executing, you must specify **-me** or **-mb** on your submit command line. Otherwise, you will only receive mail when an error occurs.

When trying to configure a plotter, you receive the error " NQS manager (FATAL): Unable to open the network file descriptor file."

Cause:	You may have loaded NQS for the first time and not rebooted your machine.
Solution:	Reboot your machine.

You cannot execute the qstat(1) command to a remote machine; however, no error displays and no prompt is returned within 45 seconds.

Jobs move between the ROUTING and WAITING states in the queue, but are not serviced.

Cause:	The remote node is down.
Solution:	Attempt to use the **visit(1)** command to log in to the remote node. If you cannot log in, the remote node is down. Try again later.
Cause:	NQS is not loaded on the remote node.
Solution:	Ask your system administrator to load NQS on the remote node or use a different node.
Cause:	One of the NQS daemons has died on the remote node.
Solution:	Ask the administrator of the remote node to stop and restart NQS on the node.
Cause:	There is an address conflict in the **/etc/hosts** file. This means that the node you are attempting to contact has the same Internet address as another node.
Solution:	Update the clearinghouse, run **namex(1M)**, and execute **/etc/dodini**.

You receive the error "NQS (FATAL): Multiple host names defined for *nnn.nnn.nnn.nnn* in /etc/hosts."

Cause:	Another node has an Internet address identical to yours or your node name aliases are improperly scoped.
Solution:	Read the "BSD Network Configuration Tutorial" for information on Internet addresses and ask your system administrator to assign a new Internet address to your node.

You receive the error "NQS (FATAL): No Internet Address," but your node has an Internet address.

You receive the error "Local network database error at transaction peer."

Cause:	The clearinghouse needs to be updated.
Solution:	Update the clearinghouse, run **namex**(1M), and execute **/etc/dodini**.

You receive the error "No local daemon at host."

Cause:	The local daemon, **/usr/lib/nqs/nqsdaemon**, has died.
Solution:	Start NQS by keying in **/etc/init.d/nqs start** at the super-user (#) prompt.

You receive the error "Insufficient privilege at local host."

Cause:	You do not have privileges to perform the operation you attempted.
Solution:	Ask your system administrator to assign you NQS manager or operator privileges.

You submitted a job to the queue and received a message that the job was successfully submitted, but it did not execute.

Cause:	The pipe or device queue may be stopped. (If you are plotting, see your *InterPlot User's Guide.*)
Solution:	Use the **qstat**(1) command to see if the queue is stopped. If it is, restart it with the **start queue** command.
Cause:	The device may be disabled.
Solution:	Use the **qstat**(1) command to see if the device is disabled. If it is, enable it with the **enable device** command.
Cause:	Another device is running on the same mux as the device you have submitted your job to.
Solution:	Check the devices connected to your mux. If a device is running on the mux, your job will run when the job has completed.

Cause:	The queue complex limit has been reached.
Solution:	Check the current queue complex limits with the **qstat -c** command. Ensure that no other queues are running and causing the queue complex run limit to be reached.

8. RCS Tutorial

Chapter 8: RCS Tutorial

Introduction

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, such as programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions.

- RCS stores and retrieves multiple revisions of a program and other text. Thus, you can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original; previous revisions remain accessible.
  - RCS Maintains each module as a tree of revisions.
  - Project libraries can be organized centrally, decentralized, or any other way.
  - RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
- RCS maintains a complete history of changes. Thus, you can find out what happened to a module easily and quickly, without having to compare source listings or having to consult colleagues.
  - RCS performs automatic record keeping.
  - RCS logs all changes automatically.
  - RCS guarantees project continuity.
- RCS manages multiple lines of development.
- RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, changes are merged automatically.
- RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.
- RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other.

- RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
- RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.
- RCS automatically identifies modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules compose a given configuration.
- RCS Provides high-level management visibility. Thus, it is easy to track the status of a software project.
    - RCS provides a complete change history.
    - RCS records who did what when to which revision of which module.
- RCS is fully compatible with existing software development tools. RCS is unobtrusive; its interface to the file system allows all your existing software tools to be used as before.
- RCS's basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned toward advanced software development environments and the experienced software professional.
- RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
- RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

Suppose you have a file **f.c** that you wish RCS to control. Invoke the checkin command as follows:

```
ci f.c
```

This command creates **f.c,v**, stores **f.c** in it as revision 1.1, and deletes **f.c**. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

Files ending in ",v" are called RCS files ("v" stands for "versions"); the others are called working files. To get back the working file **f.c** in the previous example, use the checkout command:

```
co f.c
```

This command extracts the latest revision from **f.c,v** and writes it in **f.c**. You can now edit **f.c** and check it back in by invoking:

```
ci f.c
```

ci(1) increments the revision number properly. If **ci(1)** complains with the following message

```
ci error: no lock set by <your login>
```

your system administrator has decided to create all RCS files with the locking attribute set to "strict." With strict locking, you must lock the revision during the previous checkout. Thus, your last checkout should have been as follows:

```
co -l f.c
```

Locking assures that only you can check in the next update, and avoids nasty problems if several people are working on the same file. Of course, it is too late now to checkout with locking, because you probably modified **f.c** already, and a second checkout would overwrite your changes. Instead, invoke the following:

```
rcs -l f.c
```

This command will lock the latest revision for you unless somebody else locked it. If someone else has the lock you will need to negotiate your changes with them.

If your RCS file is private, (if you are the only person who is going to revise it), strict locking is not needed and you can turn it off. If strict locking is turned off, the owner of the RCS file need not have a lock for checkin; all others still do. Strict locking is turned off and on with the following commands:

```
rcs -U f.c      and      rcs -L f.c
```

You can set the locking to strict or nonstrict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called **RCS** in your working directory and move all your RCS files there. RCS commands will look first in that directory to find needed files. All commands discussed above will still work without any change.

Pairs of RCS and working files can really be specified in three ways: both are given, only the working file is given, or only the RCS file is given. Both files may have arbitrary path prefixes; RCS commands pair them intelligently.

To avoid deleting the working file during checkin (if you want to continue editing), invoke the following:

```
ci -l f.c
```

This command checks in **f.c** as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. An option **-u** for **ci(1)** also performs a checkin followed by a checkout without locking. This is useful if you want to compile the file after the checkin. Both options also update the identification markers in your file (see below).

You can give **ci(1)** the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The following command

```
ci -r2 f.c      or      ci -r2.1 f.c
```

assigns the number **2.1** to the new revision. From then on, **ci(1)** will number the subsequent revisions with 2.2, 2.3, etc. The corresponding **co(1)** commands

```
co -r2 f.c      and      co -r2.1 f.c
```

retrieve the latest revision numbered 2.*x* and the revision 2.1, respectively. **co(1)** without a revision number selects the latest revision on the "trunk" (the highest revision with a number consisting of two fields). Numbers with more than two fields are needed for branches. For example, to start a branch at revision 1.3, invoke the following:

```
ci -r1.3.1 f.c
```

This command starts a branch numbered **1** at revision **1.3** and assigns the number **1.3.1.1** to the new revision. For more information about branches, see **rcsfile**(4) in the *CLIX Programmer's & User's Reference Manual.*

Automatic Identification

RCS can put special strings for identification in your source and object code. To obtain such identification, place the following marker

$Header$

in your text (for instance in a comment). RCS will replace this marker with a string with the following form:

$Header: *filename revisionnumber date time author state*
$

You never need to touch this string, because RCS updates it automatically. To propagate the marker to your object code, simply put it in a literal character string. In C, this is accomplished as follows:

```
static char rcsid[] = "$Header$";
```

The command **ident(1)** extracts such markers from any file, even object code. Thus, **ident(1)** helps you to find out which revisions of which modules were used in a given program.

