NAME

intro – Introduction to TCP/IP networking facilities and files

SYNOPSIS

```
#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This section describes the TCP/IP networking facilities, protocols, and data files available in UNICOS. Unless otherwise noted, all include (header) files mentioned in this section are in the /usr/include directory.

Protocols

All network protocols are associated with a protocol family. A *protocol family* provides the services that allow the protocol implementation to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, although the current protocol implementations do not. A protocol family usually is composed of several protocols. The current system fully supports the DARPA Internet protocol family. Raw socket interfaces are provided to the Internet Protocol (IP) and Internet Control Message Protocol (ICMP) layer of the DARPA Internet. For a description of the user protocols of the Internet protocol family, see inet(4P); the protocols are further detailed in tcp(4P) and udp(4P). For a description of Internet Protocol (IP) and Internet Control Message Protocol (ICMP), see the icmp(4P) and ip(4P) man pages.

A network interface is similar to a device interface. *Network interfaces* compose the lowest layer of the networking subsystem, mapping the network system to the device drivers. Associated with a protocol family is an *address format*. An interface may support more than one protocol family or address format. Protocols generally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family and network architecture. TCP/IP uses the following address format:

```
#define  AF_INET  2     /* internetwork: UDP, TCP, etc. */
```

The network facilities provide limited packet routing. A *routing table* is a table in the UNICOS kernel composed of a set of data structures; it is used to select the appropriate network interface when transmitting packets. This table contains one entry for each route to a specific network or host. A user process, called the *routing daemon* (gated(8)), maintains this database with the aid of a routing socket (see route(4P)). Only the super user may perform routing table manipulations.

A routing table entry has the following format, as defined in the net/route.h include file:

```
struct rtentry {
        struct  radix_node rt_nodes[2]; /* tree glue, and other values */
#define      rt_key(r)       ((struct sockaddr *)((r)->rt_nodes->rn_key))
#define      rt_mask(r)      ((struct sockaddr *)((r)->rt_nodes->rn_mask))
        struct  sockaddr *rt_gateway;   /* value */
#ifndef _CRAY
        short   rt_flags;               /* up/down?, host/net */
#else
        int     rt_flags;               /* up/down?, host/net */
#endif
        short   rt_refcnt;              /* # held references */
        u_long  rt_use;                 /* raw # packets forwarded */
        struct  ifnet *rt_ifp;          /* the answer: interface to use */
        struct  ifaddr *rt_ifa;         /* the answer: interface to use */
        struct  sockaddr *rt_genmask;   /* for generation of cloned routes */
        caddr_t rt_llinfo;              /* pointer to link level info cache */
        struct  rt_metrics rt_rmx;      /* metrics used by rx'ing protocols */
        short   rt_idle;                /* easy to tell llayer still live */
#ifdef _CRAY
#define NRTGID 32                       /* maximum size of the gid list */
        long    rt_gid[NRTGID];         /* list of gids for restricting */
        u_long  rt_admmtu;              /* Administrator mtu for the route */
        int     rt_time;                /* Time when the route was added */
        long    rt_pathmtu;             /* Est. minimum MTU over path */
        int     rt_pmtuchanged;         /* Timer for last change to pathmtu */
                                        /* 0 means it has never been changed */
      /*
        * Yes, these are IP specific.  When it becomes necessary to
        * break up rentries according to AF, we'll do something.
        */
        char    rt_iptos;               /* 8-bit IP TOS field */

        /* Handle group id list as a variable length array */
        int      rt_gidcnt;             /* number of gid's in gidlist */
        long    *rt_gidlist;            /* gid list */

#endif /* _CRAY */
};
```

The rt_flags variable is defined as follows:

```
#define RTF_UP          0x1             /* route usable                     */
#define RTF_GATEWAY     0x2             /* destination is a gateway         */
#define RTF_HOST        0x4             /* host entry (net otherwise)       */
#define RTF_REJECT      0x8             /* host or net unreachable          */
#define RTF_DYNAMIC     0x10            /* created dynamically (by redirect) */
#define RTF_MODIFIED    0x20            /* modified dynamically (by redirect)*/
#define RTF_DONE        0x40            /* message confirmed                */
#define RTF_MASK        0x80            /* subnet mask present              */
#define RTF_CLONING     0x100           /* generate new routes on use       */
#define RTF_LLINFO      0x400           /* generated by ARP or ESIS         */
#define RTF_STATIC      0x800           /* manually added                   */
#define RTF_NOFORWARD   0x1000          /* do not forward through           */
#define RTF_EXCLGID     0x2000          /* gid list is exclusive            */
#define RTF_PROTO2      0x4000          /* protocol-specific routing flag   */
#define RTF_PROTO1      0x8000          /* protocol-specific routing flag   */
#define RTF_TOSMATCH    0x10000         /* high-level match required for TOS */
#define RTF_NOMTUDISC   0x40000         /* don't do path MTU discovery      */
```

Three types of entries are in a routing table: the route for a host, the route for all hosts on a network, and the route for any destination not matched by entries of the first two types (a wildcard route). When you boot the system, each network interface that has been configured automatically installs a routing table entry when it wants to have packets sent through it. Typically, the interface specifies the route through a direct connection to the destination host or network. If the route is direct (that is, not through a gateway), the transport layer of a protocol family usually requests that the packet be sent to the host specified in the packet. Otherwise, the request may be to address the packet to a host different from the eventual recipient (that is, the packet is forwarded).

Routing table entries installed by a user process may not specify the hash field (rt_hash), reference count field (rt_refcnt), use field (rt_use), interface field (rt_ifp), time when the route was added (rt_time), discovered path MTU field (rt_pathmtu), or timer for the last change to the path MTU (rt_pmtuchanged); the routing routines fill these in.

If a route is in use when it is deleted (that is, rt_refcnt has a nonzero value), the resources associated with it are not reclaimed until further references to it are released.

The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a nonexistent entry, or ENOBUFS if insufficient resources were available to install a new route.

User processes may read the routing tables through a routing socket or the /dev/mem special file (see mem(4)).

The rt_use field contains the number of packets sent along the route.

The system administrator may use the rt_mtu field to specify a maximum transmission unit size for connections established over the route.

A wildcard routing entry is specified with a destination address value of INADDR_ANY. (The symbol INADDR_ANY is defined in the netinet/in.h include file as 0.) Wildcard routes are used only when the system does not find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

Addressing

An address format is associated with each protocol family. All network addresses adhere to a general structure, called a *sockaddr*. However, each protocol imposes finer and more specific structure, generally renaming the variant.

```
struct sockaddr {
      u_char  sa_len:32;
      u_char  sa_family:32;
      char    sa_data[32];
};
```

The sa_len field contains the total length of the structure, which may exceed 16 bytes. The following address values for sa_family are known to the system (and additional formats are defined for possible future implementation):

```
#define AF_UNIX         1       /* local to host (pipes, portals) */
#define AF_INET         2       /* internetwork: UDP, TCP, etc. */
```

Interfaces

Each network interface in a system corresponds to a path through which messages may be sent and received. The TCP/IP network interfaces supported on UNICOS are the loopback interface (lo(4)), the HYPERchannel adapter interface (hy(4)), the VME network interface (vme(4)), the HSX channel interface (hsx(4)), and the HIPPI interface (hippi(4)). A network interface usually has a hardware device associated with it, although certain interfaces (such as lo) do not.

ioctl(2) Requests

You also can use the following ioctl(2) requests to manipulate network interfaces. Unless specified, the request takes an ifreq structure as its parameter. This structure is defined in the net/if.h include file, as follows:

```
struct ifreq {
#define      IFNAMSIZ     16
       char   ifr_name[IFNAMSIZ];       /* if name, for example, "en0" */
       union {
              struct sockaddr ifru_addr;
              struct sockaddr ifru_dstaddr;
              struct sockaddr ifru_broadaddr;
              short  ifru_flags;
              int    ifru_metric;
              caddr_t      ifru_data;
              struct {
                     mac_label_t ifru_minlabel;
                     mac_label_t ifru_maxlabel;
              } ifru_seclabel;
              long ifru_auth;
       } ifr_ifru;
#define      ifr_addr        ifr_ifru.ifru_addr     /* address */
#define      ifr_dstaddr     ifr_ifru.ifru_dstaddr  /* other end of p-to-p link */
#define      ifr_broadaddr   ifr_ifru.ifru_broadaddr/* broadcast address */
#define      ifr_flags       ifr_ifru.ifru_flags    /* flags */
#define      ifr_metric      ifr_ifru.ifru_metric   /* metric */
#define      ifr_data        ifr_ifru.ifru_data     /* for use by interface */
#ifdef _CRAY
#define ifr_mtu              ifr_ifru.ifru_metric   /* if mtu */
#define ifr_rbufs            ifr_ifru.ifru_metric   /* if read buffers */
#define ifr_wbufs            ifr_ifru.ifru_metric   /* if write buffers */

#define      ifr_minlabel    ifr_ifru.ifru_seclabel.ifru_minlabel /* minimum sec level */
#define      ifr_maxlabel    ifr_ifru.ifru_seclabel.ifru_maxlabel  /* maximum sec level */
#define      ifr_auth        ifr_ifru.ifru_auth            /* valid authorities */
#endif /* _CRAY */
};
```

A list of the available `ioctl`(2) requests follows:

`SIOCGIFADDR`	Gets the interface address.
`SIOCGIFBRDADDR`	Gets the broadcast address for protocol family and interface.
`SIOCGIFCONF`	Gets the interface configuration list. This request takes an `ifconf` structure as a value-result parameter. Initially, you should set the `ifc_len` field of this structure to the size of the buffer to which `ifc_buf` points. On return, it contains the length, in bytes, of the configuration list.
`SIOCGIFDSTADDR`	Gets the point-to-point address for the interface.
`SIOCGIFFLAGS`	Gets the interface flags.
`SIOCGIFLABEL`	Gets the interface security label.

SIOCGIFMETRIC	Gets interface metric.
SIOCGIFMTU	Gets interface maximum transmission unit size.
SIOCGIFNETMASK	Gets interface subnet mask.
SIOCGIFRBUFS	Gets count of read buffers posted to low-level driver.
SIOCGIFWBUFS	Gets maximum count of write buffers that may be posted to low-level driver.
SIOCSIFADDR	Sets the interface address. Following the address assignment, the initialization routine for the interface is called.
SIOCSIFBRDADDR	Sets the broadcast address for protocol family and interface.
SIOCSIFDSTADDR	Sets the point-to-point address for the interface.
SIOCSIFFLAGS	Sets the interface flags field. If the interface is marked as being down, any processes currently routing packets through the interface are notified.
SIOCSIFLABEL	Sets the interface security label.
SIOCSIFMETRIC	Sets interface routing metric. Only user-level routers use the metric.
SIOCSIFMTU	Sets interface maximum transmission unit size.
SIOCSIFRBUFS	Sets count of read buffers to post to low-level driver.
SIOCSIFWBUFS	Sets maximum count of write buffers that may be posted to low-level driver.

The `ifconf` structure is defined in `net/if.h`, as follows:

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration for machine
 * (useful for programs that must know all networks accessible).
 */

struct  ifconf {
        int     ifc_len;                /* size of associated buffer    */
        union  {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf       /* buffer address               */
#define ifc_req     ifc_ifcu.ifcu_req     /* array of structures returned */
} ;
```

Network Data Files

TCP/IP commands (described in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011, and the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022) and TCP/IP library functions (described in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080) use network data files.

A list of the network data files follows.

Data file	Man page entry	Description
`ftpusers`	`ftpusers`(5)	List of unacceptable `ftp`(1B) users
`hosts`, `hosts.bin`	`hosts`(5)	Contains network host name database
`hosts.equiv`	`hosts.equiv`(5)	Public information for validating remote autologin
`.netrc`	`netrc`(5)	TCP/IP autologin information for outbound `ftp` requests
`networks`	`networks`(5)	Network name database
`protocols`	`protocols`(5)	Protocol name database
`.rhosts`	`rhosts`(5)	List of trusted remote hosts and account names
`services`	`services`(5)	Network service name database

FILES

`/usr/include/net/if.h`	Include file for `ifreq` structure
`/usr/include/net/route.h`	Kernel packet forwarding database
`/usr/include/sys/socket.h`	Include file that defines address families

SEE ALSO

`hippi`(4), `hsx`(4), `hy`(4), `icmp`(4P), `inet`(4P), `ip`(4P), `mem`(4), `route`(4P), `tcp`(4P), `udp`(4P), `vme`(4), `ftpusers`(5), `hosts`(5), `hosts.equiv`(5), `netrc`(5), `networks`(5), `protocols`(5), `rhosts`(5), `services`(5)

`ftp`(1B), `remsh`(1B), `rlogin`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`ioctl`(2), `socket`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`getprot`(3C), `getserv`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`gated`(8), `route`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

TCP/IP Network User's Guide, Cray Research publication SG–2009

"Internet Transport Protocols," XSIS 028112, Xerox System Integration Standard

NAME

arp – Address Resolution Protocol

SYNOPSIS

```
pseudo-device  ether
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The arp protocol dynamically maps between DARPA Internet and 10-Mbyte/s Ethernet addresses. The 10-Mbyte/s Ethernet and FDDI interface drivers use arp. It is not specific to the Internet protocols, to FDDI, or to 10-Mbyte/s Ethernet, but this implementation currently supports only Ethernet and FDDI.

The arp protocol caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, arp queues the message, which requires the mapping, and broadcasts a message on the associated network that requested the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. arp queues at most one packet while waiting for a response to a mapping request; only the most recently transmitted packet is kept. If the target host does not respond after several requests, the host is considered to be down for a short period (normally about 20 seconds), allowing an error to be returned to transmission attempts during this interval. The error is EHOSTDOWN for a non-responding destination host, and EHOSTUNREACH for a non-responding router.

The arp cache is stored in the system routing table as dynamically-created host routes. The route to a directly-attached broadcast network is installed as a "cloning" route (one with the RTF_CLONING flag set), causing routes to individual hosts on that network to be created on demand. These routes time out periodically (normally 20 mintues after validation); entries are not validated when not in use. An entry for a host which is not responding is a "reject" route (one with the RTF_REJECT flag set).

arp entries may be added, deleted, or changed with the arp(8) utility. Manually-added entries may be temporary or permanent, and may be "published," in which case the system will respond to ARP requests for that host as if it were the target of the request. In the past, ARP was used to negotiate the use of a trailer encapsulation. This is no longer supported.

The arp protocol watches passively for hosts impersonating the local host (that is, a host that responds to an ARP mapping request for the local host's address).

MESSAGES

The following message indicates that arp has discovered another host on the local network that responds to mapping requests for its own Internet address:

```
duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x
```

BUGS

ARP packets on the Ethernet use only 42 bytes of data; however, the smallest legal Ethernet packet is 60 bytes (not including the cyclic redundancy code (CRC)). Some systems may not enforce the minimum packet size.

SEE ALSO

`route`(4P) in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

`route`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

`inet`(4P) for a description of Internet protocol family

`arp`(8) to display address resolution display and control

`ifconfig`(8) to configure network interface parameters in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

An Ethernet Address Resolution Protocol, RFC 826, Dave Plummer, Network Information Center, SRI

NAME

icmp – Internet Control Message Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Internet Control Message Protocol (ICMP) is a mechanism that hosts and gateways in an IP internetwork use to exchange error messages and other maintenance information. The ICMP is defined in *DARPA Internet Request for Comments*, RFC 792. UNICOS provides a limited interface to ICMP for user programs; sending incorrect ICMP information can cause problems throughout the internetwork.

Transmission

ICMP is a datagram protocol; therefore, a raw ICMP socket has no connections (listen(2) and accept(2) return errors). The sendto(2) system call is the normal method for transmission through an ICMP socket. The connect(2) system call, though not supported by the underlying protocol, permanently associates the socket with a destination for future transmissions, thus enabling use of the send(2) and write(2) system calls on the socket.

The entire contents of a transmission (the buffer in a sendto(2) system call, send(2) system call, or write(2) system call) is packaged into the data portion of one datagram for transmission. To be accepted at the destination, the buffer to be transmitted must contain a valid ICMP datagram, beginning with an ICMP header with a valid checksum. UNICOS does not enforce this locally; datagrams that are not valid appear to be sent with no problems, but they confuse or are ignored by their recipients. An ICMP datagram structure is defined in the netinet/ip_icmp.h include file.

Reception

The recvfrom(2) system call is the normal method for receiving data through an ICMP socket. The connect(2) system call, though not supported by the underlying protocol, permanently associates the socket with a source of future transmissions, thus enabling use of the recv(2) and read(2) system calls on the socket.

Received ICMP datagrams are presented to the user from the socket with their ICMP header included. The ICMP datagram structure is defined in netinet/icmp.h as the icmp structure.

Only ICMP datagrams that have correct data checksums are passed to user programs. Datagrams that have incorrect checksums (which may have been corrupted in transit) are discarded without notice. The following four kinds of ICMP datagrams are not available to user programs:

- Echo request (type 8)
- Time-stamp request (type 13)
- Information request (type 15)
- Address mask request (type 17)

These ICMP datagrams are handled in the kernel and then discarded.

If you do not provide enough buffer space for the entire available datagram in a call to `recvfrom`(2), `read`(2), or `recv`(2), excess bytes at the end of the datagram will be silently discarded.

NOTES

Only the super user may create an ICMP socket.

ICMP provides no mechanism for out-of-band data.

You cannot specify IP options for an outbound ICMP datagram.

MESSAGES

A socket operation may fail with one of the following errors returned:

`EADDRNOTAVAIL`
: Returned if the process tried to create a socket with a network address for which no network interface exists.

`EISCONN`
: Returned if the socket already has a connection when a connection is tried, or if the process is trying to send a datagram with the destination address specified when the socket is already connected.

`ENOBUFS`
: Returned if the system ran out of memory for an internal data structure.

`ENOTCONN`
: Returned if the process is trying to send a datagram, but no destination address has been specified and the socket is not already connected.

FILES

`/usr/include/netinet/in.h`	Include file for Internet addresses
`/usr/include/netinet/ip_icmp.h`	Defines ICMP datagram structure
`/usr/include/sys/socket.h`	Include file that defines address families

SEE ALSO

`inet`(4P), `intro`(4P), `ip`(4P)

`accept`(2), `connect`(2), `listen`(2), `read`(2), `recv`(2), `send`(2), `write`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`ping`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

DARPA Internet Request for Comments, RFC 792

NAME

igmp – Internet Group Management Protocol

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Internet Group Management Protocol (IGMP) is a mechanism that hosts and routers on the physical network use to identify which hosts currently belong to which multicast groups. Multicast routers use this information to determine which multicast diagrams to forward on to potential interfaces. The IGMP is defined in DARPA Internet Request for Comments, RFC 1112.

IGMP is considered part of the Internet Protocol (IP) layer, and messages are transmitted in IP datagrams.

SEE ALSO

inet(4P), intro(4P), ip(4P)

NAME

inet – Description of Internet protocol family

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Internet protocol family is a collection of protocols, piled on top of the Internet Protocol (IP) layer, that use the Internet address format. The Internet protocol family is composed of the IP itself, the Internet Control Message Protocol (ICMP), the Transmission Control Protocol (TCP), and the user datagram protocol (UDP).

The Internet protocol family provides protocol support for the socket types SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW. TCP supports the SOCK_STREAM abstraction, UDP supports the SOCK_DGRAM abstraction, and the SOCK_RAW socket type provides a raw interface to IP and ICMP.

Internet addresses are 4-byte quantities stored in network standard format. The netinet/in.h include file defines an Internet address, as follows:

```
struct  {
      u_long       st_addr:32;
} s_da;
#define s_addr    s_da.st_addr
```

Sockets bound to the Internet protocol family use the following addressing structure:

```
struct  sockaddr_in {
      u_long sin_len:32;
      u_long sin_family:32;
      _SHORTPAD
      u_short sin_port;
      struct      in_addr  sin_addr;
      char  sin_zero[16];
} ;
```

You may create sockets by using address INADDR_ANY to cause wildcard matching on incoming messages.

FILES

`/usr/include/netinet/in.h`	Include file for Internet addresses
`/usr/include/sys/types.h`	Include file for socket types

SEE ALSO

`icmp`(4P), `ip`(4P), `tcp`(4P), `udp`(4P)

NAME

ip – Description of Internet Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
s = socket(AF_INET, SOCK_RAW, protocol);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Internet Protocol (IP) is the network layer protocol that the Internet protocol family uses. IP provides the functions necessary to deliver an Internet datagram from a source to a destination over a TCP/IP network. IP is defined in MIL-STD 1777 and in RFC 791. You probably will not have to use IP sockets unless you are developing new upper-layer protocols. The Transmission Control Protocol (TCP) and user datagram protocol (UDP) are for more general use. (For more information, see tcp(4P) and udp(4P), respectively.)

Transmission

IP is a datagram protocol, so a raw IP socket has no connections; that is, listen(2) and accept(2) return errors. The sendto(2) system call is the normal method for transmission through an IP socket. The connect(2) system call, though not supported by the underlying protocol, permanently associates the socket with a destination for future transmissions, thus enabling use of the send(2) and the write(2) system calls on the socket.

The entire contents of a transmission (the buffer in a sendto(2) system call, send(2) system call, or write(2) system call) is packaged into the data portion of one datagram for transmission. The kernel provides the IP header. The protocol number in the IP header is the protocol number specified in the socket(2) system call used to create the socket.

Reception

The recvfrom(2) system call is the normal method for receiving data through an IP socket. The connect(2) system call, though not supported by the underlying protocol, permanently associates the socket with a destination for future transmissions, thus enabling use of the recv(2) and read(2) system calls on the socket.

Received IP datagrams are presented to the user from the socket with the IP header included. The IP header structure is defined in the netinet/ip.h include file as the ip structure. The datagram contents immediately follow the header in the receiving buffer. Because IP neither generates nor tests checksums on the data, the data may be garbled in transmission.

Only incoming datagrams that carry the protocol number specified in the socket(2) system call are available from an IP socket.

If you do not provide enough buffer space for the entire available datagram in a call to recvfrom(2), read(2), or recv(2), excess bytes at the end of the datagram will be discarded without notice. To determine whether bytes are missing, check the number of bytes returned from recvfrom(2) against the ip_len field in the header of the received datagram.

Options

You may set options at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). You also may access the IP protocol through a raw socket when developing higher-level protocols or developing special-purpose applications. The following are IP-level setsockopt(2) and getsockopt(2) system call options:

IP_OPTIONS
: Provides IP options to be transmitted in the IP header of each outgoing packet or examines the header options of incoming packets. You may use IP options with any socket type in the Internet family. The IP protocol specification (RFC 791) specifies the format of IP options to be sent, except the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address is extracted from the option list and the size adjusted accordingly before use. To disable previously specified options, use a zero-length buffer. An example of this option follows:

```
setsockopt (s, IPPROTO_IP, IP_OPTIONS, NULL, 0);
```

IP_RECVDSTADDR
: Causes the recv(2) system call to return the destination IP address for a UDP datagram. This option is enabled only on a SOCK_DGRAM socket.

IP_TOS, IP_TTL
: Sets the type-of-service (TOS) and time-to-live (TTL) fields in the IP header for SOCK_STREAM and SOCK_DGRAM sockets. The following example includes both these parameters:

```
int tos = IPTOS_LOWDELAY;  /* see netinet/in.h */
setsockopt (s, IPPROTO_IP, IP_TOS, &tos, sizeof(tos));

int ttl = 60;              /* max = 255 */
setsockopt (s, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
```

Multicast Options

IP multicasting is supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW and only on networks in which the interface driver supports multicasting. The following are multicast options:

IP_ADD_MEMBERSHIP
: Places a host in a multicast group. A host must become a member of a multicast group before it can receive datagrams sent to the group. An example follows:

```
struct ip_mreq mreq;
setsockopt (s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

In the previous example, `mreq` has the following structure:

```
struct ip_mreq {
     struct in_addr imr_multiaddr;  /* multicast group to join */
     struct in_addr imr_interface;  /* interface to join on */
}
```

The `imr_interface` parameter should be `INADDR_ANY` to choose the default multicast interface, or the IP address of a particular multicast-capable interface if the host is multihomed. Membership is associated with one interface; programs that run on multihomed hosts may have to join the same group on more than one interface. You may add up to the specified value in the `IP_MAX_MEMBERSHIPS` option on one socket.

`IP_DROP_MEMBERSHIP`

Drops a membership in a multicast group. The `mreq` parameter contains the same values as those used to add the membership. An example follows:

```
struct ip_mreq mreq;
setsockopt (s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));+
```

When the socket is closed or the process exits, memberships also are dropped.

`IP_MULTICAST_IF`

Specifies the interface on which each multicast transmission is sent. The following is an example of this option:

```
struct in_addr addr;
setsockopt (s, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr));
```

In this example, you can use `addr` to specify the local address of the desired interface, or `INADDR_ANY` to specify the default interface. To obtain the local IP address and multicast capability of an interface, use the `ioctl`(2) system calls `SIOCGIFCONF` and `SIOCGIFFLAGS`. Most applications do not have to use this option.

`IP_MULTICAST_LOOP`

Gives the sender explicit control over whether subsequent datagrams are looped back. The following is an example of this option:

```
u_char loop;          /* 0 = disable, 1 = enable (default) */
setsockopt (s, IPPROTO-IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. This option improves performance for applications that may have no more than one instance on a single host (such as a router daemon) by eliminating the overhead of receiving their own transmissions. Generally, applications for which more than one instance may be on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time-querying program) should not use this option.

A multicast datagram sent with an inital TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent if the host belongs to the destination group on that other interface. The loopback control option does not affect such delivery.

`IP_MULTICAST_TTL`

Changes the TTL for outgoing multicast datagrams to control the scope of the multicasts. The following example uses this option:

```
u_char ttl;         /* range:  0 to 255, default = 1 */
setsockopt (s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

Datagrams with a TTL of 1 are not forwarded beyond the local network. Multicast datagrams with a TTL of 0 are not transmitted on any network, but they may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket. If a multicast router is attached to the local network, multicast datagrams with a TTL greater than 1 may be forwarded to other networks.

Raw IP Options

Raw IP sockets are connectionless, and they usually are used with the `sendto`(2) and `recvfrom`(2) system calls. The `connect`(2) system call also may fix the destination for future packets, in which case, you may use the `read`(2) or `recv`(2), and `write`(2) or `send`(2) system calls.

If the *protocol* argument is 0, the default protocol `IPPROTO_RAW` is used for outgoing packets, and only incoming packets destined for that protocol number are received. If `proto` is a nonzero value, that value is used on outgoing packets and is used to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number with which the socket is created), unless the `IP_HDRINCL` option has been set. Incoming packets are received with header and options intact. A raw IP option follows:

`IP_HDRINCL`

Indicates the complete IP header is included with the data. You may use this option only with the `SOCK_RAW` type. An example follows:

```
#include <netinet/ip.h>
int hincl = 1;
setsockopt (s, IPPROTO_IP, IP_HDRINCL, &hincl, sizeof(hincl));
```

Unlike previous UNICOS releases, the program must set all of the fields of the IP header, including the following:

```
ip->ip_v = IPVERSION;
ip->ip_hl = hlen >> 2;
ip->ip_id = 0;  /* 1 = on, 0 = off */
ip->ip_off = offset;
```

If the header source address is set to the kernel, the program chooses an appropriate address.

NOTES

Only the super user may create an IP socket.

IP provides no mechanism for out-of-band data.

MESSAGES

If a socket operation fails, it returns one of the following error messages:

EADDRNOTAVAIL
: The process tried to create a socket by using a network address for which no network interface exists.

EISCONN
: The socket already has a connection when a connection is tried, or the process is trying to send a datagram with the destination address specified when the socket is already connected.

ENOBUFS
: The system ran out of memory for an internal data structure.

ENOTCONN
: The process is trying to send a datagram, but no destination address has been specified and the socket is not already connected.

When you are setting or getting IP options, the following errors specific to IP can occur:

[EINVAL]
: An unknown socket option name was specified.

[EINVAL]
: An unknown socket descriptor was specified.

[EINVAL]
: The IP option field was improperly formed (for example, an option field was shorter than the minimum value or longer than the option buffer provided).

FILES

/usr/include/netinet/in.h	Include file for Internet addresses
/usr/include/netinet/ip.h	Defines the IP header structure
/usr/include/sys/socket.h	Defines address families

SEE ALSO

icmp(4P), igmp(4P), inet(4P), intro(4P), tcp(4P), udp(4P)

accept(2), connect(2), getsockopt(2), listen(2), read(2), recv(2), recvfrom(2), send(2), socket(2), write(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

RFC 791

NAME

nfs – UNICOS network file system (UNICOS NFS)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The UNICOS network file system (UNICOS NFS) is an implementation of the network file system (NFS) for Cray Research systems running UNICOS. NFS was originally designed and developed to reduce the need for local disk storage in distributed environments. NFS executes on a wide variety of computing hardware and is implemented on operating systems other than its development home in the Berkeley UNIX environment.

NFS allows file systems to be shared across a network of machines. The standard system calls for file operations (close(2), open(2), read(2), and write(2)) system calls are used to access both local and network-based files; the location of files can be transparent to the user. Standard permission mechanisms are used to control file access.

The user interface to NFS is transparent; that is, there is no user interface. After a system is configured, users simply read and write their files, whether they are on local disk or exist elsewhere on the network.

Client and Server Modes

Client NFS allows users to make standard system calls (close(2), create(2), delete(3C), open(2), read(2), and write(2)) to a portion of file name space on which a network file system has been mounted (by using mount(8)). These calls are intercepted by a process in the client system, translated into the corresponding NFS requests, and packaged for transmission across a network to a server machine. With UNICOS NFS, the file system switch (FSS) facilitates mapping between local requests and NFS requests.

Server NFS implementations allow a portion of their local file system to be exported (made available for remote mounting). When NFS requests for exported file systems are received, the server performs the indicated operation; about 20 file system operations are supported. In the case of read or write requests, the indicated data is returned to the remote NFS client (for a read operation), or written to local disk (for a write operation).

Client and server implementations are logically separate. Some implementations (for example, the implementation of NFS for MS-DOS) are client only; that is, they can make use of remote systems but cannot export file systems of their own. Other implementations (for example, the implementation of NFS for VMS) are server only; they can respond to requests from client systems but do not allow remote file systems to be mounted on their local namespace. UNICOS NFS includes both client and server modes.

A server can grant access to a specific file system to certain clients by adding an entry for that file system to the server's /etc/exports file (see exports(5)). A client gains access to that file system by using the mount(2) system call, which requests a file handle for the file system itself. After the client mounts the file system, the server issues a file handle to the client for each file (or directory) the client accesses. If the file is somehow removed on the server side, the file handle becomes stale (disassociated with a known file).

A client cannot export file systems that it has mounted over the network; therefore, clients must mount file systems directly from the server on which the file systems reside. The user ID (UID) and group ID (GID) mappings must be the same between client and server; however, the server maps UID 0 (the super user) to UID-2 before performing access checks for a client. This inhibits super-user privileges on remote file systems.

Network Interface (RPC, XDR, and UDP/IP)

NFS is a set of high-level protocols, based on a remote procedure call (RPC) model; NFS network requests are made through calls to remote procedures that implement NFS file system semantics. (The libraries that contain these RPC procedures have been available since the UNICOS 2.0 release.)

RPC makes use of an intermediate data representation for all information sent to and received from the network. This intermediate form is called *External data representation* (XDR). (The libraries that contain these XDR procedures have been available since the UNICOS 2.0 release.)

Stateless Servers

Within NFS, all state information (such as open file status) is maintained by the client implementations, while the servers are said to be *stateless*. Servers are thus relatively simple, and recovery operations are more reliable than with *stateful* servers.

Mount and Lock Managers

Managers handle operations that are related to particular implementations of file systems, such as UNIX or UNICOS file systems, but are not deemed to be universal operations. Currently, two managers are defined for NFS: one handles the mount protocol; the other handles file locking. The managers typically run as user-level processes, and they communicate with the kernel implementation in very carefully defined ways.

NOTES

UNICOS NFS is a licensed product that also requires UNICOS and UNICOS TCP/IP licenses. Therefore, UNICOS NFS may not be available at your site.

In most NFS implementations, user ID (UID) and group ID (GID) values are the same between client and server. The network information service (NIS) distributed data lookup service is often used to manage `passwd`(5) and `group`(5) files to ensure consistency across an entire NFS domain. For more information about using the network information service (NIS) feature, see the *UNICOS Networking Facilities Administrator's Guide*, Cray Research publication SG–2304.

UNICOS NFS sites also can use the ID mapping facility, which provides for the operation of UNICOS NFS in environments that are not administratively homogeneous. For more information, see the *UNICOS Networking Facilities Administrator's Guide*, Cray Research publication SG–2304. Typically, the server maps UID 0 (root) to UID-2 before performing access checks for a client.

MESSAGES

Generally, physical disk I/O errors detected on the server are returned to the client for action. If the server is down or inaccessible, the client sees the following console message:

```
NFS:  file server not responding: still trying.
```

The client continues to send the request until it receives an acknowledgment from the server. This means that the server can crash (or power down) and come back up without any special action required by the client. It also means that the client process requesting the I/O will block and remain insensitive to signals, sleeping inside the kernel at priority `PRIBIO`.

SEE ALSO

`exports`(5), `fstab`(5)

`mount`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`intro`(3C), `rpc`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`mount`(8), `nfsaddmap`(8), `nfsclear`(8), `nfsd`(8), `nfslist`(8), `nfsmerge`(8), `nfsstat`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

NAME

`route` – Kernel packet forwarding database

SYNOPSIS

```
#include <sys/socket.h>
#include <net/if.h>
#include <net/route.h>

int family
s = socket(PF_ROUTE, SOCK_RAW, family);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `route` facility performs packet routing services. The kernel maintains a routing information database known as a *routing table*, which selects the appropriate network interface when packets are transmitted. A user process (or possibly multiple cooperating processes) maintains this routing table by sending messages over a special kind of socket. The use of this routing table supersedes the use of the fixed-size `ioctl`, implemented in earlier releases. Only the super user can make changes to the routing table.

The operating system might spontaneously emit routing messages in response to external events, such as receipt of a redirect, or failure to locate a suitable route for a request.

Routing table entries can exist for a specific host or for all hosts on a generic subnetwork (as specified by a bit mask and value under the mask). To achieve the effect of wildcard or default routes, use a mask of all 0's. Some routes might be hierarchical.

When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. Usually, the protocol specifies the route through each interface as a direct connection to the destination host or network. If the route is specified as direct, the transport layer of a protocol family usually requests that the packet be sent to the same host specified in the packet. Otherwise, it requests the interface to address the packet to the gateway listed in the routing entry (that is, the packet is forwarded).

When the kernel is routing a packet, it first tries to find a route to the destination host. Failing that, it makes a search for a route to the network of the destination. Finally, it chooses any route to a default (or wildcard) gateway. If no entry is found, the destination is declared to be unreachable, and if any processes are listening for messages on the routing socket, a routing-miss message is generated.

A wildcard routing entry is specified with a destination address value of 0. Wildcard routes are used only when the system does not find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

You can open the channel for passing routing control messages by using the socket call shown in the SYNOPSIS section. You can designate the *family* argument to be AF_UNSPEC, which provides routing information for all address families, or you can restrict it to a specific address family by designating a specific *family* argument. You can have more than one routing socket open per system.

Messages are formed with a header followed by a small number of socket addresses (sockaddr fields), interpreted by position, and delimited by the length entry in the sockaddr field (this length is variable).

A bit mask within the header specifies which address is present; the position sequence is least-significant to most-significant bit within the vector. The kernel returns any messages it receives, and copies are sent to all interested listeners. The kernel provides the process ID for the sender; the sender can use an additional sequence field to distinguish between outstanding messages. However, when kernel buffers are exhausted, message replies might be lost.

The kernel can reject certain messages; it indicates rejection by filling in the rtm_errno field. The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a nonexistent entry, or ENOBUFS if insufficient resources were available to install a new route. In the current implementation, all routing processes run locally; the values for rtm_errno are available through the normal errno mechanism, even if the routing reply message is lost.

A process can avoid the expense of reading replies to its own messages by issuing a setsockopt() call (see getsockopt(2)), indicating that the SO_USELOOPBACK option at the SOL_SOCKET level will be turned off. A process can ignore all messages from the routing socket by issuing a shutdown(2) system call for further input.

If a route is in use when it is deleted, the routing entry is marked down and removed from the routing table, but the resources associated with it are not reclaimed until all references to it are released. User processes can obtain information about the routing entry to a specific destination by using a RTM_GET message, or by calling the sysctl routine.

Messages

Following is a list of the messages and their meanings that the routing facility generates:

```
#define RTM_ADD      0x1 /* Add route */
#define RTM_DELETE   0x2 /* Delete route */
#define RTM_CHANGE   0x3 /* Change metrics, flags, or gateway */
#define RTM_GET      0x4 /* Report information */
#define RTM_LOSING   0x5 /* Kernel suspects partitioning */
#define RTM_REDIRECT 0x6 /* Told to use different route */
#define RTM_MISS     0x7 /* Lookup failed on this address */
#define RTM_RESOLVE  0xb /* Request to resolve dst to LL addr */
#define RTM_LOCK     0x8 /* Lock metric values */
```

Message Headers

An example of a message header follows:

```
struct rt_msghdr {
        u_short rmt_msglen;         /* Skip nonunderstood messages */
        u_char  rtm_version;        /* Future binary compatibility */
        u_char  rtm_type;           /* Message type */
        u_short rmt_index;          /* Index for associated ifp  */
        pid_t   rmt_pid;            /* Identify sender */
        int     rtm_addrs;          /* Bit mask for sockaddrs in msg */
        int     rtm_seq;            /* For sender to identify action */
        rtm_errno;                  /* Why failed
        int     rtm_flags;          /* Kernel and message flags */
        int     rtm_use;            /* From rtentry */
        u_long  rtm_inits;          /* Values to be initialized */
        struct  rt_metrics rtm_rmx; /* Metrics themselves */
};
```

Metrics Structure

The structure for the metrics is as follows:

```
struct rt_metrics {
        u_long rmx_locks;     /* Kernel must leave these values alone */
        u_long rmx_mtu;       /* MTU for this path */
        u_long rmx_hopcount;  /* Max hops expected */
        u_long rmx_expire;    /* Lifetime for route ( e.g., redirect) */
        u_long rmx_recvpipe;  /* Inbound delay-bandwith product */
        u_long rmx_sendpipe;  /* Outbound delay-bandwith product */
        u_long rmx_ssthresh;  /* Outbound gateway buffer limit */
        u_long rmx_rtt;       /* Estimated round-trip time */
        u_long rmx_rttvar;    /* Estimated rtt variance */
};
```

Flags

Flags include the following values:

```
#define RTF_UP        0x1     /* Route usable */
#define RTF_GATEWAY   0x2     /* Destination is a gateway */
#define RTF_HOST      0x4     /* Host entry (net otherwise) */
#define RTF_REJECT    0x8     /* Host or net unreachable */
#define RTF_DYNAMIC   0x10    /* Created dynamically (by redirect) */
#define RTF_MODIFIED  0x20    /* Modified dynamically (by redirect) */
#define RTF_DONE      0x40    /* Message confirmed */
#define RTF_MASK      0x80    /* Subnet mask present */
#define RTF_CLONING   0x100   /* Generate new routes on use */
#define RTF_LLINFO    0x400   /* Generated by ARP or ESIS */
#define RTF_STATIC    0x800   /* Manually added */
#define RTF_NOFORWARD 0x1000  /* Do not forward through */
#define RTF_EXCLGID   0x2000  /* gid list is exclusive */
#define RTF_PROTO2    0x4000  /* Protocol-specific routing flag */
#define RTF_PROTO1    0x8000  /* Protocol-specific routing flag */
#define RTF_TOSMATCH  0x10000 /* High-level match required for TOS */
#define RTF_NOMTUDISC 0x40000 /* Do not do path MTU discovery */
```

Metric Value Specifiers

Specifiers for metric values in `rmx_locks` and `rtm_inits` are as follows:

```
#define RTV_MTU      0x1   /* Initialize or lock _mtu */
#define RTV_HOPCOUNT 0x2   /* Initialize or lock _hopcount */
#define RTV_EXPIRE   0x4   /* Initialize or lock _hopcount */
#define RTV_RPIPE    0x8   /* Initialize or lock _recvpipe */
#define RTV_SPIPE    0x10  /* Initialize or lock _sendpipe */
#define RTV_SSTHRESH 0x20  /* Initialize or lock _ssthresh */
#define RTV_RTT      0x40  /* Initialize or lock _rtt */
#define RTV_RTTVAR   0x80  /* Initialize or lock _rttvar */
```

Address Specifiers

Specifiers for which addresses are present in the messages are as follows:

```
#define RTA_DST     0x1    /* Destination sockaddr present */
#define RTA_GATEWAY 0x2    /* Gateway sockaddr present */
#define RTA_NETMASK 0x4    /* Netmask sockaddr present */
#define RTA_GENMASK 0x8    /* Cloning mask sockaddr present */
#define RTA_IFP     0x10   /* Interface name sockaddr present */
#define RTA_IFA     0x20   /* Interface addr sockaddr present */
#define RTA_AUTHOR  0x40   /* sockaddr for author of redirect */
```

SEE ALSO

`getsockopt`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`route`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

tcp – Transmission Control Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_STREAM, 0);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Transmission Control Protocol (TCP) provides reliable, flow-controlled, two-way byte streams between pairs of programs running on hosts in an Internet Protocol (IP) network. The protocol is defined in *DARPA Internet Request for Comments*, RFC 793.

TCP allows for multiple byte streams between a pair of hosts by associating each connection on a host with a port number. The Internet addresses of the connected hosts and two port numbers, one on each host, uniquely identifies a connection. Port numbers are integers, specified in the bind(2) or connect(2) system call, ranging in value from 1 to 65,535. By Internet convention, some ports are reserved for well-known services (for example, hosts provide TELNET service by listening for connections on port 23). These services are listed in *DARPA Internet Request for Comments*, RFC 1010. By Berkeley UNIX convention, only the super user may bind to ports that have numbers lower than 1024.

Connection

Sockets that use the TCP protocol are either active or passive. *Active sockets* initiate connections to passive sockets. TCP sockets are created as active sockets; to create a passive socket, you must have the listen(2) system call after the socket is bound to a port that has the bind(2) system call. Only passive sockets may use the accept(2) system call to accept incoming connections. Only active sockets may use the connect(2) system call to initiate connections. After a socket has been made passive, it cannot be made active again.

Passive sockets may ‘‘underspecify’’ their location to match incoming connection requests from multiple networks. This technique, termed *wildcard addressing*, allows one server to provide service to clients on multiple networks. To create a socket that listens on all networks, bind the socket to the Internet address INADDR_ANY (defined in the netinet/in.h include file). The TCP port may still be specified at this time; if you do not specify the port, the system will assign one.

Transmission

TCP is a byte-stream protocol with connections; write(2) is the usual method of sending on a TCP socket. The send(2) system call is useful for sending out-of-band data. The sendto(2) system call (see send(2)) works, but it is misleading because the destination address is ignored.

TCP is a byte stream, not a word stream. Although you can send data types other than bytes over a TCP connection, for example, by passing the address of an integer to `write`(2), no guarantee exists that an integer will come out at the other end. The receiving host may have a different word size, a different byte order, or other incompatibilities.

Reception

The `read`(2) system call is the usual method of receiving on a TCP socket. The `recv`(2) system call is useful for receiving out-of-band data. The `recvfrom`(2) system call (see `recv`(2)) works, but it is misleading because the source address is ignored.

Because TCP presents data from a socket in arbitrary chunks as the data becomes available from the Internetwork, TCP sockets are more likely than regular files to return fewer bytes than requested. Be sure to check the return value from `read`(2).

Disconnection

Executing the `shutdown`(2) system call on a TCP socket before the `close`(2) system call allows you to close down one side of the connection.

Options

The `SO_KEEPALIVE` option (defined in the `sys/socket.h` include file) causes the code in the kernel that handles TCP protocol periodically to send packets that contain no data; these packets are acknowledged and discarded. This tests whether the data path to the other end of the connection is open. If the other end fails to respond to these keep-alive packets, the connection will be closed and the next socket operation will return `ETIMEDOUT`.

Other options have their socket-level effects; the socket-level effect is explained in the `getsockopt`(2) system call.

You can use options at the IP transport level with TCP (see `ip`(4P).

TCP supports several socket options that you can set by using `setsockopt`(2) and test by using `getsockopt`(2). The option level for the `setsockopt`(2) call is the protocol number for TCP, available from `getprotobyname`(3C).

Most socket-level options take an `int` type value. For `setsockopt`(2), the value must be nonzero to enable a Boolean option, or 0 to indicate that the option will be disabled. You can use the following options with TCP:

Option	Description
`TCP_MAXSEG`	Gets maximum segment size. You cannot set this option.
`TCP_NODELAY`	Toggles the `no delay` flag. Under most circumstances, TCP sends data when it is presented. When outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in one packet after an acknowledgment has been received. For a few clients (such as window systems that send a stream of mouse events that receive no replies), this packetization might cause significant delays. TCP provides this Boolean option to defeat this algorithm.

TCP_WINSHIFT — Sets the TCP window shift count. You must set this option on a socket before the use of the connect(2) or accept(2) system call. A value of –1 turns off window shift; a value of 0 through 14 turns on window shift with the requested window size. getsockopt(2) returns up to 24 bytes (the TR_SENDWNDSHIFT value in the first word, the send window shift value in the second word, and the receive window shift value in the third word).

NOTES

Only the super user may bind a socket to a port number lower than 1024.

MESSAGES

A socket operation may fail with one of the following values in errno:

EADDRINUSE
: An attempt is made to create a socket by using a port that has already been allocated.

EADDRNOTAVAIL
: The process tried to create a socket with a network address for which no network interface exists.

ECONNREFUSED
: The remote peer actively refuses connection establishment (usually because no process is listening to the port).

ECONNRESET
: The remote peer forces the connection to be closed.

EINVAL
: An option value or socket that is not valid was specified for the setsockopt(2) system call.

EISCONN
: The socket already has a connection when a connection is tried on the connect(2) system call, or you cannot use the setsockopt(2) TCP_WINSHIFT option on an established connection.

ENOBUFS
: The system ran out of memory for an internal data structure.

ETIMEDOUT
: A connection was dropped because of excessive retransmissions.

FILES

/usr/include/netinet/in.h	Include file for Internet addresses
/usr/include/netinet/tcp.h	Include file for TCP addresses
/usr/include/sys/socket.h	Address family definition

SEE ALSO

inet(4P), intro(4P), ip(4P)

accept(2), bind(2), close(2), connect(2), getsockopt(2), listen(2), read(2), recv(2), setsockopt(2), shutdown(2), write(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

getprotobyname(3c) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

DARPA Internet Request for Comments, RFC 793 and RFC 1010

NAME

udp – Internet User Datagram Protocol

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The Internet User Datagram Protocol (UDP) is a simple datagram protocol that tftp(1B) and the rpc(3C) library routines use. The protocol is defined in *DARPA Internet Request for Comments*, RFC 768.

UDP allows for multiple endpoints on a host by associating each endpoint with a port number. Port numbers are integers, specified by the bind(2) or connect(2) system call, ranging in value from 1 to 65,535. By Internet convention, some ports are reserved for well-known services, listed in *DARPA Internet Request for Comments*, RFC 1010; for example, hosts provide TFTP (see tftp(1B)) service by listening for datagrams on port 69. By Berkeley UNIX convention, only the super user may listen on ports with numbers fewer than 1024.

Transmission

UDP is a datagram protocol, therefore, a UDP socket has no connections; that is, the listen(2) and accept(2) system call return errors. The sendto(2) system call (see send(2)) is the normal method for transmission through a UDP socket. The connect(2) system call, though not supported by the underlying protocol, permanently associates the socket with a destination for future transmissions, thus enabling use of the send(2) system call and the write(2) system call on the socket.

The entire contents of a transmission (the buffer in a sendto(2), send(2), or write(2) system call) are packaged into the data portion of one datagram for transmission. The kernel provides the UDP and Internet Protocol (IP) headers. You can configure the kernel to calculate the UDP data checksum; if not, UDP packets will go out without protection against transmission errors.

Reception

The recvfrom(2) system call (see recv(2)) is the normal method for receiving data through a UDP socket. The connect(2) system call, though not supported by the underlying protocol, permanently associates the socket with a destination for future transmissions, thus enabling use of the recv(2) system call and the read(2) system call on the socket.

The kernel checks the UDP data checksum in arriving datagrams and discards garbled datagrams without presenting them to the user through the socket.

If you do not provide enough buffer space for the entire available datagram in a call to `recvfrom`(2), `read`(2), or `recv`(2), excess bytes at the end of the datagram will be discarded without notice.

MESSAGES

A socket operation may fail with one of the following errors returned:

`EADDRINUSE`
: The process tries to create a socket by using a port that has already been allocated.

`EADDRNOTAVAIL`
: The process tried to create a socket with a network address for which no network interface exists.

`EISCONN`
: The socket already has a connection when a connection is tried, or the process is trying to send a datagram with the destination address specified when the socket is already connected.

`ENOBUFS`
: The system ran out of memory for an internal data structure.

`ENOTCONN`
: The process is trying to send a datagram, but no destination address has been specified and the socket is not already connected.

FILES

`/usr/include/netinet/in.h`	Include file for Internet addresses
`/usr/include/netinet/udp.h`	UDP header file
`/usr/include/sys/socket.h`	Address family definition

SEE ALSO

`inet`(4P), `intro`(4P), `tcp`(4P)

`tftp`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`accept`(2), `bind`(2), `connect`(2), `listen`(2), `read`(2), `recv`(2), `send`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`intro_svc`(3R) in the *Remote Procedure Call (RPC) Reference Manual*, Cray Research publication SR–2089

DARPA Internet Request for Comments, RFC 768 and RFC 1010

NAME

intro – Introduction to file formats

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Section 5 outlines the formats of certain UNICOS files. These files include header files, data files, and output files from UNICOS utilities. The files in this section fall into one of three categories:

- User category
- Administrator category
- Analyst category

Within each of these categories, there are two subdivisions; files a user, administrator, or kernel process can change, and files used as a reference, template, or data file.

The following three tables detail the division of the entries in section 5. Some entries fall into more than one category; these are entries that a user references, but an administrator changes. For example, the group(5) entry describes the /etc/group file, which can be viewed by a user and changed by an administrator.

Many entries in this section describe *header files*. Header files are files in a specific format that more than one program (such as compilers, assemblers, and system utilities) use, often for data interchange between programs. You must enter the names of header files in the predefined format that is shown on the man page. When applicable, the C struct declarations for the file formats are given. In this manual, header files are referred to as *include files* because they usually are found in the /usr/include or /usr/include/sys directory. In the DESCRIPTION section of the entries, the full path name for a header file is given only when it is not in either of these directories.

User Category

The man pages in this category describe entries of interest to UNICOS users. An * symbol marks files that users set up or modify.

Man page	Description	File(s)
aliases	Define alias database for sendmail(8)	/usr/lib/aliases
a.out	Loader output file	/usr/include/a.out.h
ar	Archive file format	/usr/include/ar.h
bld	Relocatable library files format	/usr/src/cmd/bld/bld.h
cpio	cpio(1) archive file format	
cshrc	C shell start-up and termination files	.cshrc*, .login*, .logout*
def_seg	Loader directives files	/lib/segdirs/def_seg*
exrc	Start-up files for ex(1) and vi(1)	.exrc*

Man page	Description	File(s)
fcntl	File control options	/usr/include/fcntl.h
group	Group-information file format	/etc/group
mailrc	Start-up files for mailx(1)	.mailrc*
motd	File that contains message of the day	/etc/motd
netrc	TCP/IP autologin information file for outbound ftp(1B) requests	$HOME/.netrc*
nl_types	Defines message system variables	nl_types.h
passwd	Password file format	/etc/passwd
profile	Format of Posix shell start-up file	.profile*
publickey	Public key database	/etc/publickey*
relo	Relocatable object table format under UNICOS	/usr/include/relo.h
rhosts	List of trusted remote hosts and account names	.rhosts*
sccsfile	Source Code Control System (SCCS) file format	s.*file*
symbol	UNICOS symbol table entry format	/usr/include/symbol.h
tapetrace	Tape daemon trace file format	/usr/spool/tape/trace.daemon /usr/spool/tape/trace.bmx*xxx* /usr/include/tapereq.h
taskcom	Task common table format	
types	Definition of primitive system data types	/usr/include/sys/types.h
updaters	Configuration file for NIS updating	/etc/yp/updaters*
uuencode	Encoded uuencode file format	
values	Machine-dependent values definition file	/usr/include/values.h

Administrator Category

The man pages in this category describe files of interest to an administrator. An * symbol marks files that the administrator modifies.

Man page	Description	File(s)
acid	Account ID information file format	/etc/acid
acl	User access control lists format	/usr/include/sys/acl.h
aft	ASCII flaw table	/etc/aft/*
confval	Configuration file for various products	
cshrc	C shell start-up and termination files	/etc/cshrc*
dump	Incremental file system dump format	/usr/src/cmd/fs/dump
exports	Directories to export to NFS clients	/etc/exports*

Man page	Description	File(s)
fslrec	File system error log record format	/dev/fslog /usr/include/sys/fslog.h /usr/include/sys/fslrec.h /usr/include/sys/types.h
fstab	File that contains static information about file systems	/etc/fstab*
ftpusers	List of unacceptable ftp(1B) users	/etc/ftpusers*
gated-config	Gated configuration file syntax	/etc/gated.conf
gettydefs	Speed and terminal settings used by getty(8)	/etc/gettydefs
group	Group-information file format	/etc/group*
hosts	TCP/IP host name database	/etc/hosts*
hosts.equiv	Public information for validating remote autologin	/etc/hosts.equiv*
inetd.conf	Internet super-server configuration file	/etc/inetd.conf*
inittab	Script for init process	/etc/inittab*
iptos	IP Type-of-Service database	/etc/iptos
issue	Login message file	/etc/issue*
krb.conf	Kerberos configuration file	
krb.realms	Host to Kerberos realm translation file	
ldesc	Logical disk descriptor file	/usr/include/sys/ldesc.h
mailrc	Start-up files for mailx(1)	/usr/lib/mailx/mailx.rc*
masterfile	Internet domain name server master data file	
mib.txt	Management information base for SNMP applications and SNMP agents	/etc/mib.txt
mnttab	Mounted file system table format	/etc/mnttab
motd	File that contains message of the day	/etc/motd*
named.boot	Domain name server configuration file	/etc/named.boot*
netgroup	List of network groups	/etc/netgroup
networks	Network name database	/etc/networks*
nl_types	Defines message system variables	nl_types.h
passwd	Password file format	/etc/passwd*
printcap	Printer capability database	/etc/printcap*
profile	Format of POSIX shell start-up file	/etc/profile*
proto	Prototype job file for at	/usr/lib/cron/.proto
protocols	Protocol name database	/etc/protocols*
publickey	Public key database	/etc/publickey*
queuedefs	Queue description file for at, batch, and cron	/usr/lib/cron/queuedefs

Man page	Description	File(s)
quota	Quota control file format	/sys/quota.h*
resolv.conf	Domain name resolver configuration file	/etc/resolv.conf*
rmtab	List of remotely mounted file systems	/etc/rmtab
sectab	Format for table of defined security names and values	/usr/include/sys/sectab.h
sendmail.cf	Configuration file for TCP/IP mail service	/usr/lib/sendmail.cf*
services	Network service name database	/etc/services*
share	Fair-share scheduler parameter table	/usr/include/sys/share.h
shells	List of available user shells	/etc/shells
text_tapeconfig	Tape subsystem configuration file	/etc/config/text_tapeconfig*
tapereq	User tape daemon interface	/usr/include/tapereq.h
tar	Tape archive file format	
term	Format of compiled term file	/usr/include/term.h
terminfo	Terminal capability database	/usr/lib/terminfo/*
tmpdir.users	List of authorized users for tmpdir(1)	/etc/tmpdir.users*
udb	Format of the user database file	/etc/udb /etc/udb.public
updaters	Configuration file for NIS updating	/etc/yp/updaters
utmp	utmp(5) and wtmp file formats	/etc/utmp /etc/wmtp
ypfiles	Network information service (NIS) database and directory structure	

Analyst Category

The man pages in this category describe files of interest to Cray Research analysts. Man pages in this subcategory describe internal files, including those that the UNICOS kernel uses as a reference. Cray Research or customer analysts do not change these files.

The files that an analyst sets up or modifies (to install or configure a UNICOS system) are not described in this manual; for more information on these files, see *General UNICOS System Administration*, Cray Research publication SG–2301.

Man page	Description	File(s)
acct	Per-process accounting file format	/usr/include/sys/acct.h
core	Core file format	
dir	Directory file format	/usr/include/sys/fs/c1dir.h

Man page	Description	File(s)
dirent	File-system-independent directory entry format	/usr/include/sys/dirent.h
errfile	Format of error-log file	/usr/adm/errfile /usr/include/sys/err.h
fs	File system partition format	/usr/include/sys/fs/c1filsys.h
infoblk	Loader information table	/usr/include/infoblk.h
inode	Inode format	/usr/include/sys/ino.h
lnode	Kernel user limits structure for fair-share scheduler	/usr/include/sys/lnode.h
ipc	Interprocess communication (IPC) access structures	/usr/include/sys/ipc.h
msg	Message queue structures	/usr/include/sys/msg.h
sem	Semaphore facility	/usr/include/sys/sem.h
shm	Shared memory facility	/usr/include/sys/shm.h
slrec	Security log record format	/usr/include/sys/slrec.h
sysdump	System dump files	/core.sys

NAME

acct – Per-process accounting file format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/accthdr.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Accounting files are produced if the acct(2) system call has enabled the system process accounting routine. The sys/acct.h include file gives the structure of these files.

Systems Accounting File Structure

The file structures differ slightly for accounting records on various Cray Research systems; the differences are indicated in the following file structures:

```
/*
 *     Kernel accounting structures.
 *
 *     Note: All of the structures in the unified accounting record
 *           must have the ac_flag field following the header.
 *
 */
typedef unsigned long comp_t;   /* 21-bit floating-point number   */
                                /* 5-bit exponent, 16-bit         */

/*
 *     Base-level accounting record.
 */
struct acctbs
{
        struct achead   ac_header;      /* header                       */
        unsigned        ac_flag:8;      /* accounting flags             */
        unsigned        ac_stat:8;      /* exit status                  */
        unsigned        ac_uid:24;      /* user ID                      */
        unsigned        ac_gid:24;      /* group ID                     */
        dev_t           ac_tty:32;      /* control typewriter           */
        time_t          ac_btime:32;    /* beginning time (seconds)     */
        comp_t          ac_utime:21;    /* user CPU time (clocks)       */
        comp_t          ac_stime:21;    /* system CPU time (clocks)     */
        comp_t          ac_etime:21;    /* elapsed time (clocks)        */
        comp_t          ac_mem:21;      /* 1st memory integral          */
                                        /* (click-tics)                 */
```

```
        comp_t          ac_mem2:21;     /* 2nd memory integral            */
                                        /* (click-tics)                   */
        comp_t          ac_mem3:21;     /* 3rd memory integral            */
                                        /* (click-tics)                   */
        comp_t          ac_io:21;       /* number of chars transferred    */
        comp_t          ac_rw:21;       /* number of physical I/O reqsts. */
        comp_t          ac_iowtime:21;  /* I/O wait time (clocks)         */
                                        /* runs while process is locked   */
                                        /* in memory                      */
        comp_t          ac_iowmem:21;   /* I/O wait time memory integral  */
                                        /* (click-tics) runs while        */
                                        /* process is locked in memory    */
        comp_t          ac_iosw:21;     /* I/O swap count                 */
        comp_t          ac_lio:21;      /* number of logical I/O requests */
        unsigned        ac_pid:21;      /* process ID                     */
        unsigned        ac_ppid:21;     /* parent process ID              */
        comp_t          ac_ctime:21;    /* process connect time (clocks)  */
        unsigned        ac_acid:24;     /* account ID                     */
        unsigned        ac_jobid:24;    /* job ID                         */
        unsigned        ac_nice:16;     /* nice value                     */
        char            ac_comm[8];     /* command name                   */
        comp_t          ac_iobtim:21;   /* I/O wait time (clocks)         */
        comp_t          ac_himem:21;    /* Hiwater memory mark (words)    */
        comp_t          ac_sctime:21;   /* system call time               */
};

/*
 *      End of Job record. Written when the last process is put to rest.
 */
struct  accteoj {
        struct achead   ac_header;              /* header                    */
        unsigned        ac_flag:8;              /* accounting flag           */
        unsigned        ace_jobid:24;           /* Job ID                    */
        unsigned        ace_uid:24;             /* User ID                   */
        comp_t          ace_himem:21;           /* Hiwater mem. mark(clicks)*/
        comp_t          ace_sdshiwat:21;        /* SDS Hiwater mark          */
        unsigned        ace_nice:16;            /* Nice value                */
        long            ace_fsblkused;          /* #of fs blocks consumed    */
        time_t          ace_etime:32;           /* time at end of job        */
        comp_t          ace_shmint:21;          /* shmat integral (click-tics)*/
        comp_t          ace_shmsize:21;         /* shmget total size (words) */
};

/*
 *      Device specific I/O accounting record
 *      type field is filled in from superblock on block devices and
 *      from major number | ACCT_CHSP when used with a character device.
 */
```

```
  #define NODEVACCT     8              /* devio entries per account*/
                                       /* records                  */
  #define ACCT_CHSP     0200           /* marker for character     */
                                       /* special devices          */
  #define ACCT_PERF     0400           /* marker for performance   */
                                       /* accounting               */
  #define MAXPERFLVL    1              /* number of performance    */
                                       /* accounting levels        */

  struct acctio {
         struct achead ac_header;      /* header                   */
         unsigned     ac_flag:8;       /* accounting flags         */
         struct {
                uint   acd_type:8;     /* major device no.         */
                comp_t acd_ioch:21;    /* characters transferred   */
                comp_t acd_lio:21;     /* logical I/O reqs count   */
         } ac_devio[NODEVACCT];
  };

/*
 *     SDS accounting record (except on CRAY EL series)
 */
struct acctsds {
       struct achead  ac_header;       /* header                   */
       char           ac_flag;         /* accounting flag          */
       comp_t         acs_mem:21;      /* memory integral - based */
                                       /* on residency time,       */
                                       /* not execution time       */
       comp_t         acs_lio:21;      /* logical I/O reqs count  */
       comp_t         acs_ioch:21;     /* chars transferred        */
       comp_t         acs_memsw:21;    /* mem integral - suspend/resume */
};
```

The following MPP accounting record of the `acct` file is for use only with Cray MPP systems:

```
    /*
     *      MPP accounting record.
     */
    struct  acctmpp {
            struct achead   ac_header;      /* header                   */
            char            ac_flag;        /* accounting flag          */
            unsigned        ac_mpppe:16;    /* MPP processing elements */
            unsigned        ac_mppbb:8;     /* MPP barrier bits         */
            comp_t          ac_mpptime:21;  /* MPP time (in clocks)     */
    };
```

The following multitasking accounting record substructure is shared by all Cray Research systems, and the record structures for specified systems are given.

```
/*
 *      Multitasking accounting record substructure.
 */
struct  mu {
        uint              :1;
        comp_t          m0:16;
        comp_t          m1:16;
        comp_t          m2:16;
        comp_t          m3:16;
};

struct  acctmu {
        struct achead   ac_header;          /* header                        */
        unsigned        ac_flag:8;          /* accounting flag               */
        long            ac_smwtime;         /* semaphore wait time (clocks) */
        struct mu       ac_mutime[MUSIZE];  /* time (compressed) connected  */
                                            /* to (i+1) CPUs (1/100 sec)    */
};
```

The rest of the `acct` file applies to all Cray Research systems, except as specified:

```
/*
 *     Error accounting record.
 */
struct accter {
       struct achead  ac_header       /* header                          */
       unsigned       ac_flag:8;      /* accounting flag                 */
       short          ac_errno;       /* u_error returned from writei()  */
       struct acerror ac_error;       /* error info from writei()        */
};

/*
 *     Performance accounting record.
 */
struct acctperf {
     struct achead  ac_header;         /* header                          */
     unsigned       ac_flag:8;         /* accounting flag                 */
     comp_t         acp_rtime:21;      /* process start time (in clocks)  */
                                       /* past ac_btime                   */
     comp_t         acp_tiowtime:21;   /* terminal I/O wait time          */
     comp_t         acp_srunwtime:21;  /* SRUN wait time (in seconds)     */
     comp_t         acp_swapclocks:21; /* swapped time (in clocks)        */
     long           acp_rwblks:21;     /* # of bufrd physical blks moved  */
     long           acp_phrwblks:21;   /* # of raw physical blks moved    */
};

/*
 *     Unified accounting record.
 */
union acct {
       struct acctbs    acctbs;
       struct acctio    acctio;
       struct acctmu    acctmu;
       struct accter    accter;
       struct acctsds   acctsds;
       struct acctmpp   acctmpp;
       struct acctperf  acctperf;
       struct accteoj   accteoj;
};

#ifdef  KERNEL
extern struct acctind acctp[];          /* inode of accting files   */
#endif                                  /* KERNEL                   */

/*
 *      Maximum number of acct records per process.
 *
```

```
 *      1 Base record + 1 Multitasking record + 1 SDS record + 1 MPP record +
 *      1 performance record + _MAXDEVIOREC device.
 *      Note the end of job record is not added since it is always singular.
 */
#define _MAXDEVIOREC ((MAXBDEVNO + MAXCDEVNO + NODEVACCT - 1)/NODEVACCT)
#define NOACCTREC       (1+1+1+1+1+_MAXDEVIOREC)

/*
 *      Flag definitions, for ac_flag.
 */
#define AFORK         01                /* has executed fork         */
                                        /* but no exec               */
#define ASU           02                /* used super-user privileges*/
#define AMORE         04                /* more accounting records   */
                                        /* follow for this process   */
#define ACCTR         0370              /* record type               */
#define ACCTBASE      0000              /* base-level acctg records  */
#define ACCTIO        0010              /* device-specific I/O       */
                                        /* accounting record         */
#define ACCTMU        0020              /* multitasking acctg record */
#define ACCTERR       0030              /* error accounting record   */
#define ACCTSDS       0040              /* SDS accounting record     */
#define ACCTEOJ       0050              /* EOJ accounting record     */
#define ACCTPERF      0060              /* performance accting rcrd  */
#define ACCTMPP       0070              /* MPP accounting record     */

/*
 *      Function types for devacct system call.
 */
#define ACCT_ON       1                 /* Device accounting on      */
#define ACCT_OFF      2                 /* Device accounting off     */
#define ACCT_LABEL    3                 /* Label block special       */
                                        /* device                    */
#define PERF_01       1                 /* Additional performance    */
                                        /* accounting on             */

#ifdef  KERNEL
/*
 *     Accounting file vnode pointers.
 */
struct acctind {
       int            did;              /* daemon identifier         */
       struct vnode   *vno;             /* acct file vnode pointer   */
};
#endif                                  /* KERNEL                    */
#include <sys/cdefs.h>

__BEGIN_DECLS
```

```
extern int devacct __((char *_Device, int _Func, int _Type));
__END_DECLS

#endif                                  /* KERNEL                    */
```

FILES

`/usr/include/sys/acct.h`	Structure of per-process accounting files
`/usr/src/cmd/acct/include/cacct.h`	Structure of condensed accounting files
`/usr/src/cmd/acct/include/session.h`	Structure of session record files
`/usr/src/cmd/acct/include/tacct.h`	Structure of per-process total accounting files

SEE ALSO

`acctcom`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`acct`(2), `devacct`(2), `exec`(2), `fork`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`acct`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

acid – Format of the account ID information file

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/acid file contains the following information for each account:

- Account name
- Account ID

The acid file is an ASCII file, which resides in the /etc directory. The fields are separated by colons; each account record is separated from the next by a newline character. udbgen(8) maintains the acid file automatically to match the information in the udb file.

The acid file maps numeric account IDs (called *ACID*s in the UDB) to account names. The account names belong to the accounting subsystem and are not user names.

NOTES

Unlike the /etc/passwd file, you must update the /etc/acid file manually to include new account IDs and account names. When you update /etc/acid, ensure that the udbgen(8) utility is not running, because udbgen would overwrite any changes to /etc/acid.

FILES

/etc/acid Format of account ID information file

SEE ALSO

acct(5), group(5), passwd(5), udb(5)

udbsee(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

udbgen(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`acl` – User access control lists format

SYNOPSIS

```
#include <sys/acl.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

An access control list (ACL) is a mechanism for user (discretionary) file access control. An ACL contains entries that define the allowed access to a file on a specific user and/or group basis.

To create and maintain an ACL file, use the `spacl`(1) command. You can use the `spset -a` command to assign an existing ACL file to a file or list of files. The `spclr -a` command removes an ACL from a file or list of files.

An ACL file consists of multiple entries, one entry per user/group name pair. Each entry has the following format:

user`:`*group*`:`*permissions*

user User name; to represent all users, use an `*` symbol.

group Group name; to represent all groups, use an `*` symbol; to represent the owning group, use `( )`.

permissions Permissions for access. Permissions are specified as follows:

- `r` Grants read permission
- `w` Grants write permission
- `x` Grants execute permission
- `n` Denies access

You can specify any combination of `r`, `w`, and `x`, or `n`.

You can specify `()` for the group only when the specified user is `*`. You cannot specify `*` for both user and group.

The format of an ACL file is defined in the `sys/acl.h` include file, as follows:

```
struct  acl  {
        uint    ac_usid :24,    /* user ID */
                ac_grid :24,    /* group ID */
                ac_flag :4,     /* ACL entry type */
                ac_mode :4;     /* access mode - r/w/x */
                ac_sort :2,     /* sort flag */
                ac_same :6;     /* same uid count */
};

struct  acl_rec  {
        long    ac_magic;
        uint    ac_size   :24,
                ac_owner  :24,
                ac_type   :8,
                ac_fill   :8;
        uint    ac_links  :24,
                ac_gmode  :3,
                ac_vsn    :6,
                ac_resrv  :31;
        struct acl acl[ACLSIZE];
};

#define ACLMAGIC     0xac0ff12ee21ff0ca
/*
 *      ACL entry types
*/
#define FLAG_UIDGID  01         /* uid.gid acl entry */
#define FLAG_GIDONLY 02         /* gid only acl entry */
#define FLAG_UIDONLY 04         /* uid only acl entry */
#define FLAG_OWNGID  010        /* owning group ACL entry */
```

NOTES

The file's group permission bits are used as a mask, which is intersected with the ACL entry permissions to determine the allowed access. This means that the group permission bits of the file always show the maximum amount of access allowed any user and/or group specified in the ACL. You must specify the permissions in both the mask and ACL entry to be allowed. For example, if the file's permission bits are set to 750 (that is, the group bits are set to r-x) and a user's ACL entry is set for read and write access only (rw-), the user is allowed only read access to that file. The user is not allowed write or execute access because both entries did not specify these permissions.

For a complete description of the masking operation and the order that ACL entries are checked, see the Security section in the *General UNICOS System Administration*, Cray Research publication SG–2301.

For a description of ACL creation and maintenance operations, see the `spacl`(1) command.

EXAMPLES

Example 1: The following ACL entry defines read, write, and execute permission to user `fred`, in any group:

```
fred:*:rwx
```

Example 2: The following ACL entry defines user `betty` read and write permission when she is in group `admin`:

```
betty:admin:rw
```

Example 3: The following ACL entry denies user `ralph` any permissions, in any group:

```
ralph:*:n
```

Example 4: The following ACL entry defines read access for owning group:

```
*::r:
```

The ACL mask is intersected with the ACL entry to determine the type of access granted.