You may also find it useful to put the following marker

Log

in your text, inside a comment. This marker accumulates the log messages requested during checkin. Thus, you can maintain the complete history of your file directly inside it. Several additional identification markers exist; see co(1) for details.

How to Combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule in your makefile. Do not use a rule with the form ".c,v.c" because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use the following:

```
.SUFFIXES: .c,v

        co -q $*.c
        cc $(CFLAGS) -c $*.c
        rm -f $*.c

prog:   f1.o f2.o .....
        cc f1.o f2.o ..... -o prog
```

This rule has the following effect. If a file **f.c** does not exist and **f.o** is older than **f.c,v**, **make**(1) checks out **f.c**, compiles **f.c** into **f.o**, and then deletes **f.c**. From then on, **make**(1) will use **f.o** until you change **f.c,v**.

If **f.c** exists (presumably because you are working on it), the default rule ".c.o" takes precedence and **f.c** is compiled into **f.o**, but not deleted.

If you keep your RCS file in the directory **./RCS**, this will not work and you need to write explicit checkout rules for every file, as the following:

f1.c: RCS/f1.c,v; co -q f1.c

Unfortunately, these rules do not have the property of removing unneeded .c files.

Additional Information on RCS

If you want to know more about RCS (for example how to work with a tree of revisions and how to use symbolic revision numbers), read the following paper:

Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Looking at the manual page **rcsfile**(4) should also help you understand the revision tree permitted by RCS.

Chapter 9: RPC/XDR Tutorial

Introduction

This document is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives based on sockets. The reader must be familiar with the C programming language and should have a working knowledge of network theory.

Programs which communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada **rendezvous**. The method used by the NFS is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs the service requested, sends back the reply, and the procedure call returns to the client.

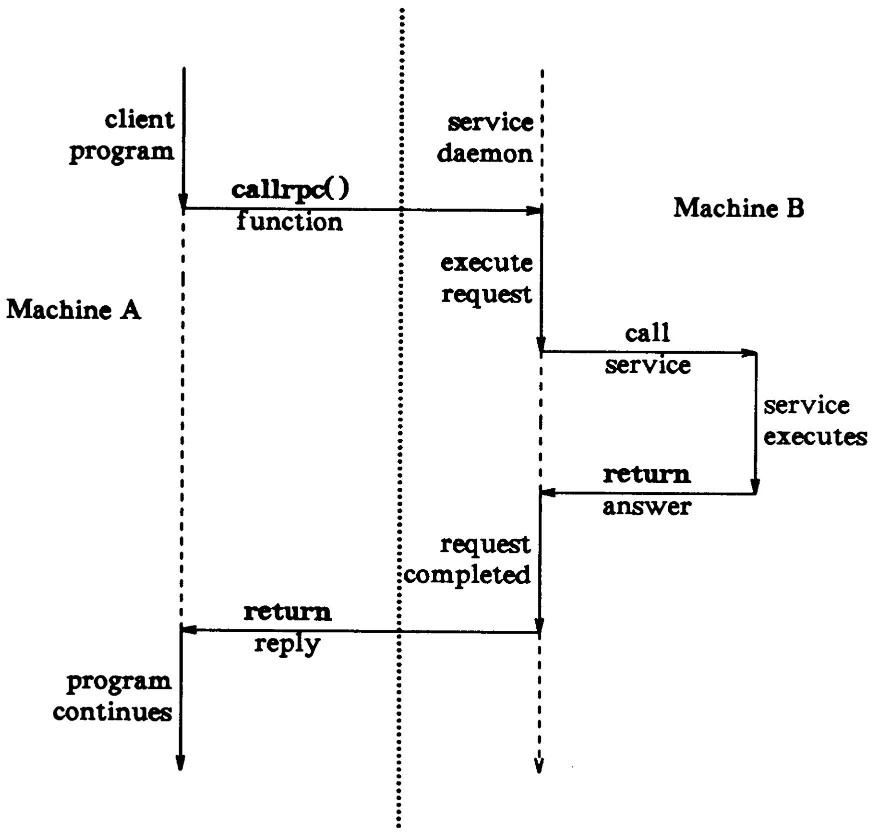
The RPC interface is divided into three layers. The highest layer is totally transparent to the programmer. To illustrate, at this level a program can contain a call to **rnusers**( ), which returns the number of users on a remote machine. The user need not be aware that RPC is being used, since the call is simply made in a program, just as **malloc**(3R) would be called.

At the middle layer, the routine **registerrpc**( ) and **callrpc**( ) are used to make RPC calls: **registerrpc**( ) obtains a unique system-wide number, while **callrpc**( ) executes a remote procedure call. The **rnusers**( ) call is implemented using these two routines. The middle-layer routines are designed for most common applications and shield the user from needing to know about sockets.

The lowest layer is used for more sophisticated applications, which may want to alter the defaults of the routines. At this layer, sockets used for transmitting RPC messages can be explicitly manipulated. This level should be avoided if possible.

Although this document only discusses the interface to C, remote procedure calls can be made from any language. The libraries needed are **libbsd.a** and **librpcsvc.a**. Related #include files are located in **/usr/include/RPC** and **/usr/include/RPCSVC.n**. Even though this document discusses RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same machine.

The following is a diagram of the RPC paradigm.



Network Communication with RPC

Introductory Examples

Highest Layer

Consider a program that needs to know how many users are logged in to a remote machine. This can be done by calling the library routine **rnusers()** as illustrated below:

```
#include <stdio.h>

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned num;

        if (argc < 2) {
                fprintf(stderr, "usage: rnusers hostname\n");
                exit(1);
        }
        if ((num = rnusers(argv[1])) < 0) {
                fprintf(stderr, "error: rnusers\n");
                exit(-1);
        }
        printf("%d users on %s\n", num, argv[1]);
        exit(0);
}
```

Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions **callrpc( )** and **registerrpc( )**. Using this method, another way to get the number of remote users is:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned long nusers;

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if (callrpc(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
            xdr_void, 0, xdr_u_long, &nusers) != 0) {
                fprintf(stderr, "error: callrpc\n");
                exit(1);
        }
        printf("number of users on %s is %d\n", argv[1], nusers);
        exit(0);
}
```

A program number, version number, and procedure number defines each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example) a new program number does not have to be assigned.

When a procedure is to be called to find the number of remote users, the appropriate program, version, and procedure numbers are looked up in a manual in a similar manner to looking up the name of memory allocator when memory is to be allocated.

The simplest routine in the RPC library used to make remote procedure calls is **callrpc( )**. It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and procedure numbers. The following two parameters define the argument of the RPC call, and the final two parameters are for the returned value of the call. If it completes successfully, **callrpc( )** returns zero, but nonzero otherwise. The exact meaning of the return codes is found in **<rpc/clnt.h>**, and is, in fact, an enum clnt_stat cast into an integer.

Since data types may be represented differently on different machines, **callrpc( )** needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For **RUSERSPROC_NUM**, the returned value is an unsigned long, so **callrpc( )** has *xdr_u_long* as its first returned parameter, which says that the result is of type unsigned long, and has *&nusers* as its second returned parameter, which is a pointer to where the long result will be placed. Since **RUSERSPROC_NUM** takes no argument, the argument parameter of **callrpc( )** is *xdr_void*.

After trying several times to deliver a message, if **callrpc( )** gets no answer, it returns with an error code. The delivery mechanism is UDP, which stands for User Datagram Protocol. Methods for adjusting the number of retries or for using a different protocol require the use of the lower layer of the RPC library, discussed later in this document. The remote server procedure corresponding to the above might look like this:

```
char *
nuser(indata)
        char *indata;
{
        static int nusers;
        /*
         * code here to compute the number of users
         * and place result in variable nusers
         */
        return ((char *)&nusers);
}
```

It takes one argument, a pointer to the input of the remote procedure call (ignored in the above example), and returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so both the input argument and the returned value are cast to char *.

Normally, a server registers all of the RPC calls it plans to handle and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server would look like this:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
        registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser,
            xdr_void, xdr_u_long);
        svc_run();                  /* never returns */
        fprintf(stderr, "Error: svc_run returned!\n");
        exit(1);
}
```

The **registerrpc**( ) routine establishes what C procedure corresponds to each RPC procedure number. The first three parameters, *RUSERPROG*, *RUSERSVERS*, and *RUSERSPROC_NUM* are the program, version, and procedure numbers of the remote procedure to be registered; *nuser* is the name of the C procedure implementing it; and *xdr_void* and *xdr_u_long* are the types of the input to and output from the procedure.

Only the UDP transport mechanism can use **registerrpc**( ); thus, it is always safe with calls generated by **callrpc**( ) .

The UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.

Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536,870,912) according to the following chart:

0	-	1FFFFFFF	defined by Sun Microsystems
20000000	-	3FFFFFFF	defined by user
40000000	-	5FFFFFFF	transient
60000000	-	7FFFFFFF	reserved
80000000	-	9FFFFFFF	reserved
A0000000	-	BFFFFFFF	reserved
C0000000	-	DFFFFFFF	reserved
E0000000	-	FFFFFFFF	reserved

Sun Microsystems administers the first group of numbers, and the intent is that they will be be identical across all systems and applications. If a customer develops an application that might be of general interest, that application should be given a number assigned by Sun from the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

Passing Arbitrary Data Types

In the previous example, the RPC call passes a single unsigned long. RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a network standard called eXternal Data Representation (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*. The type field parameters of **callrpc( )** and **registerrpc( )** can be a built-in procedure like **xdr_u_long( )** in the previous example or a user-supplied one. XDR has these built-in type routines:

xdr_int()	**xdr_u_int()**	**xdr_enum()**
xdr_long()	**xdr_u_long()**	**xdr_bool()**
xdr_short()	**xdr_u_short()**	**xdr_string()**

As an example of a user-defined type routine, if it was wished to send the structure

```
struct simple {
        int a;
        short b;
} simple;
```

callrpc( ) should be called as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);
```

where **xdr_simple( )** is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
        XDR *xdrsp;
        struct simple *simplep;
{
        if (!xdr_int(xdrsp, &simplep->a))
                return (0);
        if (!xdr_short(xdrsp, &simplep->b))
                return (0);
        return (1);
}
```

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of XDR is in the *SDR Protocol Specification*, so this section only gives a few examples of XDR implementation. For details on obtaining a complete copy, call Intergraph Support.

In addition to the built-in primitives, there are also the prefabricated building blocks:

xdr_array()	**xdr_bytes()**
xdr_reference()	**xdr_union()**

To send a variable array of integers, they might be packaged as a structure like this

```
struct varintarr {
        int *data;
        int arrlnth;
} arr;
```

and make an RPC call such as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_varintarr, &arr...);
```

with **xdr_varintarr()** defined as:

```
xdr_varintarr(xdrsp, varintarr)
        XDR *xdrsp;
        struct varintarr *arrp;
{
        xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, MAXLEN,
                sizeof(int), xdr_int);
}
```

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, the following could also be used to send out an array of length SIZE:

```
        int intarr[SIZE];

        xdr_intarr(xdrsp, intarr)
                XDR *xdrsp;
                int intarr[];
        {
                int i;

                for (i = 0; i < SIZE; i++) {
                        if (!xdr_int(xdrsp, &intarr[i]))
                                return (0);
                }
                return (1);
        }
```

XDR always converts quantities to four-byte multiples when deserializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine **xdr_bytes( )**, which is like **xdr_array( )** except that it packs characters. It has four parameters which are the same as the first four parameters of **xdr_array( )**. For null-terminated strings, there is also the **xdr_string( )** routine, which is the same as **xdr_bytes( )** without the length parameter. On serializing, it gets the string length from **strlen( )** and on deserializing, it creates a null-terminated string.

Here is a final example that calls the previously written **xdr_simple( )** as well as the built-in functions **xdr_string( )** and **xdr_reference( )**, which chases pointers:

```
struct finalexample {
        char *string;
        struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
        XDR *xdrsp;
        struct finalexample *finalp;
{
        int i;

        if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
                return (0);
        if (!xdr_reference(xdrsp, &finalp->simplep,
            sizeof(struct simple), xdr_simple);
                return (0);
        return (1);
}
```

Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically. This section shows how to change the defaults by using lower layers of the RPC library. It is assumed that the reader is familiar with sockets and the system calls for dealing with them.

More on the Server Side

A number of assumptions are built into **registerrpc()**. One is that the UDP datagram protocol is being used. Another is that the user does not want to do anything unusual while deserializing, since the deserialization process happens automatically before the user's server routine is called. The server for the **nuser()** program shown below is written using a lower layer of the RPC package, which does not make these assumptions.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

int nuser();

main()
{
        SVCXPRT *transp;

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL){
                fprintf(stderr, "couldn't create an RPC server\n");
                exit(1);
        }
        pmap_unset(RUSERSPROG, RUSERSVERS);
        if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
            IPPROTO_UDP)) {
                fprintf(stderr, "couldn't register RUSER service\n");
                exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "should never reach this point\n");
}
```

```
nuser(rqstp, tranp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        unsigned long nusers;

        switch (rqstp->rq_proc) {
        case NULLPROC:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        case RUSERSPROC_NUM:
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
                if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

First, the server gets a transport handle, which is used for sending out RPC messages. **registerrpc()** uses **svcudp_create()** to get a UDP handle. If a reliable protocol is required, **svctcp_create()** should be called instead. If the argument to **svcudp_create()** is *RPC_ANYSOCK*, the RPC library creates a socket on which to send out RPC calls. Otherwise, **svcudp_create()** expects its argument to be a valid socket number. If the user specifies his own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of **svcudp_create()** and **clntudp_create()** (the low-level client routine) must match.

When the user specifies *RPC_ANYSOCK* for a socket or gives an unbound socket, the system determines port numbers in the following way: when a server starts, it advertises to a port mapper daemon on its local machine, which picks a port number for the RPC procedure if the socket specified to **svcudp_create( )** is not already bound. When the **clntudp_create( )** call is made with an unbound socket, the system queries the port mapper on the machine to which the call is being made and gets the appropriate port number. If the port mapper is not running or has no port corresponding to the RPC call, the RPC call fails. Users can make RPC calls to the port mapper themselves. The appropriate procedure numbers are in the include file **<rpc/pmap_prot.h>**.

After creating an SVCXPRT, the next step is to call **pmap_unset( )** so that if the *nusers* server crashed earlier, any previous trace of it is erased before restarting. More precisely, **pmap_unset( )** erases the entry for RUSERS from the port mapper's tables.

Finally, the program number for *nusers* is associated with the procedure **nuser( )** . The final argument to **svc_register( )** is normally the protocol being used which, in this case, is *IPPROTO_UDP*. Notice that unlike **registerrpc( )**, there are no XDR routines involved in the registration process. Also, registration is done on the program rather than procedure level.

The user routine **nuser( )** must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by **nuser( )** which are handled automatically by **registerrpc( )**. The first is that procedure **NULLPROC** (currently zero) returns with no arguments. This can be used as a simple test for detecting if a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, **svcerr_noproc( )** is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via **svc_sendreply( )**. Its first parameter is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned. Not previously illustrated is how a server handles an RPC program that passes data. As an example, a procedure RUSERSPROC_BOOL, which has an argument *nusers* and returns TRUE or FALSE depending on whether there are *nusers* logged on, can be added. It would look like this:

```
case RUSERSPROC_BOOL: {
        int bool;
        unsigned nuserquery;

        if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
                svcerr_decode(transp);
                return;
        }
        /*
         * code to set nusers = number of users
         */
        if (nuserquery == nusers)
                bool = TRUE;
        else
                bool = FALSE;
        if (!svc_sendreply(transp, xdr_bool, &bool)){
                 fprintf(stderr, "couldn't reply to RPC call\n");
                 exit(1);
        }
        return;
}
```

The relevant routine is **svc_getargs()**, which takes as arguments an SVCXPRT handle, the XDR routine, and a pointer to where the input is to be placed.

Memory Allocation with XDR

XDR routines not only do input and output; they also allocate memory. This is why the second parameter of **xdr_array()** is a pointer to an array, rather than the array itself. If it is NULL, **xdr_array()** allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine **xdr_chararr1()**, which deals with a fixed array of bytes with length SIZE:

```
xdr_chararr1(xdrsp, chararr)
        XDR *xdrsp;
        char chararr[];
{
        char *p;
        int len;

        p = chararr;
        len = SIZE;
        return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

It might be called from a server like this,

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

where *chararr* has already allocated space. If XDR was wanted to do the allocation, this routine would have to be rewritten in the following way:

```
xdr_chararr2(xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
        return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The RPC call might then look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * use the result here
 */
svc_freeargs(xdrsp, xdr_chararr2, &arrptr);
```

After using the character array, it can be freed with **svc_freeargs()**. In the routine **xdr_finalexample()** given earlier, if finalp->string was NULL in the call

```
svc_getargs(transp, xdr_finalexample, &finalp);
```

then

```
svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

frees the array allocated to hold finalp->string; otherwise, it frees nothing. The same is true for finalp->simplep.

To summarize, each XDR routine is responsible for serializing, deserializing, and allocating memory. When an XDR routine is called from **callrpc()**, the serializing part is used. When called from **svc_getargs()**, the deserializer is used. When called from **svc_freeargs()**, the memory deallocator is used. When building simple examples like those in this section, a user does not have to worry about the three modes.

The Calling Side

When **callrpc()** is used, there is no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that allows adjustment of these parameters, consider the following code to call the *nusers* service:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
        int argc;
        char **argv;
{
        struct hostent *hp;
        struct timeval pertry_timeout, total_timeout;
        struct sockaddr_in server_addr;
        int addrlen, sock = RPC_ANYSOCK;
        register CLIENT *client;
```

continued

```
        enum clnt_stat clnt_stat;
        unsigned long nusers;

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if ((hp = gethostbyname(argv[1])) == NULL) {
                fprintf(stderr, "cannot get addr for '%s'\n", argv[1]);
                exit(-1);
        }
        pertry_timeout.tv_sec = 3;
        pertry_timeout.tv_usec = 0;
        addrlen = sizeof(struct sockaddr_in);
        bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = 0;
        if ((client = clntudp_create(&server_addr, RUSERSPROG,
            RUSERSVERS, pertry_timeout, &sock)) == NULL) {
                perror("clntudp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 20;
        total_timeout.tv_usec = 0;
        clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void, 0,
            xdr_u_long, &nusers, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(-1);
        }
        clnt_destroy(client);
}
```

The low-level version of **callrpc()** is **clnt_call()**. It takes a CLIENT pointer rather than a host name. The parameters to **clnt_call()** are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

The CLIENT pointer is encoded with the transport mechanism. **callrpc()** uses UDP. Thus, it calls **clntudp_create()** to get a CLIENT pointer. To get TCP (Transmission Control Protocol), **clnttcp_create()** would be used.

The parameters to **clntudp_create()** are the server address, the length of the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. The final argument to **clnt_call()** is the total time to wait for a response. Thus, the number of tries is the **clnt_call()** timeout divided by the **clntudp_create()** timeout.

One thing should be noted when using the **clnt_destroy()** call: it deallocates any space associated with the CLIENT handle, but it does not close the socket associated with it, which was passed as an argument to **clntudp_create()**. The reason is that, if there are multiple client handles using the same socket, it is possible to close one handle without destroying the socket that other handles are using.

To make a stream connection, the call to **clntudp_create()** is replaced with a call to **clnttcp_create()**.

```
clnttcp_create(&server_addr, prognum, versnum, &socket, inputsize, output-
size);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the **clnttcp_create()** call is made, a TCP connection is established. All RPC calls using that CLIENT handle would use this connection. The server side of an RPC call using TCP has **svcudp_create()** replaced by **svctcp_create()**.

Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling **svc_run()**. However, if the other activity involves waiting for a file descriptor, the **svc_run()** call will not work. The code for **svc_run()** is as follows:

```
void
svc_run()
{
        int readfds;

        for (;;) {
                readfds = svc_fds;
                switch (select(32, &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
                        perror("rstat: select");
                        return;
                case 0:
                        break;
                default:
                        svc_getreq(readfds);
                }
        }
}
```

svc_run( ) can be bypassed, and **svc_getreq**( ) called directly. To do this, the file descriptors of the socket(s) associated with the programs which are being waited for must be known. Thus, users can write their own **select**(2B)s which wait on both the RPC socket and their own descriptors.

Broadcast RPC

The portmap and RPC protocols implement broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

1. Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
2. Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.
3. The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
4. All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.

Broadcast RPC Synopsis

```
#include <rpc/pmap_clnt.h>

enum clnt_stat          clnt_stat;

clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults, resultsp, eachresult)
u_long          prog;           /* program number */
u_long          vers;           /* version number */
u_long          proc;           /* procedure number */
xdrproc_t       xargs;          /* xdr routine for args */
caddr_t         argsp;          /* pointer to args */
xdrproc_t       xresults;       /* xdr routine for results */
caddr_t         resultsp;       /* pointer to results */
bool_t (*eachresult)();     /* call with each result obtained */
```

The procedure **eachresult()** is called each time a valid result is obtained. It returns a boolean that indicates whether the client wants more responses.