FILES

`/usr/include/sys/acl.h` Format of user access control lists

SEE ALSO

`slog`(4), `slrec`(5)

`chmod`(1), `cpio`(1), `spacl`(1), `spclr`(1), `spset`(1), `tar`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

General UNICOS System Administration, Cray Research publication SG–2301

NAME

aft – ASCII flaw table

IMPLEMENTATION

Cray PVP systems with an IOS model E

DESCRIPTION

The files in /etc/aft contain information about physical disk defects. One aft file represents each physical device. The aft files are used by the bb command, which translates physical disk addresses into logical relative block addresses.

The files in /etc/aft are named for the I/O paths of the physical devices they represent. They are created by the ift(8) command.

The aft files may be edited to add, delete, or change entries. They can then be used to initialize the physical device spare sector maps by using the spmap(8) command.

EXAMPLES

A typical aft file follows:

```
*
*  engineering flaw table for DD-49
*
*  factory flaw map date:  09-08-89
*
*       S/N     7009

#       0 0 0 0

*       count   head    sector  cylinder

        1       7       43      1307
        1       1       47      1547
        1       1       50      1547
        1       1       50      1557
```

To initialize an aft file, enter the following command line:

```
ift /dev/ift/0130.1 >/etc/ift/0130.1
```

FILES

`/etc/aft/*`

SEE ALSO

`bb`(8), `ift`(8), `spmap`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

aliases – Defines alias database for sendmail(8)

SYNOPSIS

/usr/lib/aliases

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /usr/lib/aliases file defines the alias database for sendmail(8). The format of this file consists of the following:

alias_name: *recipient_1*, *recipient_2*, *recipient_3*, ...

alias_name is the name to alias, and *recipient_n* is the alias for that name. Lines beginning with white space (spaces or tabs) are continuation lines.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

After aliasing has been done, local and valid recipients who have a .forward file in their home directory will have their messages forwarded to the list of users defined in that file.

This is only the raw data file; the actual aliasing information is placed into a binary format in the /usr/lib/aliases.pag file, using the program newaliases(1). A newaliases command must be executed each time the aliases file has been changed before the changes will take effect.

NOTES

Blank lines and lines that begin with a # are comments.

The file must contain an alias for Postmaster and MAILER_DAEMON.

EXAMPLES

The following is an example of entries in an /usr/lib/aliases file:

```
Postmaster: root
MAILER-DAEMON: postmaster
```

FILES

`/usr/lib/aliases` File that contains the alias database for `sendmail`(8)

SEE ALSO

`newaliases`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011
`dbm`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080
`sendmail`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

a.out – Loader output file

SYNOPSIS

```
#include <a.out.h>
```

IMPLEMENTATION

All Cray Research systems except Cray MPP systems.

DESCRIPTION

The a.out file is the output file generated when the ld(1) or segldr(1) loader command is executed. If all errors occurring during the load process were at the caution level or lower, both commands make file a.out executable.

When the UNICOS operating system executes an a.out file that does not use shared text, it loads the file, as follows:

1. Reads a_text words into common memory at location a_origin (usually 0); see the header format that follows.
2. Reads the following a_data words of initialized data in at location a_origin + a_text.
3. Fills a_bss words at location a_origin + a_text + a_data with 0's.
4. Begins execution at parcel address a_entry.

When the UNICOS operating system executes an a.out file that uses shared text, it loads the file, as follows:

1. Reads a_text words into the instruction address space at location a_origin (usually 0); see the header format that follows.
2. Reads a_data words of initialized data into memory at address 0 of the data address space.
3. Fills a_bss words at location a_origin + a_data in the data space with 0's.
4. Begins execution at parcel address a_entry in the instruction space.

The following shows the header format:

```
struct exec      {
   union  {
     long   omagic;        /* old magic number                    */
     struct {
       unsigned     :32;   /* new, reserved - must be zero        */
       unsigned st  : 1;   /* new, shared text indicator          */
       unsigned     : 7;   /* new, reserved - must be zero        */
       unsigned pmt : 8;   /* new, primary machine type           */
       unsigned id  :16;   /* magic identifier                    */
     } nmagic;
   } u_mag;
   long    a_text;         /* size of text area in words          */
   long    a_data;         /* size of data area in words          */
   long    a_bss;          /* size of bss area in words           */
   long    a_syms;         /* size of symbol table in words       */
   long    a_entry;        /* entry point (parcel address)        */
   long    a_origin;       /* old base address (usually zero)     */
   union {
     long  ofill1;         /* flag, 1 = relocation info stripped  */
     struct {
       unsigned ptr  :32;  /* new, byte offset of _infoblk        */
       unsigned      :31;  /* new, reserved - must be zero        */
       unsigned str  : 1;  /* new, stripped bit                   */
     } info;
   } u_fill1;
};

/* defines for compatibility */
#define   a_magic         u_mag.nmagic.id
#define   a_omagic        u_mag.omagic
#define   a_fill1         u_fill1.info.str

/* defines for new fields */
#define   a_id            u_mag.nmagic.id
#define   a_st            u_mag.nmagic.st
#define   a_pmt           u_mag.nmagic.pmt
#define   a_infoptr       u_fill1.info.ptr
#define   a_str           u_fill1.info.str

#define   A_MAGIC1        0407  /*  normal magic                     */
#define   A_MAGIC2        0410  /*  shared text                      */
#define   A_MAGIC3        0411  /*  normal ymp-32 bit magic          */
```

```
#define   A_MAGIC4        0412  /*  shared text ymp-32 bit magic    */

#define   A_MAGIC_ID      0407  /* new magic id                     */
/* --- primary machine types ---*/
#define   A_PMT_UNDF      0     /* undefined machine type =>old hdr */
#define   A_PMT_INC       1     /* incremental load code fragment   */
#define   A_PMT_CRAY1     2     /* CRAY-1S                          */
#define   A_PMT_XMP_NOEMA 3     /* CRAY-X/MP, 22-bit mode           */
#define   A_PMT_XMP_ANY   4     /* CRAY-X/MP, mode indifferent      */
#define   A_PMT_XMP_EMA   5     /* CRAY-X/MP, 24-bit mode           */
#define   A_PMT_CRAY2     6     /* CRAY-2                           */
#define   A_PMT_YMP       7     /* CRAY-Y/MP                        */
#define   A_PMT_C90       8     /* CRAY C90                         */
```

NOTES

The UNICOS object file format is unique. AT&T common object files are not supported.

FILES

`/usr/include/a.out.h`	Default, executable, output file header format, which the `ld`(1) and `segldr`(1) commands produce

SEE ALSO

`mpp.a.out`(5) for the description of the Cray MPP loader `a.out` file
`relo`(5) for information about the relocatable object table format under the UNICOS operating system

`ld`(1) to invoke the link editor with traditional UNIX invocation
`segldr`(1) to invoke the Cray segment loader (SEGLDR)
in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

Segment Loader (SEGLDR) and ld Reference Manual, Cray Research publication SR–0066

NAME

ar – Archive file format

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This entry describes the format of an archive file. The ar(1) archive command combines several files into one. You can use archives as libraries through which the link editors ld(1) and segldr(1) search; however, the bld(1) utility is recommended for this purpose.

Each archive begins with the following archive magic strings:

```
#define  ARMAG  "!<arch>\n"   /* magic string                    */
#define  SARMAG 8             /* length of magic string          */
```

The individual files, which are called *archive file members*, follow the archive magic string. Each file member is preceded by a file member header, which has the following format:

```
#define  ARFMAG  "`\n"        /* header trailer string           */

struct ar_hdr                 /* file member header              */
{
         char  ar_name[16];   /* '/' terminated file member name  */
         char  ar_date[12];   /* file member date                 */
         char  ar_uid[6];     /* file member user identification  */
         char  ar_gid[6];     /* file member group identification */
         char  ar_mode[8];    /* file member mode (octal)         */
         char  ar_size[10];   /* file member size                 */
         char  ar_fmag[2];    /* header trailer string            */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for ar_mode, which is in octal). Thus, if the archive contains printable files, you can print the archive itself.

The ar_name field is blank-filled and slash (/) terminated. The ar_date field contains the modification date of the file at the time of its insertion into the archive. If you use the ar(1) portable archive command, you can move common format archives from system to system.

Only the name field has any provision for overflow. If any file name consists of more than 14 characters or contains an embedded <space>, the string "#1/" followed by the ASCII length of the name is written in the name field. The file size (stored in the archive header) is incremented by the length of the name. The name is then written immediately following the archive header.

Each archive file member begins on an even-byte boundary; if necessary, <newline> characters are inserted between files. If the file name is less than or equal to 14 characters and does not contain an embedded <space>, the size specified (ar_size) reflects the actual size of the file, exclusive of padding. Otherwise, the size specified reflects the actual size of the file, plus the number of characters in the file name.

No provision exists for empty areas in an archive file.

FILES

/usr/include/ar.h	Format of archive files

SEE ALSO

a.out(5) for loader output file information

ar(1) which is the archive and library maintainer for portable archives
bld(1) to maintain relocatable libraries
segldr(1) to invoke the Cray Research segment loader (SEGLDR)
in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

`arrayd.conf`, `arrayd.auth` – Array services configuration files

SYNOPSIS

`/usr/lib/array/arrayd.conf`

`/usr/lib/array/arrayd.auth`

IMPLEMENTATION

IRIX and UNICOS systems

DESCRIPTION

The `arrayd.conf` and `arrayd.auth` files describe the configuration of one or more arrays. The default configuration files are `/usr/lib/array/arrayd.conf` and `/usr/lib/array/arrayd.auth`, although the system administrator can override this or specify additional files. Every machine running an array services daemon (which should be every machine that is part of an array) must have its own configuration file or files. The configuration files contain information about which arrays are known to the array services daemon and the machines in each of them, the commands that can be executed by the array services daemon, various local options, and information used for authenticating messages passed between array services daemons on different machines.

The `arrayd.conf` file is typically readable by all users, while the `arrayd.auth` file is generally readable only by root. Other than their initial access permissions upon installation, there is no functional difference between the two files; either may contain any sort of configuration information. However, because `arrayd.auth` is not readable by most users, it is most appropriate for secure information such as authentication keys, while `arrayd.conf` is intended to contain public information such as the array and command definitions.

The initial configuration files that are installed with array services are very minimal. These files describe a single array, made up only of the local machine, and no authentication. Every site installing array services will need to customize the configuration file to describe its local arrangement.

General Syntax

The configuration file itself is made up of regular, human-readable ASCII text. Blank lines and comments (introduced by a # character) are ignored. There are four types of entries in the configuration file: array definitions, command definitions, local options, and authentication information. A typical entry may consist of several subentries; by convention, each should be on a separate line. Similarly, some subentries may have options, which should be on separate lines as well. Leading white space is ignored, so subentries and options can (and should) be indented for improved readability. The entries in a configuration file and the subentries within an individual entry need not be in any particular order.

Arguments

Most of the various entries, subentries, and options take arguments. `arrayd.conf` accepts the following arguments:

names These are simple identifiers, similar to variable names. They can contain upper- and lower-case letters, the characters `-` and `_`, and numeric digits (although the first character must not be a digit).

numbers These are treated as signed 64-bit integers and may be specified in hexadecimal, octal, or decimal, with hexadecimal values being preceded by `0x` and octal values being preceded by `0`.

environment variables
A name preceded by a `$` is presumed to refer to an environment variable and will be substituted accordingly.

strings Any arbitrary string of characters enclosed in double quotes. Double quotes and backslashes can be embedded within the string by preceding them with a backslash. Newlines and tabs can be included using `"\n"` and `"\t"`, respectively. A real newline may also be embedded by preceding it with a backslash, thus allowing a string to span several lines in a configuration file.

substitution variables
A name preceded by a `%` is referred to as a *substitution variable* and will be replaced with some other value. Recognized substitution variables include the following:

`%1, %2, ..., %9`
These represent the first nine arguments specified for an `array` command. For example, if a user invokes an `array` command with `array killjob 1354 token`, then `%1` would be replaced with `1354` (the first argument to the command `killjob`), and `%2` would be replaced with `token`. Arguments that do not exist (`%3` in this case) are replaced with an empty string.

`%ALLARGS`
This is replaced with all of the arguments that were specified for an `array` command. When used with subentries that take multiple arguments, each individual command-line argument is treated as an individual argument in the subentry as well. When used with subentries that take only a single argument, only the first command-line argument is actually substituted.

`%ARRAY`
This is replaced with the name of the array that is the target of the current `array` command. This is primarily of use when a machine belongs to two or more separate arrays.

`%ASH` This is replaced with the array session handle of the program that invoked the current `array` command. It is in hexadecimal and is preceded by the string `0x`.

`%GROUP`
This is replaced with the name corresponding to the effective group ID of the process that invoked the current `array` command.

%LOCAL
: This is replaced with the name of the local machine, as specified in a `LOCAL HOSTNAME` entry. This is useful if several machines share a configuration file containing commands.

%ORIGIN
: This is replaced with the primary hostname of the network interface that transmitted the request from the client machine. If the client and server are the same machine, then this is `localhost`. This is often not the same as the client's machine name, as it typically includes the network name as well (for example, `machine.domain.com`, not just `machine`).

%OUTFILE
: This variable is valid only as part of a merge command. It is replaced with a list of one or more temporary files. Each file contains the output from a single machine of the related `array` command. When the merge command is finished, the temporary files are automatically removed. The files in the list are not in any particular order; if the merge command needs to know which machine a specific file came from, the original `array` command should include that data in its output. When used with subentries that take multiple arguments, each individual pathname is treated as an individual argument in the subentry as well. When used with subentries that take only a single argument, only the first output file pathname is actually substituted.

%REALGROUP
: If the process that invoked the current `array` command has different real and effective group IDs, then this is replaced with the name corresponding to the real group ID. If the real and effective group IDs are the same, then `<same>` is substituted instead.

%REALUSER
: If the process that invoked the current `array` command has different real and effective user IDs, then this is replaced with the name corresponding to the real user ID. If the real and effective user IDs are the same, then `<same>` is substituted instead.

%USER This is replaced with the name corresponding to the effective user ID of the process that invoked the current `array` command.

Note that the names of these substitution variables may be in either upper- or lower-case. If an unrecognized variable name is specified, a warning is issued, and the variable is replaced with an empty string.

substitution functions
: A substitution variable followed immediately by one or more arguments enclosed in parentheses is a *substitution function*. An argument to a substitution function can generally be anything that is valid as the argument to an entry or subentry, except for another substitution function. Recognized substitution functions include

%ARG(*number*)
This is replaced with the command argument specified by *number*, which should be a numeric value. If the argument does not exist, a warning is generated, and an empty string is substituted.

%OPTARG(*number*)
This is similar to %ARG(...), except that no warning is generated if the specified argument does not exist. This is useful for specifying optional arguments.

%PID(*ash*)
ash specifies an array session handle. This is replaced with a list of all process IDs (PIDs) that belong to the specified array session on the local machine. For entries that take more than one argument, each PID is treated as a separate argument (see %ALLARGS).

As with *substitution variables*, an unrecognized substitution function is replaced with an empty string and causes a warning to be generated.

literal arguments
A *literal argument* is any argument that can be evaluated when the array services daemon is first started. This includes names, strings, numbers, and environment variables, but specifically does not include substitution variables or functions.

numeric arguments
A *numeric argument* is an argument that can be resolved to a numeric value when the array services daemon is first started. This includes actual numbers, as well as strings and environment variables. An error occurs if a string or environment variable cannot be converted to proper numeric values.

Array Entries

An array entry is a configuration file entry that defines the machines and other details that make up a particular array. The general format is as follows:

```
ARRAY array-name
ARRAY_ATTRIBUTE name=value
ARRAY_ATTRIBUTE litarg...
IDENT number
SEQFILE pathname
MACHINE machine-name-1
machine options
                                    ...
MACHINE machine-name-2
                          ...
```

Keywords such as ARRAY, MACHINE, and IDENT may be in either upper- or lower-case; upper-case is used here to distinguish them from other fields. The various subentries do not necessarily have to occur in any particular order. However, they should not appear between options in a MACHINE subentry.

array-name is the name that will be used to refer to the array as a whole; it may be of any length. This is the name that would be used with the -a option of the array(1) command.

The ARRAY_ATTRIBUTE subentry is used to specify one or more arbitrary values that will be maintained in the configuration database, but will otherwise be ignored by the array services daemon. Programs that obtain array configuration information (for example, using the aslistarrays(3X) function) will be provided with a list of these attributes. Thus, these could be useful for maintaining miscellaneous configuration information that may be needed by other programs. The ARRAY_ATTRIBUTE subentry may be specified more than once. If the attribute starts with a simple identifier followed by an equal sign, then the remainder of the line (with multiple blanks and tabs converted to a single space) is appended to form a single attribute. Such an attribute could be used along with the asgetattr(3X) function in a manner similar to environment variables. If the attribute is formed of any other literal argument, it is presumed to end as soon as white space is encountered. In this case, multiple attributes could be specified on a single line.

The SEQFILE subentry specifies the pathname of a file used to keep an array session sequence number for the array. The default sequence file is located in the directory specified by LOCAL DIR (see below) and has a name formed by appending the array name to the string .seqfile..

The IDENT subentry specifies a numeric value that is used when generating global array session handles for the array. No other array should have the same IDENT value. If an IDENT value is not specified, a random one will be generated. The value should be in the range of 1 to 32767.

Each MACHINE subentry specifies a single machine that is a member of the array. Each ARRAY entry must have at least one MACHINE subentry. *machine-name* is the name that is used to refer to this machine. Ordinarily this would be the machine's host name; however, that is merely a convention and not a requirement. A MACHINE subentry may have zero or more options. These include

MACHINE_ATTRIBUTE *litarg...* or *name=value*
: The MACHINE_ATTRIBUTE option is similar to the ARRAY_ATTRIBUTE subentry in that it is used to specify one or more arbitrary values that are maintained in the configuration database, but otherwise are ignored by the array services daemon. Programs that obtain machine configuration information (for example, using the aslistmachines(3X) function) are provided with a list of these attributes. Thus, these are useful for maintaining miscellaneous configuration information that may be needed by other programs. The MACHINE_ATTRIBUTE option may be specified more than once, and it has the same syntax as ARRAY_ATTRIBUTE.

[SERVER] HOSTNAME "*string*"
: This specifies the full host name or IP address of the machine. The value should be enclosed in double quotation marks. If a HOSTNAME is not specified, the machine name will be used. The string SERVER is optional.

SERVER IDENT *number*

This specifies the numeric identifier of the array services daemon on the specified machine. This value may be used for generating global array session handles or uniquely identifying the machine. If a SERVER IDENT is specified for a machine, it should match the LOCAL IDENT that is specified in that machine's local array services configuration file. Unlike the syntax for the HOSTNAME and PORT options, the string SERVER that comes before IDENT is required.

[SERVER] PORT *number*

This specifies the port on which the array services daemon for this machine is listening. This would override the default port number of 5434. The string SERVER is optional.

Command Definitions

A command entry defines the actual program that is invoked by the array services daemon when it receives an array command. Its format is similar to that for an array entry:

```
COMMAND cmd-name
INVOKE any-args...
MERGE any-args...
GROUP any-arg
USER any-arg
OPTIONS litarg...
```

cmd-name specifies the actual command name. This is what the user would use when invoking the command with array(1).

The INVOKE subentry specifies the actual program to be executed, plus any arguments that should be supplied to it. Any number of arguments may be specified for the INVOKE subentry. Groups of arguments that are not separated by white space are concatenated to form single values (white space embedded in a string is not considered to be white space for these purposes). Each resulting value is passed to the program to be executed as a single argument. Thus, if a user typed array foo a b c, and the INVOKE subentry for the command foo were as follows:

```
INVOKE /usr/bin/test%1 %2"this is a test" %3
```

The argument list for the program to be executed would consist of the following:

```
argv[0] = "/usr/bin/testa"
argv[1] = "bthis is a test"
argv[2] = "c"
```

The first value in the argument list also specifies the actual pathname of the program to be executed (/usr/bin/testa in this case). The array services daemon does not have a search path, so this must specify either an absolute path to the file to be executed, or a path relative to the array services daemon's current directory (see the DIR local option).

The MERGE subentry is used to specify a merge command. Ordinarily, when an array command is run on several machines, the results and output from each machine are returned as separate streams of data. However, if a merge command is specified, it is run after the array command itself has been completed on all machines, and only the results and output of the merge command are returned. When used with the %OUTFILES substitution variable, this could be a convenient way to consolidate or summarize the results of the array command. The MERGE command is executed in the same way as a normal INVOKE command, except that it always runs on the same machine as the array services daemon, even if that particular machine is not a member of the array on which the array command was run.

The GROUP and USER subentries are optional and specify the name of the group and user under which the program should be run. Each of these take a single argument. To run with the IDs of the user who invoked the array command, these could be specified as %GROUP and %USER, respectively. If these are not specified for a particular command entry, they default first to the values set in the local options, or, if those are not present, to user and group guest. By default, the GROUP and USER subentries affect only the effective group and user IDs of the program; the real group and user IDs will be the same as those of the process that invoked the program. This behavior can be changed by using the SETRGID and SETRUID command options (see below).

The OPTIONS subentry is used to specify additional details about how the command should be processed. It should be followed by one or more arguments from the following list. The arguments may be in either upper- or lower-case. They may also be preceded by the string NO to negate their effects.

LOCAL — Executes the command on the same machine as the array services daemon only, even if a target array was specified explicitly or by default.

NEWSESSION — Executes the command in a new global array session. Normally the command would be run in the same array session as the process that invoked it.

QUIET — Discards any output generated by the command. If a merge command has been specified, QUIET applies to the merge command and not the invoke command. This would allow a merge command to quietly act on the output of the invoke commands.

SETRGID — Runs the command with both its real and effective group IDs set to the value specified by the GROUP subentry. Normally, only the effective group ID is taken from the GROUP subentry, while the real group ID is taken from the process that invoked the command.

SETRUID — Runs the command with both its real and effective user IDs set to the value specified by the USER subentry. Normally, only the effective user ID is taken from the USER subentry, while the real user ID is taken from the process that invoked the command.

WAIT — Waits for each invoked program to complete execution before returning control to the process that requested the command. This is the default behavior. If NOWAIT is specified, control is returned to the requester immediately after the invoked programs are started. NOWAIT implies QUIET and causes any merge command to be ignored.

Local Options

A local options entry specifies options to be used by the array services daemon itself. If more than one local options entry is specified, settings in later entries silently override those in earlier entries. A local options entry looks like this:

LOCAL

DIR *literal-arg* DESTINATION ARRAY *literal-arg* GROUP *literal-arg*
HOSTNAME *literal-args* IDENT *num-arg* PORT *num-arg*
USER *literal-arg* OPTIONS *literal-arg*...

All of the subentries in a local entry are optional.

DIR Specifies an absolute pathname for the array services daemon's working directory. The default is /usr/lib/array.

DESTINATION ARRAY
Specifies the default target array for array commands when one has not been specified explicitly by the user. There is no default value unless only one array is defined (in which case it becomes the default); if a user omits the target array and there is no default, an error occurs.

GROUP and USER
Specify the names of the group and user under which an array command should be run. A GROUP or USER specified in a particular command entry always overrides these values. These subentries default to the group and user that is running the array services daemon.

HOSTNAME
Specifies the value that is returned by the %LOCAL substitution variable. The results of array services commands initiated with ascommand(3X) also refer to this name. The default is the actual host name of the local machine.

IDENT Specifies a numeric value that is included in global array session handles generated by this array services daemon. Some versions of UNICOS may also make use of this value to generate their own global array session handles. No other array services daemon should have the same IDENT value. If an IDENT value is not specified, one is generated from the hostid of the local machine. The value must be in the range of 1 to 32767.

PORT Specifies the network port on which this array services daemon listens for requests. The default is the standard sgi-arrayd service, 5434.

OPTIONS
Specifies additional details about the operation of the array services daemon. It should be followed by one or more arguments from the following list. The arguments may be in either upper- or lower-case. They may also be preceded by the string NO to negate their effects.

CHKLOCALID Instructs arrayd to make certain authentication checks when accepting a connection from a local user, such as ensuring that the user is formally authorized for their current group. Note that these checks may fail on systems that have mechanisms for changing the real group of a user to a setting that is not in one of the standard administrative files (for example, /etc/group or its corresponding network information service (NIS) map).

SETMACHID Some versions of UNICOS permit setting a system machine identifier, which is used by the kernel for generating global array session handles. If the current system has this facility, and SETMACHID is specified, arrayd sets the machine ID to the value specified by a LOCAL IDENT statement in the configuration file or on the command line with the -m option.

SVR4SIGS Instructs arrayd to use SVR4 semantics for the SIGXCPU and SIGXFSZ signals when starting a new process to handle a remote execution request (such as those issued by the Array Services arshell(1) command). In this mode, the new process ignores SIGXCPU and SIGXFSZ signals unless it specifically alters the behavior for those signals with a system call such as signal(2) or sigset(2). This is different from the default behavior for processes started by arrayd to handle remote execution requests, in which SIGXCPU and SIGXFSZ will cause the process to abort with a core dump. (This feature requires the Array Services 3.1 for IRIX release or later.)

Authentication Information

An authentication information entry is used to describe the type of authentication that should be done when passing messages to and from another array services daemon. Authentication information entries do not accumulate: if more than one is encountered in the various configuration files processed by an array services daemon, only the last one has any effect; all information from previous entries is discarded. There is currently only one type of authentication provided, although more may be provided in the future. Its entry is as follows:

AUTHENTICATION SIMPLE

HOSTNAME *literal-arg* KEY *num-arg* HOSTNAME *literal-arg* KEY *num-arg* ...

This entry contains one or more subentries consisting of machine/key pairs. *literal-arg* is the network host name of a machine. Notice that the network host name is not necessarily the same as the machine name used to identify a machine in an array entry (see above). *num-arg* is a 64-bit unsigned integer that is to be used as the authentication key for all messages originating from that machine. If a key of 0 is specified, authentication is not performed on messages originating from that machine. Similarly, if a machine has no subentry at all, no authentication is performed on messages received from it.

If a machine appears in more than one array entry, it needs to have only one subentry in the authentication information. Conversely, the machine in an authentication information subentry does not need to appear in any array entries.

With the SIMPLE scheme, a *digital signature* is calculated for each message by using the authentication key associated with the sending machine, and this value is then sent along with the message. When an array services daemon receives a message from another machine, it checks its private database for the authentication key associated with the machine that sent the message, recalculates the digital signature, and ensures that it matches the one sent with the message. This provides some basic protection against forged messages because a forger (presumably) would not have access to the authentication key that is required to calculate a proper digital signature.

Because this approach depends on the secrecy of the authentication keys, it is important to put this type of authentication information entry in a configuration file that is not accessible to general users (for example, the arrayd.auth file in the default installation). Because both the sender and receiver need to have the same authentication key for a given machine, the administrator must take special care to ensure that the authentication information in each machine's configuration files is consistent with that in the corresponding file.

There are some circumstances in which array services may be needed on an array of only one machine (for example, systems that use the MPI message passing library). For these systems, an alternative to using simple authentication is to simply disallow any requests from remote systems. This can be done by specifying an authentication information entry of the form

```
AUTHENTICATION NOREMOTE
```

For the purposes of array services, any request to an IP address other than 127.0.0.1 is considered to be remote. Therefore, the HOSTNAME entry for the local machine in any array should be either 127.0.0.1 or localhost if NOREMOTE is being used. While this blocks any incoming array services requests from remote machines, it does not prevent outgoing array services requests originating on the local machine from being sent to remote machines.

If an array is on a private network with trusted peers, or perhaps is carefully hidden behind a good firewall, authentication may be unnecessary. It is possible to disable authentication entirely by using an authentication information entry of the form

```
AUTHENTICATION NONE
```

This is the default setting when the array services are first installed. However, unless the environment is reasonably secure, this should be changed to one of the other authentication settings as soon as possible.

WARNINGS

The UNICOS operating system is dependent on the nobody user being configured in order to use array services and the message passing interface (MPI).

SEE ALSO

arrayd(8) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

arshell(1)

array_services(7), array_sessions(7)

NAME

bld – Relocatable library files format

IMPLEMENTATION

Cray PVP systems only

DESCRIPTION

When the bld(1) command collects relocatable modules, it creates a relocatable library file called a *build file* or a bld *file*. The *build file* consists of a header table, a collection of relocatable modules, and a contents table (also called a *termination table*).

The header table precedes the relocatable modules; it has the following format:

```
struct  bld_hdr {
        struct  tbl_hdr  hdr;
        long    pdt_offset;    /* file offset to the build     */
                               /* termination table            */
                               /* (1 = no pdt entries)         */
        long    pdt_size;      /* size (in words) of the build */
                               /* termination table            */
                               /* (1 = no pdt entries)         */
};
```

The contents table follows the relocatable modules; it consists of a table header followed by copies of all Program Descriptor tables (PDTs) that occur in the modules within the relocatable library. (See relo(5) for descriptions of the table header and PDTs.) The ld(1) and segldr(1) loader commands use the build header table and contents table.

The bld(1) command uses the pdtscl field in each PDT in the build contents table to store a file pointer to the associated module.

FILES

/usr/src/cmd/bld/bld.h Format of relocatable library files

SEE ALSO

ar(5), relo(5)

ar(1), bld(1), segldr(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

confval – Configuration file for various products

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The confval file contains configuration information for various products in the following format:

product.*field* : *value* [*value* . . .]

product Name of the product

field Product-specific field identifier

value Text string (or list of text strings, separated by white space) that the product expects to see for the given configuration field

You must separate the *product* and *field* strings by using a period (.), and you must use a colon (:) to separate the *field* and first *value* strings. The backslash continuation character (\) is honored if it is immediately followed by a newline character (\n).

Any line that starts with a # symbol is considered a comment line and is ignored. Blank lines also are ignored.

To delimit the starting and ending locations of a value, use the " symbol; however, new lines are not allowed within a delimited value.

Options:

login.deflbl_as_minlbl
: This is a UNICOS centralized user Identification/Authentication (I/A) option for determining a user's mandatory access control (MAC) attributes. If this option is selected, the user's default label also will be used as the user's minimum label. This provides the ability to define a user's minimum compartment set. ia_mlsuser(3C) processes this option.

login.logbadpass
: This is a UNICOS centralized user Identification/Authentication (I/A) failure processing option. If this option is selected, the failed I/A attempts are logged in the syslog by ia_failure(3C). This configuration option is only for systems that have SECURE_SYS configured off. ia_failure(3C) processes this option.

CAUTIONS

If you edit this file on a running (multiuser) system, binary files may not detect the new configuration information because of the internal buffering of data performed by getconfval(3C). For best results, restart the affected binary file and/or binary files.

EXAMPLES

A partial example of a `/etc/config/confval` file follows:

```
#  Partial example for gated(8) configuration
#
gated.debug:    1
gated.rip:      quiet
gated.static:   "128.162.82.124 rip metric 1 active"

#
#  Partial example login(8) configuration, set so that:
#    1. Causes user's default label to be used as both the default and minimum
#       label for all UDB references  for user's minimum label (not just login)
#    2. Failed login attempts are not put into the system log
#    3. The user has unlimited attempts during a connection to try to log in

login.deflbl_as_minlbl:  1
login.logbadpass:        0
login.login_attempts:    0
```

SEE ALSO

`getconfval`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

core – Core file format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/user.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

A core file is the image of a terminated process; the UNICOS system writes out a core file when various errors occur. The most common errors are memory violations, illegal instructions, and user-generated quit signals; see signal(2) for a complete list of possible errors. The process image is written to a file called core in the working directory of the process if that directory is writable or if a core file already exists and is writable. If extended core file naming is turned on, the process image is written to a file named core.*pid* in the working directory of the process if that directory is writable. A process with an effective user ID different from the real user ID does not produce a core file. See setuid(2) for more information on setting user and group IDs.

The core image has two sections. The first section of the image contains the user common structure, ucomm, which is of size UCSIZE (in clicks) (on Cray Research systems, a *click* is 4096 bytes). It is described in the sys/param.h include file. The ucomm structure is followed by one or more user structures; each user structure is size ULSIZE (in clicks). USIZE is still available for compatibility.

The format of a user structure also is described in the sys/user.h include file. When the process is not multitasked, exactly one user structure exists; when it is multitasked, one user structure exists for each process (task). The number of user structures in the core file is specified by the uc_core variable in the ucomm structure. The user structures start at offset UCSIZE clicks in the core file and continue for uc_usoff clicks; each user structure has a flag in the user structure, u_active, set to a nonzero value if the user structure is in use.

The second section of the image is the user memory area. The second section of the core image is written only when the size of the process is less than the core file size limit, as defined in the pc_corelimit field in sys/proc.h. (The core file size limit for each user defaults to unlimited, but might have been reduced by the system administrator using the ue_pcorelim field in the user database (UDB) or by the user using the limit(1) command with the -d option. For information about determining the core file size limits, see the limit(1) man page.) If the attempt to write a complete restartable core file fails, an attempt is made to write a truncated core file, in which only the first section of the core image is written.

Only the data area is dumped if the instruction area is separate from the data area (this is called *split I&D* or *shared text*).

NOTES

The `crash`(8) command can write a core file.

FILES

`/usr/include/sys/param.h`	System parameter file
`/usr/include/sys/proc.h`	Format of the process common structure
`/usr/include/sys/types.h`	Data type definition file
`/usr/include/sys/user.h`	Format of the user common structure

SEE ALSO

`adb`(1), `limit`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`setuid`(2), `signal`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`crash`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

cpio – cpio archive file format

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The cpio archive file is the output from the cpio(1) command, which collects files into an archive. Each file within the archive is preceded by a header that has two possible formats. When you omit the -c option of cpio(1), the header structure is as follows:

```
struct header {
       int    h_magic,
              h_dev;
       uint   h_ino,
              h_mode,
              h_uid,
              h_gid;
       int    h_nlink,
              h_rdev;
       int    h_param[8];
       long   h_mtime;
       int    h_namesize;
       long   h_filesize;
       char   h_name[h_namesize rounded to word];
}
```

When the -c option of cpio(1) is used, the header information is described by the following:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
       &Hdr.h_magic,  &Hdr.h_dev,  &Hdr.h_ino,  &Hdr.h_mode,
       &Hdr.h_uid,  &Hdr.h_gid,  &Hdr.h_nlink,  &Hdr.h_rdev,
       &Longtime,  &Hdr.h_namesize,  &Longfile,  Hdr.h_name);
```

Longtime and Longfile are equal to Hdr.h_mtime and Hdr.h_filesize, respectively. The contents of each file immediately follow the archive header that describes the file.

If h_aclcount is nonzero, the access control list (ACL) entries immediately precede the header for the following file.

Each instance of h_magic contains the constant 070707 (octal). Items h_dev through h_mtime correspond to the items in the stat structure explained in stat(2). The length of the null-terminated path name h_name, including the null byte, is given by h_namesize.

The last record of the `cpio` archive always contains the name `TRAILER!!!`. Special files, directories, and the trailer are recorded with `h_filesize` equal to 0.

For a `cpio` archive that contains security labeling, the `-c` option is not allowed. The following header structure precedes the previously described header structure for each archived file:

```
/* Secure cpio header format */
struct sheader {
       int     h_smagic;
       int     h_slevel;
       long    h_compart;
       long    h_acldsk;
       int     h_aclcount;
       long    h_hdrvsn;
}
```

Each instance of `h_smagic` contains the constant 060606 (octal). The `h_slevel` and `h_compart` fields contain the file's security level and compartments, respectively. The `h_acldsk` field is a flag that indicates whether an ACL has been archived for this file, and `h_aclcount` holds the number of entries in that ACL.

The following secondary security header structure immediately follows the `sheader` structure:

```
/*
 *  Additional cpio secure header
 */
struct nheader {
       int     h_nmagic;
       int     h_intcls;
       long    h_intcat;
       long    h_secflg;
       int     h_minlvl;
       int     h_maxlvl;
       long    h_valcmp;
       long    h_reserved[16];
}
```

Each instance of `h_nmagic` contains the constant 050505 (octal).

If `PHdr` is in the archive, the first item in the archive is the `PHdr`. The `PHdr` contains the privilege authorization list (PAL) header. The PAL header has the following structure and a magic number of 040404:

```
struct      pheader {
                int     h_pmagic;
                pal_t   h_pal;
}PHdr;
```

SEE ALSO

`cpio`(1), `find`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

cshrc, login, logout – C shell start-up and termination files

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/cshrc, $HOME/.cshrc, and $HOME/.login files are C shell start-up files. On login, the system checks the shell field in a user's entry in the UDB file (/etc/udb) to see what shell it specifies. If you specify /bin/sh or /bin/ksh, /etc/profile then $HOME/.profile is run. For more information, see sh(1), ksh(1), and profile(5).