```
    bool_t              done;

    done =
    eachresult(resultsp, raddr)
    caddr_t             resultsp;
    struct sockaddr_in  *raddr; /* address of machine that sent response*/
```

If *done* is TRUE, broadcasting stops and **clnt_broadcast()** returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT. To interpret clnt_stat errors, feed the error code to **clnt_perrno()** .

Batching

The RPC architecture is designed so that clients send a call message and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgement for every message sent. It is possible for clients to continue computing while waiting for a response, using RPC batch facilities.

RPC messages can be placed in a "pipeline" of calls to a desired server; this is called batching. Batching assumes the following:

1. Each RPC call in the pipeline requires no response from the server, and the server does not send a response message.
2. The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer many call messages and send them to the server in one **write(2)** system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually perform a legitimate call in order to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like the following:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){
        fprintf(stderr, "couldn't create an RPC server\n");
        exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS, windowdispatch,
        IPPROTO_TCP)) {
            fprintf(stderr, "couldn't register WINDOW service\n");
            exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

```
void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char *s = NULL;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "couldn't reply to RPC call\n");
            exit(1);
        }
        return;
    case RENDERSTRING:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "couldn't decode arguments\n");
            svcerr_decode(transp); /* tell caller he screwed up */
            break;
        }
        /*
         * call here to to render the string s
         */
        if (!svc_sendreply(transp, xdr_void, NULL)) {
            fprintf(stderr, "couldn't reply to RPC call\n");
            exit(1);
        }
        break;

    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "couldn't decode arguments\n");
            /*
             * we are silent in the face of protocol errors
             */
            break;
        }
        /*
         * call here to to render the string s,
         * but sends no reply!
         */
        break;
    default:
        svcerr_noproc(transp);
        return;
    }
```

continued

```
    /*
     * now free string allocated while decoding arguments
     */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Of course, the service could have one procedure that takes the string and a boolean to indicate whether the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes:

1. The result's XDR routine must be zero (NULL).
2. The RPC call's timeout must be zero.

Here is an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int addrlen, sock = RPC_ANYSOCK;
    register CLIENT *client;
```

continued

```
    enum clnt_stat clnt_stat;
    char buf[1000];
    char *s = buf;

    /*
     * initial as in example 3.3
     */
    if ((client = clnttcp_create(&server_addr, WINDOWPROG,
        WINDOWVERS, &sock, 0, 0)) == NULL) {
        perror("clnttcp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;
    while (scanf("%s", s) != EOF) {
        clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
            xdr_wrapstring, &s, NULL, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "batched rpc");
            exit(-1);
        }
    }
    /*
     * now flush the pipeline
     */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC,
        xdr_void, NULL, xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }

    clnt_destroy(client);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The above example was completed to render all of the (2000) lines in the file **/etc/termcap**. The rendering service did nothing but throw the lines away. The example was run in the following four configurations:

1. Machine to itself, regular RPC
2. Machine to itself, batched RPC
3. Machine to another, regular RPC
4. Machine to another, batched RPC

The results are as follows:

1. 50 seconds
2. 16 seconds
3. 52 seconds
4. 10 seconds

Running **fscanf**(3S) on **/etc/termcap** only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network file system, require stronger security measures than those which have been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type none.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support. However, this section deals only with UNIX-type authentication, which besides none, is the only supported type.

The Client Side

When a caller creates a new RPC client handle as in the following:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

the appropriate transport instance defaults the associate authentication handle to be the following:

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use UNIX-style authentication by setting clnt->cl_auth after creating the RPC client handle:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with *clnt* to carry with it the following authentication credentials structure:

```
/*
 * UNIX style credentials.
 */
struct authunix_parms {
                        u_long    aup_time;/* credentials creation time */
                        char      *aup_machname;/* host name of client machine */
                        int       aup_uid;/* client's UNIX effective UID */
                        int       aup_gid;/* client's current UNIX group ID */
                        u_int     aup_len;/* the element length of aup_gids array */
                        int       *aup_gids;/* array of groups to which user belongs */
};
```

These fields are set by **authunix_create_default()** by invoking the appropriate system calls.

Since the RPC user created this new style of authentication, he is responsible for destroying it with:

```
auth_destroy(clnt->cl_auth);
```

The Server Side

It is more difficult for service implementors dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC Service request
 */
struct svc_req {
        u_long       rq_prog;       /* service program number */
        u_long       rq_vers;       /* service protocol version number*/
        u_long       rq_proc;       /* the desired procedure number*/
        struct opaque_auth rq_cred; /* raw credentials from the "wire" */
        caddr_t      rq_clntcred;   /* read only, cooked credentials */
};
```

The *rq_cred* is mostly opaque except for one field of interest: the style of authentication credentials:

```
/*
 * Authentication info.  Mostly opaque to the programmer.
 */
struct opaque_auth {
    enum_t    oa_flavor;      /* style of credentials */
    caddr_t   oa_base;        /* address of more auth stuff */
    u_int     oa_length;      /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

1. That the request's *rq_cred* is well formed. Thus, the service implementor may inspect the request's *rq_cred.oa_flavor* to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of *rq_cred* if the style is not one of the styles supported by the RPC package.

2. That the request's *rq_clntcred* field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. As only UNIX style is currently supported, *rq_clntcred* could be cast to a pointer to an *authunix_parms* structure. If *rq_clntcred* is NULL, the service implementor may wish to inspect the other (opaque) fields of *rq_cred* in case the service knows about a new type of authentication that the RPC package does not know about.

The remote user's service example can be extended so that it computes results for all users except UID 16:

```
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;

    /*
     * we don't care about authentication for the null procedure
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "couldn't reply to RPC call\n");
            exit(1);
         }
         return;
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        unix_cred = (struct authunix_parms *) rqstp->rq_clntcred;
        uid = unix_cred->aup_uid;
        break;
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }
    switch (rqstp->rq_proc) {
    case RUSERSPROC_NUM:

        /*
         * make sure the caller is allow to call this procedure.
         */
        if (uid == 16) {
            svcerr_systemerr(transp);
            return;
        }
        /*
```

continued

```
            * code here to compute the number of users
            * and put in variable nusers
            */
           if (!svc_sendreply(transp, xdr_u_long, &nusers) {
               fprintf(stderr, "couldn't reply to RPC call\n");
               exit(1);
           }
           return;
       default:
           svcerr_noproc(transp);
           return;
       }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero). Second, if the authentication parameter's type is not suitable for a particular user's service, **svcerr_weakauth**( ) should be called. Finally, the service protocol should return status for access denied; in the case of the above example, the protocol does not have such a status, so the service primitive **svcerr_systemerr**( ) is called instead.

The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services must implement their own access control policies and reflect these policies as return statuses in their protocols.

Using inetd

Some systems have a utility daemon named **inetd**(1M) that simplifies the administration of RPC servers by controlling them based on an input control file.

If a server is to be started by **inetd**(1M), the only difference from the usual code is that **svcudp_create**( ) should be called as