If you specify /bin/csh in the shell field of the password file, the following actions occur as a user logs in:

1. If the /etc/cshrc file exists, the C shell (csh(1)) executes it. Among other operations, /etc/cshrc prints /etc/motd, the message of the day, if that file exists (see motd(5)).
2. If the user's login directory contains a file named .cshrc, csh(1) executes it.
3. If the user's login directory contains a file named .login, csh(1) executes it.
4. The user's terminal session begins.

Files /etc/cshrc and .login are executed only on login, but file .cshrc is executed each time csh(1) is executed. Therefore, .login is useful for setting and exporting environment variables and for executing commands desired on login (for example, calendar(1)); .cshrc is useful for setting up aliases and other environment parameters that should be set each time csh(1) is executed.

When a login C shell terminates, the $HOME/.logout file is executed. The user or system administrator creates the .logout file, which contains commands to be executed on shell termination. For example, a .logout file might include commands to clear the screen and to erase temporary files.

EXAMPLES

Example 1: An example of a typical .login file is as follows:

```
# Set file creation mask:
umask 22
# Echo a greeting:
echo "Welcome to the Cray Research computer system"
# Establish command search path
setenv path=($PATH   $HOME/bin)
```

Example 2: An example of a typical `.cshrc` file is as follows:

```
# Check for interactive mode and set prompt and history:
if  (  $?prompt  )  then
     set prompt  =  "CRAY>  "
     set history  =  22
endif

# Set some aliases:
alias l     ls -al
alias h     history -r
```

Example 3: An example of a typical `.logout` file is as follows:

```
# Remove files in personal temporary directory
rm $HOME/tmp/*
# Clear the screen
clear
```

FILES

`$HOME/.cshrc`	C shell start-up file in user's home directory
`$HOME/.login`	C shell start-up file in user's home directory
`$HOME/.logout`	C shell termination file in user's home directory
`/bin/csh`	`csh` command
`/etc/cshrc`	Systemwide C shell start-up file
`/etc/udb`	User information file

SEE ALSO

`motd`(5), `profile`(5)

`csh`(1), `env`(1), `ksh`(1), `login`(1), `mail`(1), `printenv`(1B), `sh`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

def_seg, def_ld, ld_Flib – Loader directives files

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /lib/segdirs/def_seg, /lib/segdirs/def_ld, and /lib/segdirs/ld_Flib files are loader directives files. The default directives files contain initial information used for the loading process. This information includes machine-specific program construction data, the library names (if any) that are searched by default, and the location of the libraries to be searched.

The /lib/segdirs/def_seg file is the default directives file for the segldr(1) command; /lib/segdirs/def_ld is the default directives file for the ld(1) command. When either loader command begins execution, it reads the contents of the default directives file for that command.

When you specify the -F option on the ld(1) command line, the ld(1) command uses the /lib/segdirs/ld_Flib file. It describes the libraries ld(1) should search when flowtracing has been enabled or when a user wants to include the complete set of default libraries. This file should identify the same libraries that the def_seg file specifies for segldr.

The initial contents of these files are created when the system is installed. To customize the loader actions, the system administrator can add or remove directives in any of the files.

EXAMPLES

The following examples show the contents of the three loader directives files.

Sample def_seg file:

```
system=unicos        /* set the target operating system             */
start=$START         /* declare the name of the program entry point */
callxfer=M$A$I$N     /* declare the name of the transfer reference  */
compress=1000        /* declare the compression threshold value     */
hardref=trbk         /* force hard references to entry 'trbk'       */
deflib=libc.a        /* identify the default libraries              */
deflib=libu.a
deflib=libm.a
deflib=libf.a
deflib=libio.a
deflib=libsci.a
deflib=libp.a
```

Sample `def_ld` file:

```
system=unicos        /* set the target operating system            */
start=$START         /* declare the name of the program entry point */
callxfer=M$A$I$N     /* declare the name of the transfer reference  */
compress=1000        /* declare the compression threshold value     */
lbin=_start_.o       /* load the system start-up routine first      */
```

Sample `ld_Flib` file:

```
deflib=libc.a        /* identify the default libraries             */
deflib=libu.a
deflib=libm.a
deflib=libf.a
deflib=libio.a
deflib=libsci.a
deflib=libp.a
```

FILES

`/lib/segdirs/def_ld`	Default directives file for `ld`
`/lib/segdirs/def_seg`	Default directives file for `segldr`
`/lib/segdirs/ld_Flib`	Identifies libraries used by `ld`

SEE ALSO

`ld`(1), `segldr`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

dir, ncdir – Directory file format

SYNOPSIS

```
#include <sys/fs/ncdir.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

A directory functions as a regular file, except that no user may write to a directory. The mode field of a file's inode entry indicates whether a file is a directory.

```
struct cdirect
{
       unsigned long  cd_ino;                /* Inode for name           */
       unsigned long  cd_sino;               /* Reserved for future use  */
       unsigned short cd_reserved:10,        /* Reserved for future use  */
                      cd_signature:22,       /* Name signature           */
                      cd_reclen:22,          /* Record length (bytes)    */
                      cd_namelen:10;         /* Length of name (bytes);  */
       unsigned char  cd_name[CDMAXNAMELEN]; /* Directory name           */
} ;
```

By convention, the first two entries in each directory are "." and "..". The first is an entry for the directory itself, and the second is for the parent directory. The meaning of ".." is modified for the root directory of the master file system; because no parent directory exists, ".." has the same meaning as ".".

An unused directory entry, identified by cd_ino=cd_namelen=0, is permitted only at the beginning of a block.

Directory names are null-padded to the nearest word boundary. If the name length is a multiple of 8, a null-terminator is not guaranteed.

FILES

/usr/include/sys/dir.h	Not used; retained for compatibility.
/usr/include/sys/fs/ncdir.h	NC1FS file systems.

SEE ALSO

dirent(5), fs(5)

NAME

dirent – File system-independent directory entry format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/dirent.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Different file system types may have different directory entries. The dirent structure defines a file system-independent directory entry, which contains information common to directory entries in different file system types. The getdents(2) system call returns a set of these structures; you can access these structures by using the closedir(3C), opendir(3C), readdir(3C), rewinddir(3C), seekdir(3C), and telldir(3C) routines (see directory(3C)).

The dirent structure is defined as follows:

```
struct  dirent {
        long            d_ino;
        off_t           d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};
```

d_ino — Unique number for each file in the file system.

d_off — Offset from the beginning of the file to the end of the current entry.

d_reclen — Record length of the entry; defined as the number of bytes required between the current entry and the next one to ensure that the next entry is on a word boundary.

d_name — Beginning of the character array that gives the name of the directory entry. This name is null-terminated and has a maximum character length of MAXNAMLEN characters. This results in file system-independent directory entries being variable-length entities.

FILES

/usr/include/sys/dirent.h — File system-independent directory entry definition file

/usr/include/sys/types.h — Data type definition file

SEE ALSO

`dir`(5), `fs`(5)

`getdents`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`directory`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

dump, dumpdates – Incremental file system dump format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/fs/nc1ino.h>
#include <dumprestor.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

File system dump tapes used by the dump(8) and restore(8) commands contain the following information:

- A header record
- Two groups of bitmap records
- A group of records that describes directories
- A group of records that describes nondirectory files
- A trailing bitmap

The following symbols are defined in dumprestor.h, (the entries prefaced with TS_ are used in the c_type field to indicate the header type):

```
#define CNTREC    8
#define TS_TAPE   1
#define TS_INODE  2
#define TS_BITS   3
#define TS_ADDR   4
#define TS_END    5
#define TS_CLRI   6
#define TS_ACL    7
#define TS_PAL    8
#define DR_MAGIC  (int) 60012
#define CHECKSUM  (int) 84446
```

CNTREC Number of 4096-byte records in a physical tape block.

TS_TAPE First block of dump output.

TS_INODE File or directory follows. The c_dinode field is a copy of the disk inode; it contains bits that specify the type of the file.

TS_BITS	Bit map follows. This bitmap consists of 1 bit for each inode that was dumped. At the end of the dump output, a second TS_BITS bitmap indicates the inodes that were updated during execution of dump.
TS_ADDR	A subrecord of a file description, (see the description of c_addr).
TS_END	End-of-tape record.
TS_CLRI	Bit map follows. This bitmap contains a 0 bit for all inodes that were empty when the file system was dumped.
TS_ACL	Access control list (ACL) block follows.
TS_PAL	Privilege assignment list (PAL) block follows.
DR_MAGIC	A magic number.
CHECKSUM	Checksum for header records.

Header Record Format

The dumprestor.h include file defines the format of the header record and of the first record of each description.

```
union cu_spcl {
        char dummy[BSIZE];
        struct  c_spcl {
                int     c_type;
                time_t  c_date;
                time_t  c_ddate;
                long    c_tapea;
                long    c_inumber;
                int     c_checksum;
                struct  cdinode c_dinode;
                int     c_count;
                char    c_addr[NINDIR];
        } c_spcl;
};

#define cspcl cu_spcl.c_spcl
```

c_type	Header type.
c_date	Date of dump.
c_ddate	Date of previous incremental dump.
c_tapea	Current number of this 4096-byte record.
c_inumber	Number of inode being dumped if TS_INODE is set.
c_checksum	The value needed to make the record's checksum equal to CHECKSUM.

`c_dinode` Copy of inode as it appears in the file system; for a description of the inode format, see `fs`(5).

`c_count` Count of characters in `c_addr`.

`c_addr` Array of characters that describes the blocks of the dumped file, 1 bit per character. If the block associated with that character was not present on the file system when it was dumped, a character is 0; otherwise, the character is nonzero. If the block was not present on the file system, the block will be restored as a hole in the file. If there is not sufficient space in this record to describe all of the blocks in a file, `TS_ADDR` subrecords are scattered throughout the file, each one starting where the last one left off.

Dump History

The dump history is kept in the `/etc/dumpdates` file. The format of an entry in `/etc/dumpdates` is as follows:

name level date(*timestamp*) *volume* [`:` *volume*]

name Name of dumped file system.

level Level number of dump tape (see `dump`(8)).

date Date of incremental dump in `date`(1) format.

timestamp Date of the incremental dump in seconds since 00:00:00 GMT, January 1, 1970.

volume Volume serial number of the dump tape; if the dumped file system is contained on more than one tape, the numbers are separated by colons (`:`). If the file system was not dumped to a tape, the word `NULL` appears in this field.

To specify this field, use the `-T` option on the `dump`(8) command. The default is the first 40 characters of the VSN list.

FILES

`/etc/dumpdates`	Incremental file system dump file
`/usr/include/dumprestor.h`	File system dump tape header definition
`/usr/include/sys/inode.h`	Inode structure definition
`/usr/include/sys/types.h`	Data type definition file

SEE ALSO

`fs`(5), `types`(5)

`scanf`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`dump`(8), `restore`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

errfile – Error-log file format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/erec.h>
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

When the system detects hardware errors, an error record is generated and passed to the error-logging daemon, errdemon(8). The error-logging daemon records the error record in the error-log file for later analysis. The default error-log file is /usr/adm/errfile.

The format of an error record in an error-log file depends on the type of error encountered. Each record, however, has a header with the following format defined in the sys/erec.h include file:

```
struct errhdr {
       short  e_type;        /* record type                          */
       short  e_len;         /* bytes in record (with header)        */
       time_t e_time;        /* time of day                          */
} ;
```

The permissible record types are as follows:

```
#define E_GOTS 010          /* Start for UNICOS/TS                */
#define E_STOP 012          /* Stop                               */
#define E_TCHG 013          /* Time change                        */
#define E_SSD    0100       /* SSD error record                   */
#define E_MEM    01000      /* CRAY memory error                  */
#define E_SDIS   01001      /* Single-bit error detection disabled */
#define E_SEN    01002      /* Single-bit error detection enabled  */
#define E_IOS    01003      /* IOS error packet(s)                */
#define E_DSK    01004      /* Disk driver error report           */
#define E_D29    01005      /* DD29 error record                  */
#define E_D39    01006      /* DD39 error record                  */
#define E_D49    01007      /* DD49 error record                  */
#define E_D40    01010      /* DD40 error record                  */
#define E_D10    01011      /* DD10 error record                  */
#define E_D50    01012      /* DD50 error record                  */
#define E_D11    01013      /* DD11 error record                  */
#define E_D41    01014      /* DD41 error record                  */
#define E_TAPE   01021      /* Tape error record                  */
#define E_PARITY 01030      /* Register parity                    */
#define E_HIPPI  01050      /* HIPPI error                        */
```

`E_GOTS` Error-logging start-up record; when logging is first activated, one of these is sent to the error-logging daemon.

`E_STOP` Error-logging termination record; when it stops logging errors, one of these is sent to the daemon.

`E_TCHG` Time-change record; whenever the system's time of day is changed, one of these is sent to the daemon.

`E_SSD` SSD error record; one of these is generated for each SSD error.

`E_MEM` Memory error record; one of these is generated for each memory error.

`E_SDIS` Marker record signifying that single-bit error detection is disabled.

`E_SEN` Marker record signifying that single-bit error detection is enabled.

`E_IOS` IOS error record.

`E_DSK` Disk error record.

`E_D29` DD-29 error record.

`E_D39` DD-39 error record.

`E_D49` DD-49 error record.

`E_D40` DD-40 error record.

`E_D10` DD-10 error record.

E_D50	DD-50 error record.
E_D11	DD-11 error record.
E_D41	DD-41 error record.
E_TAPE	Tape error structure.
E_PARITY	Register parity error record.
E_HIPPI	HIPPI error record.

The error file contains some administrative records. These include E_GOTS (the start-up record entered into the file when logging is activated), E_STOP (the record written when the error-logging daemon is terminated gracefully), and E_TCHG (the time-change record that accounts for changes in the system's time of day). The formats for these records are defined in the sys/erec.h include file.

FILES

/usr/adm/errfile	Default error-logging file
/usr/include/sys/erec.h	Error-log header format

SEE ALSO

errdemon(8), errpt(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

exports, xtab – Directories to export to NFS clients

SYNOPSIS

```
/etc/exports
/etc/xtab
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/exports file contains entries for directories that can be exported to NFS clients. The exportfs(8) command reads this file automatically. If you change this file, you must run exportfs(8) for the changes to affect the daemon's operation.

The /etc/xtab file contains entries for directories that are currently exported. (To remove entries from this file, use the -u option of exportfs(8).)

An entry for a directory consists of a line that has the following format:

directory -*option*[,*option*]...

directory Path name of a directory

The following are valid options:

ro Exports the directory read-only. If you omit this option, the directory is exported read-write.

rw=*hostname*[:*hostname*]...
Exports the directory read-mostly. *Read-mostly* means read-only to most hosts, but read-write to the hosts that are specified by *hostname*. If you omit this option, the directory is exported read-write to all. ro and rw are mutually-exclusive options.

anon=*uid* Specifies *uid* as the effective user ID when a request comes from an unknown user.

Users who are logged in as root (*uid 0*) are always considered unknown by the NFS server, unless they are included in the root option that follows. The default value for the anon option is –2. To disable anonymous access, set anon = -1.

root=*hostname*[:*hostname*]...
Gives root access only to the root users from a specified host. The default is that no hosts are granted root access.

access=*client*[:*client*]...
Gives mount access to each client listed. The default value allows any machine to mount the given directory.

cksum Checksums packets that are returned to clients.

`krb` Indicates that Kerberos authentication is required for access to this export.

`nosync` Specifies that write operations to this file system are delayed. This option can significantly improve write performance, but its use can cause loss of data if the server crashes before the data is written to disk.

A # symbol anywhere in the file indicates a comment, which extends to the end of the line.

The *client* argument can specify the name of a host or the name of a netgroup. For information on how to use a netgroup file, see `netgroup`(5).

CAUTIONS

You cannot export either a parent directory or a subdirectory of an exported directory that is within the same file system. When both directories reside on the same disk partition, it is illegal, for example, to export both `/usr` and `/usr/local`.

EXAMPLES

An example of an `exports` file follows:

```
/usr       -access=clients           # export to my clients
/usr/local                           # export to the world
/usr2      -access=hermes:zip:aspen  # export to only these machines
/usr/sun   -root=hermes:zip          # give root access only to these
                                     # hosts
/usr/new   -anon=0                   # give all machines root access
/usr/bin   -ro                       # export read-only to everyone
/usr/stuff -access=zip,anon=-3,ro    # several options on one line
/usr/other -rw=host1:host2:host3     # read-write to listed hosts
```

FILES

`/etc/exports` Contains a list of directories that are exportable to NFS clients

`/etc/hosts` Contains a list of known hosts on a network

`/etc/xtab` Contains a list of directories that are currently exported

SEE ALSO

`hosts`(5), `netgroup`(5)

`exportfs`(8), `nfsd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

exrc – Start-up files for ex(1) and vi(1)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The .exrc file is a start-up file for the ex(1) and vi(1) text editors. In this file, you can enter editor commands that you want the editor to execute every time you edit a file. When you invoke ex(1) or vi(1), the editor checks for a file named .exrc in your home directory ($HOME) and runs editor commands it finds there. Then the editor checks for a file named .exrc in the current directory. The .exrc file in the current directory is, by default, processed only if you are the owner. This default can be changed only to allow the processing of .exrc files that you do not own at the time the editor command is built.

You can enter each editor command on a separate line or enter several separate commands on the same line and separate the commands with a | symbol.

NOTES

If you have the EXINIT environment variable defined, the .exrc files are not processed.

EXAMPLES

Three useful editor commands that are commonly included in .exrc files are shown in the following examples. For a complete description of all editor commands, see ex(1) or vi(1).