```
        transp = svcudp_create(0);
```

since **inetd**(1M) passes a socket as file descriptor 0. Also, **svc_register**( ) should be called as

```
        svc_register(PROGNUM, VERSNUM, service, transp, 0);
```

with the final flag as 0, since the program would already be registered by **inetd**(1M). Remember that if he wishes to exit from the server process and return control to **inetd**(1M), the user must explicitly exit, since **svc_run**( ) never returns.

The format of entries in **/etc/servers** for RPC services is as follows:

rpc udp *server program version*

Server is the C code implementing the server and *program* and *version* are the program and version numbers of the service. The key word **udp** can be replaced by **tcp** for TCP-based RPC services.

If the same program handles multiple versions, the version number can be a range, as in this example:

rpc udp /usr/etc/rstatd 100001 1-2

More Examples

Versions

By convention, the first version number of program **FOO** is FOOVERS_ORIG and the most recent version is FOOVERS. Suppose there is a new version of the user program that returns an unsigned short rather than a long. If we name this version RUSERSVERS_SHORT, a server that wants to support both versions would do a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "couldn't reply to RPC call\n");
            exit(1);
        }
        return;
    case RUSERSPROC_NUM:
        /*
         * code here to compute the number of users
```

continued

```
         * and put in variable nusers
         */
        nusers2 = nusers;
        if (rqstp->rq_vers == RUSERSVERS_ORIG)
            if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                fprintf(stderr, "couldn't reply to RPC call\n");
                exit(1);
        }
        else
            if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
                fprintf(stderr, "couldn't reply to RPC call\n");
                exit(1);
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

TCP

Here is an example that is essentially rcp(1). The initiator of the RPC client call takes its standard input and sends it to the server, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
 * The xdr routine:
 *
 * on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
        XDR *xdrs;
        FILE *fp;
{
        unsigned long size;
        char buf[MAXCHUNK], *p;

        if (xdrs->x_op == XDR_FREE)/* nothing to free */
                return 1;
        while (1) {
                if (xdrs->x_op == XDR_ENCODE) {
                        if ((size = fread (buf, sizeof(char), MAXCHUNK, fp))
                            == 0 && ferror(fp)) {
                                fprintf(stderr, "couldn't fread\n");
                                exit(1);
                        }
                }
                p = buf;
                if (!xdr_bytes(xdrs, &p, &size, MAXCHUNK))
                        return 0;
                if (size == 0)
                        return 1;
                if (xdrs->x_op == XDR_DECODE) {
                        if (fwrite(buf, sizeof(char), size, fp) != size) {
                                fprintf(stderr, "couldn't fwrite\n");
                                exit(1);
                        }
                }
        }
}
```

```
/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int err;

    if (argc < 2) {
        fprintf(stderr, "usage: %s server-name\n", argv[0]);
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP, RCPVERS,
        xdr_rcp, stdin, xdr_void, 0) != 0)) {
            clnt_perrno(err);
            fprintf(stderr, " couldn't make RPC call\n");
            exit(1);
    }
}

callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;
{
    struct sockaddr_in server_addr;
    int socket = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "cannot get addr for '%s'\n", host);
        exit(-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clnttcp_create(&server_addr, prognum,
        versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
            perror("rpctcp_create");
```

continued

```
        exit(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in,
                outproc, out, total_timeout);
    clnt_destroy(client)
    return (int)clnt_stat;
}
```

```
/ * The receiving routines */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
    register SVCXPRT *transp;
    if ((transp = svctcp_create(RPC_ANYSOCK, 1024, 1024)) == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}
rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (svc_sendreply(transp, xdr_void, 0) == 0) {
            fprintf(stderr, "err: rcp_service");
            exit(1);
        }
        return;
    case RCPPROC_FP:
        if (!svc_getargs(transp, xdr_rcp, stdout)) {
            svcerr_decode(transp);
            return;
        }
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply\n");
            return;
        }
        exit(0);
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Callback Procedures

Occasionally, it is useful to have a server become a client and make an RPC call back to the process which is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a break point, the roles are reversed and the debugger wants to make an RPC call to the window program, so that it can inform the user that a break point has been reached.

In order to perform an RPC callback, a program number to make the RPC call on is needed. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5FFFFFFF. The routine **gettransient()** returns a valid program number in the transient range and registers it with the portmapper. It only talks to the portmapper running on the same machine as the **gettransient()** routine itself. The call to **pmap_set()** is a test and set operation in that it indivisibly tests whether a program number has already been registered and if it has not, reserves it. On return, the *sockp* argument will contain a socket that can be used as the argument to an **svcudp_create()** or **svctcp_create()** call.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
        int *sockp;
{
        static int prognum = 0x40000000;
        int s, len, socktype;
        struct sockaddr_in addr;
        switch(proto) {
                case IPPROTO_UDP:
                        socktype = SOCK_DGRAM;
                        break;
                case IPPROTO_TCP:
```

continued

```
                        socktype = SOCK_STREAM;
                        break;
                default:
                        fprintf(stderr, "unknown protocol type\n");
                        return 0;
        }
        if (*sockp == RPC_ANYSOCK) {
                if ((s = socket(AF_INET, socktype, 0)) < 0) {
                        perror("socket");
                        return (0);
                }
                *sockp = s;
        }
        else
                s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);
         * may be already bound, so don't check for err */
        bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0) {
                perror("getsockname");
                return (0);
        }
        while (pmap_set(prognum++, vers, proto, addr.sin_port) == 0)
                continue;
        return (prognum-1);
}
```

The following pair of programs illustrates how to use the **gettransient()** routine. The client makes an RPC call to the server, passing it a transient program number. Then, the client waits to receive a callback from the server at that program number. The server registers the program **EXAMPLEPROG** so that it can receive the RPC call informing it of the callback program number. Then, at some random time (on receiving a SIGALRM signal in this example), it sends a callback RPC call, using the program number it received earlier.

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
        char **argv;
{
        int x, ans, s;
        SVCXPRT *xprt;

        gethostname(hostname, sizeof(hostname));
        s = RPC_ANYSOCK;
        x = gettransient(IPPROTO_UDP, 1, &s);
        fprintf(stderr, "client gets prognum %d\n", x);

        if ((xprt = svcudp_create(s)) == NULL) {
            fprintf(stderr, "rpc_server: svcudp_create\n");
            exit(1);
        }
        (void)svc_register(xprt, x, 1, callback, 0);

        ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
            EXAMPLEVERS, xdr_int, &x, xdr_void, 0);
        if (ans != 0) {
                fprintf(stderr, "call: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\n");
}

callback(rqstp, transp)
        register struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        switch (rqstp->rq_proc) {
                case 0:
                        if (svc_sendreply(transp, xdr_void, 0)  == FALSE) {
                                fprintf(stderr, "err: rusersd\n");
                                exit(1);
                            }
                        exit(0);
```

continued

```
                case 1:
                        if (!svc_getargs(transp, xdr_void, 0)) {
                                    svcerr_decode(transp);
                                exit(1);
                        }
                        fprintf(stderr, "client got callback\n");
                        if (svc_sendreply(transp, xdr_void, 0)  == FALSE) {
                                fprintf(stderr, "err: rusersd");
                                exit(1);
                        }
        }
}
```

```
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;                  /*program number for callback routine */

main(argc, argv)
        char **argv;
{
        gethostname(hostname, sizeof(hostname));
        registerrpc(EXAMPLEPROG, EXAMPLEPROC_CALLBACK, EXAMPLEVERS,
            getnewprog, xdr_int, xdr_void);
        fprintf(stderr, "server going into svc_run\n");
        alarm(10);
        signal(SIGALRM, docallback);
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\n");
}

char *
getnewprog(pnump)
        char *pnump;
{
        pnum = *(int *)pnump;
        return NULL;
}

docallback()
{
        int ans;

        ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0, xdr_void, 0);
        if (ans != 0) {
                fprintf(stderr, "server: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
}
```

Appendix 1: Synopsis of RPC Routines

auth_destroy()

```
void
auth_destroy(auth)
        AUTH *auth;
```

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy()**.

authnone_create()

```
AUTH *
authnone_create()
```

Creates and returns an RPC authentication handle that passes no usable authentication information with each remote procedure call.

authunix_create()

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
        char *host;
        int uid, gid, len, *aup_gids;
```

Creates and returns an RPC authentication handle that contains UNIX authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs.

authunix_create_default()

```
AUTH *
authunix_create_default()
```

Calls **authunix_create()** with the appropriate parameters.

callrpc()

```
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
        char *host;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds or the value of **enum clnt_stat** cast to an integer if it fails. The routine **clnt_perrno()** is handy for translating failure statuses into messages.

WARNING **Calling remote procedures with this routine uses UDP/IP as a transport; see clntudp_create() for restrictions.**

clnt_broadcast()

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        resultproc_t eachresult;
```

Like **callrpc()**, except the call message is broadcast to all locally connected broadcast networks. Each time it receives a response, this routine calls **eachresult()**, the format of which is

```
eachresult(out, addr)
        char *out;
        struct sockaddr_in *addr;
```

where *out* is the same as *out* passed to **clnt_broadcast()**, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult()** returns zero,

clnt_broadcast() waits for more replies; otherwise, it returns with the appropriate status.

clnt_call()

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
        CLIENT *clnt; long procnum;
        xdrproc_t inproc, outproc;
        char *in, *out;
        struct timeval tout;
```

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clntudp_create**. The parameter *in* is the address of the procedure's argument(s), and *out* is the address where the result(s) should be placed. *Inproc* is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

clnt_destroy()

```
clnt_destroy(clnt)
        CLIENT *clnt;
```

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy()**.

WARNING **Client destruction routines do not close sockets associated with clnt; this is the responsibility of the user.**

clnt_freeres()

```
clnt_freeres(clnt, outproc, out)
        CLIENT *clnt;
        xdrproc_t outproc;
        char *out;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results and *outproc* is the XDR routine describing the results in simple primitives. This routine returns one if the results were successfully freed and zero otherwise.

clnt_geterr()

```
void
clnt_geterr(clnt, errp)
        CLIENT *clnt;
        struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address *errp*.

clnt_pcreateerror()

```
void
clnt_pcreateerror(s)
        char *s;
```

Prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon.

clnt_perrno()

```
void
clnt_perrno(stat)
        enum clnt_stat;
```

Prints a message to standard error corresponding to the condition indicated by *stat*.

clnt_perror()

```
clnt_perror(clnt, s)
        CLIENT *clnt;
        char *s;
```

Prints a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon.

clntraw_create()

```
CLIENT *
clntraw_create(prognum, versnum)
        u_long prognum, versnum;
```

This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svcraw_create()**. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

clnttcp_create()

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        int *sockp;
        u_int sendsz, recvsz;
```

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address **addr*. If addr->sin_port is zero, it is set to the actual port that the remote program is listening on. (The remote portmap service is consulted for this information.) The parameter **sockp* is a socket; if it is RPC_ANYSOCK, this routine opens a new one and sets **sockp*. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz*; values of zero choose suitable defaults. This routine returns NULL if it fails.

clntudp_create()

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        struct timeval wait;
        int *sockp;
```

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address **addr*. If addr->sin_port is zero, it is set to actual port that the remote program is listening on. (The remote portmap service is consulted for this information.) The parameter **sockp* is a socket; if it is RPC_ANYSOCK, this routine opens a new one and sets **sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out.

WARNING Since UDP-based RPC messages can only hold up to eight Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return large results.

get_myaddress()

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Places the machine's IP address in **addr* without consulting the library routines that deal with **/etc/hosts**. The port number is always set to **htons**(PMAPPORT).

pmap_getmaps()

```
struct pmaplist *
pmap_getmaps(addr)
        struct sockaddr_in *addr;
```

A user interface to the portmap service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **addr*. This routine can return NULL. The command **rpcinfo -p** uses this routine.

pmap_getport()

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
        struct sockaddr_in *addr;
        u_long prognum, versnum, protocol;
```

A user interface to the portmap service, which returns the port number on which a service that supports program number *prognum*, version *versnum* waits, and speaks the transport protocol associated with *protocol*. A return value of zero means that the mapping does not exist or that the RPC system failed to contact the remote portmap service. In the latter case, the global variable *rpc_createerr* contains the RPC status.

pmap_rmtcall()

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum,
        inproc, in, outproc, out, tout, portp)
    struct sockaddr_in *addr;
    u_long prognum, versnum, procnum;
    char *in, *out;
    xdrproc_t inproc, outproc;
    struct timeval tout;
    u_long *portp;
```

A user interface to the portmap service, which instructs portmap on the host at IP address **addr* to make an RPC call on the user's behalf to a procedure on that host. The parameter **portp* will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in **callrpc()** and **clnt_call()**; see also **clnt_broadcast()**.

pmap_set()

```
pmap_set(prognum, versnum, protocol, port)
    u_long prognum, versnum, protocol;
    u_short port;
```

A user interface to the portmap service, which establishes a mapping between the triple [*prognum,versnum,protocol*] and *port* on the machine's portmap service. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns one if it succeeds, and zero otherwise.

pmap_unset()

```
pmap_unset(prognum, versnum)
    u_long prognum, versnum;
```

A user interface to the portmap service, which destroys all mappings between the triple [*prognum,versnum,**] and ports on the machine's portmap service. This routine returns one if it succeeds, and zero otherwise.

registerrpc()

```
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
        u_long prognum, versnum, procnum;
        char *(*procname)();
        xdrproc_t inproc, outproc;
```

Registers procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s). *Progname* should return a pointer to its static result(s); *inproc* is used to decode the parameters while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, and -1 otherwise.

WARNING Remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create()** for restrictions.

rpc_createerr

```
struct rpc_createerr        rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror()** to print the reason.

svc_destroy()

```
svc_destroy(xprt)
        SVCXPRT *xprt;
```

A macro that destroys the RPC service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt*.

Use of *xprt* is undefined after calling this routine.

svc_fds

```
        int         svc_fds;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the **select(2)** system call. This is only of interest if a service implementor does not call **svc_run()**, but rather does his own asynchronous event processing. This variable is read-only yet it may change after calls to **svc_getreq()** or any creation routines.

svc_freeargs()

```
        svc_freeargs(xprt, inproc, in)
                SVCXPRT *xprt;
                xdrproc_t inproc;
                char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs()**. This routine returns one if the results were successfully freed and zero otherwise.

svc_getargs()

```
        svc_getargs(xprt, inproc, in)
                SVCXPRT *xprt;
                xdrproc_t inproc;
                char *in;
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The parameter *in* is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds and zero otherwise.

svc_getcaller()

```
        struct sockaddr_in
        svc_getcaller(xprt)
                SVCXPRT *xprt;
```

The approved way of associating the network address of the caller of a procedure with the RPC service transport handle, *xprt*.

svc_getreq()

```
        svc_getreq(rdfds)
                int rdfds;
```

This routine is only of interest if a service implementor does not call **svc_run()**, but instead implements custom asynchronous event processing. It is called when the **select(2)** system call has determined that an RPC request has arrived on some RPC socket(s); *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

svc_register()

```
        svc_register(xprt, prognum, versnum, dispatch, protocol)
                SVCXPRT *xprt;
                u_long prognum, versnum;
                void (*dispatch)();
                u_long protocol;
```

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *protocol* is nonzero, a mapping of the triple [*prognum*, *versnum*, *protocol*] to xprt->xp_port is also established with the local portmap service. (Generally, *protocol* is zero, IPPROTO_UDP or IPPROTO_TCP.) The procedure **dispatch()** has the following form:

```
                dispatch(request, xprt)
                        struct svc_req *request;
                        SVCXPRT *xprt;
```

The **svc_register** routine returns one if it succeeds and zero otherwise.

svc_run()

```
svc_run()
```

This routine never returns. It waits for RPC requests to arrive and calls the appropriate service procedure (using **svc_getreq**) when one arrives. This procedure is usually waiting for a **select(2)** system call to return.

svc_sendreply()

```
svc_sendreply(xprt, outproc, out)
        SVCXPRT *xprt;
        xdrproc_t outproc;
        char *out;
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the caller's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds and zero otherwise.

svc_unregister()

```
void
svc_unregister(prognum, versnum)
        u_long prognum, versnum;
```

Removes all mapping of the double [*prognum,versnum*] to dispatch routines and of the triple [*prognum,versnum,**] to port number.

svcerr_auth()

```
void
svcerr_auth(xprt, why)
        SVCXPRT *xprt;
        enum auth_stat why;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode()

```
void
svcerr_decode(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that cannot successfully decode its parameters. (See **svc_getargs**( ).)

svcerr_noproc()

```
void
svcerr_noproc(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that does not implement the desired procedure number the caller requested.

svcerr_noprog()

```
void
svcerr_noprog(xprt)
        SVCXPRT *xprt;
```

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_progvers()

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the

RPC package. Service implementors usually do not need this routine.

svcerr_systemerr()

```
void
svcerr_systemerr(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()

```
void
svcerr_weakauth(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls **svcerr_auth**(xprt, AUTH_TOOWEAK).

svcraw_create()

```
SVCXPRT *
svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw_create**( ). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times) without any kernel interference. This routine returns NULL if it fails.

svctcp_create()

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
        int sock;
        u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK. In this case, a new socket is created. If the socket is not bound to a local TCP port, this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket number and xprt->xp_port is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of the send and receive buffers; values of zero choose suitable defaults.

svcudp_create()

```
SVCXPRT *
svcudp_create(sock)
        int sock;
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK. In this case, a new socket is created. If the socket is not bound to a local UDP port, this routine binds it to an arbitrary port. Upon completion, xprt->xp_sock is the transport's socket number, and xprt->xp_port is the transport's port number. This routine returns NULL if it fails.

WARNING

Since UDP-based RPC messages can only hold up to eight Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return large results.

xdr_accepted_reply()

```
xdr_accepted_reply(xdrs, ar)
        XDR *xdrs;
        struct accepted_reply *ar;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The parameter *elsize* is the **sizeof()** each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_authunix_parms()

```
xdr_authunix_parms(xdrs, aupp)
        XDR *xdrs;
        struct authunix_parms *aupp;
```

Used for describing UNIX credentials externally. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds and zero otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns one if it succeeds and zero otherwise.

xdr_callhdr()

```
void
xdr_callhdr(xdrs, chdr)
        XDR *xdrs;
        struct rpc_msg *chdr;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_callmsg()

```
xdr_callmsg(xdrs, cmsg)
        XDR *xdrs;
        struct rpc_msg *cmsg;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C double precision numbers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C enums (actually integers) and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C floats and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_inline()

```
        long *
        xdr_inline(xdrs, len)
                XDR *xdrs;
                int len;
```

A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note that pointer is cast to long *.

WARNING

xdr_inline() may return 0 (NULL) if it cannot allocate a contiguous piece of a buffer. Therefore, the behavior may vary among stream instances; it exists for efficiency.

xdr_int()

```
        xdr_int(xdrs, ip)
                XDR *xdrs;
                int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_long()

```
        xdr_long(xdrs, lp)
                XDR *xdrs;
                long *lp;
```

A filter primitive that translates between C long integers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed-size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds and zero otherwise.

xdr_opaque_auth()

```
xdr_opaque_auth(xdrs, ap)
        XDR *xdrs;
        struct opaque_auth *ap;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_pmap()

```
xdr_pmap(xdrs, regs)
        XDR *xdrs;
        struct pmap *regs;
```

Used for describing parameters to various portmap procedures externally. This routine is useful for users who wish to generate these parameters without using the portmap interface.

xdr_pmaplist()

```
xdr_pmaplist(xdrs, rp)
        XDR *xdrs;
        struct pmaplist **rp;
```

Used for describing a list of port mappings externally. This routine is useful for users who wish to generate these parameters without using the portmap interface.

xdr_reference()

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        u_int size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer; *size* is the **sizeof()** the structure that **pp* points to; and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds and zero otherwise.

xdr_rejected_reply()

```
xdr_rejected_reply(xdrs, rr)
        XDR *xdrs;
        struct rejected_reply *rr;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_replymsg()

```
xdr_replymsg(xdrs, rmsg)
        XDR *xdrs;
        struct rpc_msg *rmsg;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C short integers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note that *sp* is the address of the string's pointer. This routine returns one if it succeeds and zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C unsigned integers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C unsigned long integers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C unsigned short integers and their external representations. This routine returns one if it succeeds and zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C union and its corresponding external representation. The parameter *dscmp* is the address of the union's discriminant, while *unp* is the address of the union. This routine returns one if it succeeds and zero otherwise.

xdr_void()

```
xdr_void()
```

This routine always returns one.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls **xdr_string**(xdrs,sp,MAXUNSIGNED) where MAX-UNSIGNED is the maximum value of an unsigned integer. This is useful because the RPC package passes only two parameters to XDR routines, whereas **xdr_string**( ), one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds and zero otherwise.

xprt_register()

```
void
xprt_register(xprt)
        SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

xprt_unregister()

```
void
xprt_unregister(xprt)
        SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should deregister with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.