Example 1: In the following example, the file contains the set command twice with two separate options. The first option turns off wrapscan, so that when a search is in progress, the editor does not wrap to the beginning when the end of the file has been reached. The second option sets showmatch, which tells the editor to show the match to a right parenthesis ( )) or right brace (}) when either of these characters is entered.

```
set nowrapscan | set showmatch
```

Example 2: To replace a keystroke entered in the vi(1) command mode with a series of vi(1) commands, use the map command. The following example uses map to set the "=" character to the vi(1) command 5x. When a user types the string =, vi(1) executes 5x, deleting 5 characters.

```
map  =  5x
```

Example 3: The vi(1) command, abbreviate, is specified in the following example. When this command is in effect, vi(1) automatically replaces string "cri" with "Cray Research, Incorporated" when you type the string in insert mode. (This substitution occurs only when "cri" is surrounded by spaces, tabs, or punctuation; this ensures that substitutions do not occur in the middle of words.)

```
abbreviate cri Cray Research, Incorporated
```

FILES

`$HOME/.exrc`	Editor start-up file in your home directory
`./.exrc`	Editor start-up file in the current directory

SEE ALSO

`ex`(1), `vi`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

fcntl – File control options

SYNOPSIS

```
#include <fcntl.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The fcntl(2) system call provides control over open files. The fcntl.h include file describes the available requests and arguments to fcntl(2) and open(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open)     */

#define O_RDONLY   0
#define O_WRONLY   1
#define O_RDWR     2
#define O_NDELAY   04      /* Non-blocking I/O                        */
#define O_APPEND   010     /* append (writes guaranteed at the end)   */
#define O_SYNC     020     /* synchronous write option                */
#define O_NONBLOCK 040     /* Non-blocking I/O (POSIX)                */
#define O_RAW      0100    /* direct to user space                    */
                           /* (no system buffering) I/O               */
#define O_SSD      0200    /* I/O addresses are SDS relative          */
                           /* (CRAY Y-MP systems) */
#define O_ASYNC    0200000 /* Asynch I/O: for sockets                 */

/* Flag values accessible only to open(2) */

#define O_CREAT    000400  /* open with file create                   */
                           /* (uses third open arg)                   */
#define O_TRUNC    001000  /* open with truncation                    */
#define O_EXCL     002000  /* exclusive open                          */
#define O_NOCTTY   004000  /* No controlling TTY (POSIX)              */
#define O_RESTART  040000  /* create file as a restart file           */

/* fcntl(2) requests */

#define F_DUPFD     0      /* Duplicate fildes                        */
#define F_GETFD     1      /* Get fildes flags                        */
#define F_SETFD     2      /* Set fildes flags                        */
```

```
#define F_GETFL     3      /* Get file flags                          */
#define F_SETFL     4      /* Set file flags                          */
#define F_GETLK     5      /* Get file lock                           */
#define F_SETLK     6      /* Set file lock                           */
#define F_SETLKW    7      /* Set file lock and wait                  */
#define F_CHKFL     8      /* Check legality of                       */
                           /* file flag change                        */
                           /* Internal use only                       */
#define F_GETOWN    9      /* Get SIGIO/SIGURG proc/pgrp              */
#define F_SETOWN   10      /* Set SIGIO/SIGURG proc/pgrp              */

/* file segment locking set data type                                 */
/*      - information passed to system by user                        */

struct flock {
       short       l_type;
       short       l_whence;
       long        l_start;
       long        l_len;  /* len = 0 means until end of file        */
       short       l_sysid;
       short       l_pid;
};

/* file segment locking types */

#define F_RDLCK    01      /* Read lock                               */
#define F_WRLCK    02      /* Write lock                              */
#define F_UNLCK    03      /* Remove lock(s)                          */
```

FILES

`/usr/include/fcntl.h`	File control options file

SEE ALSO

`fcntl`(2), `open`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

fs – File system partition format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/map.h>
#include <sys/fs/nc1filsys.h>
#include <sys/fs/c1filsys.h>
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The NC1FS file system is created by The mkfs(8) command. For further information about the file system structure of the NC1FS file system, see *General UNICOS System Administration*, Cray Research publication SG–2301.

The format of the file system blocks for the NC1FS file system is as follows:

```
/*
 * Inode region descriptor.
 * The first block of an inode region is a
 * bit map for the inodes in that region.
 */

struct nc1ireg_sb        {
        uint    i_unused:16,    /* reserved                              */
                i_nbl   :16,    /* number of blocks                      */
                i_sblk  :32;    /* start block number                    */
};

struct nc1ireg_db        {
        uint    i_avail;        /* number of available inodes            */
};
#define NC1MAXIREG      4       /* Maximum inode regions per partition   */
#define NC1IMAPBLKS     1       /* number of blocks in inode map         */

struct      nc1fdev_sb
{
        long            fd_name;                /* Physical device name  */
        uint            fd_sblk :32,            /* Start block number    */
                        fd_nblk :32;            /* Number of blocks      */
```

```
        struct nc1ireg_sb  fd_ireg[NC1MAXIREG]; /* Inode regions          */
};

struct      nc1fdev_db
{
        int                fd_flag;              /* flag word             */
        struct nc1ireg_db  fd_ireg[NC1MAXIREG];  /* Inode regions         */
};

#define FDNC1_DOWN      01      /* Slice not available                     */
#define FDNC1_RDONLY    2       /* Slice is read only                      */
#define FDNC1_NOALLOC   4       /* Slice is not available for allocation   */
#define FDNC1_SBDB      010     /* Slice has valid FS tables               */
#define FDNC1_RTDIR     020     /* Slice has valid ROOT Inode and directory */
#define FDNC1_SECALL    0100    /* Slice sector allocated                  */

#define NC1MAXPART      64      /* Maximum number of partitions            */


/*
 * Structure of the super-block
 */

struct  nc1filsys
{
        long    s_magic;        /* magic number to indicate file system type  */
        char    s_fname[8];     /* file system name                           */
        char    s_fpack[8];     /* file system pack name                      */
        dev_t   s_dev;          /* major/minor device, for verification       */

        daddr_t s_fsize;        /* size in blocks of entire volume            */
        int     s_isize;        /* Number of total inodes                     */
        long    s_bigfile;      /* number of bytes at which a file is big     */
        long    s_bigunit;      /* minimum number of blocks allocated for
                                   big files                                  */
        long    s_secure;       /* security: secure FS label                  */
        int     s_maxlvl;       /* security: maximum security level           */
        int     s_minlvl;       /* security: minimum security level           */
        long    s_valcmp;       /* security: valid security compartments      */
        time_t  s_time;         /* last super block update                    */
        blkno_t s_dboff;        /* Dynamic block number                       */
        ino_t   s_root;         /* root inode                                 */
        struct  nc1dblock *s_pdb; /* pointer to dynamic block (when mounted)  */
        blkno_t s_mapoff;       /* Start map block number                     */
```

```
        int     s_mapblks;      /* Last map block number                        */
        int     s_nscpys;       /* Number of copies of s.b per partition        */
        int     s_npart;        /* Number of partitions                         */
        int     s_ifract;       /* Ratio of inodes to blocks                    */
        blkno_t s_sfs;          /* reserved                                     */
        long    s_flag;         /* Flag word                                    */

        struct  nc1fdev_sb s_part[NC1MAXPART];  /* Partition descriptors      */
        int     s_iounit;       /* Physical block size                          */
        long    s_numiresblks;  /* number of inode reservation blocks           */
                                /* per region (currently 1)                     */
                                /* 0 = 1*(AU) words, n = (n+1)*(AU) words       */
        long    s_priparts;     /* bitmap of primary partitions                 */
        long    s_priblock;     /* block size of primary partition(s)           */
                                /* 0 = 1*512 words, n = (n+1)*512 words         */
        long    s_prinblks;     /* number of 512 wds blocks in primary          */
        long    s_secparts;     /* bitmap of secondary partitions               */
        long    s_secblock;     /* block size of secondary partition(s)         */
                                /* 0 = 1*512 words, n = (n+1)*512 words         */
        long    s_secnblks;     /* number of 512 wds blocks in secondary        */
        long    s_sbdbparts;    /* bitmap of partitions with file system data */
                                /* including super blocks, dynamic block        */
                                /* and free block bitmaps (only primary         */
                                /* partitions may contain these)                */
        long    s_rootdparts;   /* bitmap of partitions with root directory   */
                                /* (only primary partitions)                    */
        long    s_nudparts;     /* bitmap of no-user-data partitions            */
                                /* (only primary partitions)                    */
        long    s_fill[94];     /* reserved                                     */
};

struct      nc1dblock
{
        long    db_magic;       /* magic number to indicate file system type  */
        daddr_t db_tfree;       /* total free blocks                            */
        int     db_ifree;       /* total free inodes                            */
        int     db_ninode;      /* total allocated inodes                       */
        long    db_state;       /* file system state                            */
        time_t  db_time;        /* last dynamic block update                    */
        long    db_type;        /* type of new file system                      */
        int     db_spart;       /* Partition from which system mounted          */
        int     db_ifptr;       /* Inode allocation pointer                     */
        int     db_actype;      /* device accounting type (for billing)         */
        long    db_flag;        /* Flag word                                    */
```

```
        long    db_res1;        /* reserved                                  */
        struct  buf *db_fbuf;   /* Free block map buffer descriptor          */
        struct  map db_fpmap;   /* Free block map header - primary parts     */

        struct  nc1fdev_db db_part[NC1MAXPART];  /* Partition descriptors    */
        lockinfo_t  db_lockinf; /* proc of the process locking the filesystem */
        int     db_dpfptr;      /* primary partitions allocation pointer     */
        int     db_dsfptr;      /* secondary partitions allocation pointer   */
        daddr_t db_sfree;       /* secondary parts free blocks               */
        struct  map db_fsmap;   /* Free block map header - secondary parts   */
        long    db_fill[157];   /* reserved                                  */
};
#define db_proc db_lockinf.hi_proc /* proc of process locking filesystem     */
#define db_fptr db_ifptr
#define db_fmap db_fpmap

/*
 * Filesystem flags
 */

#define         Fs_NOSPC        1       /* Filesystem out of space           */
#define         Fs_RRFILE       2       /* Round robin file allocation       */
#define         Fs_RRALLDIR     4       /* Round robin all directories       */
#define         Fs_RR1STDIR     010     /* Round robin 1st level directories */
#define         Fs_RDONLY       020     /* File system read only             */
#define         Fs_CHECKED      040     /* File system checked               */
#define         Fs_MOUNTED      0100    /* File system mounted               */
#define         Fs_WANTED       0200    /* File system lock wanted           */
#define         Fs_LOCKED       0400    /* File system locked                */
#define         Fs_UPDATE       01000   /* File system update in progress    */
#define         Fs_WUPDAT       02000   /* File system wakeup after update   */
#define         Fs_RRALLUDATA   020000  /* Round robin all user file data    */
#define         Fs_DIRTY        0100000 /* File system dirty                 */
#define         Fs_SFS          010000000  /* File system shared             */


#define FsMAGIC_NC1 0x6e6331667331636e  /* s_magic number                    */
#define DbMAGIC_NC1 0x6e6331646231636e  /* db_magic number                   */

#define FsSECURE 0xcd076d1771d670cd     /* s_secure: secure file system      */

#define NC1NSUPER       10              /* Copies of s.b. per partition      */
#define NC1MINPARTSZ    (6+NC1NSUPER)   /* Minimum blocks per partition      */
```

```
#define NC1DB(fp) \
      ((struct nc1dblock *)(fp->s_pdb))
#define nc1getfs(mp)    \
        ((struct nc1filsys *)(((struct buf *)(mp)->m_bufp)->b_waddr))
#define nc1getdb(mp)    \
        ((struct nc1dblock *)(((struct buf *)(mp)->m_dbufp)->b_waddr))
```

FILES

`/usr/include/sys/fs/nc1filsys.h`	Format of file system partitions for `NC1FS` file systems
`/usr/include/sys/map.h`	Definitions for bit map management
`/usr/include/sys/param.h`	System parameter file
`/usr/include/sys/types.h`	Data type definition file

SEE ALSO

`dir`(5), `dirent`(5), `inode`(5)

`fsck`(8), `mkfs`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

fslrec – File system error log record format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/fslrec.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

File system error log records are written to /dev/fslog by the UNICOS kernel as part of the panic-less file system feature. The file system error log daemon, fslogd(8), reads and processes log records.

The format of each file system error log record is defined as follows:

```
struct fslgrec {
       time_t fsl_time;       /* time (seconds since '70)          */
       int    fsl_type;       /* error type                        */
       int    fsl_subtype;    /* error sub-type                    */
       char   *fsl_ptr;       /* generic pointer to struct in err  */
};
```

The following list summarizes the file system error log record types:

FSLG_GO	(Deferred) File system error log start record
FSLG_STOP	(Deferred) File system error log stop record
FSLG_FS	The UNICOS kernel has detected a file system data structure error
FSLG_DIR	The UNICOS kernel has detected a directory block error
FSLG_INODE	The UNICOS kernel has detected an error in the memory copy of a file inode

FILES

/dev/fslog	File system error log device
/usr/include/sys/fslog.h	File system error log header file
/usr/include/sys/fslrec.h	Format of file system error log record
/usr/include/sys/types.h	Data type definition file

SEE ALSO

`fslog`(4)

`fslogd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

fstab, mntent – File that contains static information about file systems

SYNOPSIS

```
#include <mntent.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/fstab file describes the file systems and swapping partitions that UNICOS uses. The "mount *directory*" form of the mount(8) command uses this information. The command searches the /etc/fstab file for an entry that has a mount point named *directory* and mounts the file system as the entry describes.

File system quota commands also use this information. For more information, see quadmin(8), quota(1), and qudu(8).

The mfsck(8) command also uses this information. For more information, see mfsck(8).

The system administrator creates the fstab file by using a text editor or the UNICOS installation and configuration menu system. The mount(8) command processes it as a source of default options. The /etc/fstab file is not changed by programs; it is only read. The system administrator must properly create and maintain this file.

The controlling agent for mounting root file systems is the /etc/config/rcoptions file, which is defined with the UNICOS installation and configuration menu system and is used at system startup.

The /etc/fstab file consists of several lines of the following form:

filesystem directory type options frequency passnumber

Each line in the file constitutes a file system entry. The entry fields are separated by white space. The mntent structure definition explains the meaning of each field. A # as the first nonwhite character on a line indicates a comment.

The entries in /etc/fstab are accessed using the routines in getmntent(3C), which return a structure that has the following format:

```
struct mntent {
       char *mnt_fsname;   /* file system name             */
       char *mnt_dir;      /* file system path prefix      */
       char *mnt_type;     /* file system type             */
       char *mnt_opts;     /* ro, quota, etc.              */
       int  mnt_freq;      /* dump frequency, in days      */
       int  mnt_passno;    /* pass number on parallel fsck */
};
```

`mnt_fsname` Name of the block special file to be mounted.

`mnt_dir` Directory mount point for the special file.

`mnt_type` Type of file system specified in `mnt_fsname`. Valid types are `NC1FS`, `NFS`, `PROC`, `INODE`, `SFS`, and `ignore`. If the `mnt_type` is specified as `ignore`, the entry is ignored. This is useful for showing disk partitions that are currently unused.

`mnt_opts` String of comma-separated options. The description of the `fsckopt` and `quota` options follow, but the other options are documented with `mount`(8).

`mnt_freq` Optional field referenced by the `-w` option of the `dump`(8) command to determine the frequency of system dumps.

`mnt_passno` Optional field referenced by the `mfsck`(8) program to determine the order that file systems are checked using the `fsck`(8) command.

`fsckopt` Specifies the file system `mfsck`(8) options when invoking `fsck`(8). This option takes the following form:

`fsckopt=q`

Using `q` as the `fsckopt` specifies that `mfsck`(8) use file system flags to determine when the file system is checked. Specifying `u` as the `fsckopt` implies an unconditional file system check, which is the default.

`quota` Specifies the file quota configuration of the `mnt_opts` entry. This option takes one of three forms:

1. `quota`=*quota_file_relative_name*

   This form is used if the `quota` control file will reside on the file system it controls. The file name is relative to the root directory of the file system, and if the default name is used as recommended, the option generally would be written as `quota=$QFILE`.

   The special name `$QFILE` means the default `quota` file name (as defined in `quadmin`(8)). The default name is `.Quota60` so that the preceding `quota` option would resolve to `quota=.Quota60` in the root directory of the file system.

2. `quota`=*quota_file_full_name*

   This form is used if the `quota` control files will reside in some arbitrary place (for example, if the `quota` files were to reside in the `/etc/admin/quota60` directory, this form could be written as `quota=/etc/admin/quota60/$FILESYS`).

   The special name `$FILESYS` is the last component of the *filesystem* name on this `fstab` line. If this line had been written as

   ```
   /dev/dsk/slash_b /b NC1FS quota=/etc/admin/quota60/$FILESYS
   ```

   it would be resolved to

```
quota=/etc/admin/quota60/slash_b
```

A directory, `quota60`, was created to hold all of the `quota` control files. The file system name identifies each individual `quota` control file within the directory.

3. `quota`*=/dev/dsk/filesystem_name*

   This form shows that this file system is under the control of a `quota` file defined and used to control another file system. When multiple file systems are controlled as a group, this form is used. For example, assume that three lines from `/etc/fstab` were written as follows:

```
/dev/dsk/tmp_1 /tmp_1 NC1FS quota=$QFILE
/dev/dsk/tmp_2 /tmp_2 NC1FS quota=/dev/dsk/tmp_1
/dev/dsk/tmp_3 /tmp_3 NC1FS quota=/dev/dsk/tmp_1
```

   These lines define the `quota` control file as `.Quota60` residing in the root directory of `/dev/dsk/tmp_1`. The same `quota` control file controls file systems `/dev/dsk/tmp_2` and `/dev/dsk/tmp_3`; therefore, the `quota` information for usage of any or all of the three file systems is common and reflects the combined usages of all three.

   The rule for using this form is if the right-hand side of a `quota` option matches one of the other file system names in `/etc/fstab`, it is the third form of declaration (as defined previously), and the file system must contain a `quota` option naming a file. Only one level of indirection is supported.

EXAMPLES

File system `/usr/sierra` from remote host `sierra` will be mounted on local directory `/nfs/sierra`. File system type is NFS with options `bg`, `soft`, `rsize`, and `wsize`. For a description of the options, see `mount`(8).

```
sierra:/usr/sierra  /nfs/sierra  NFS  soft,bg,rsize=8192,wsize=8192
```

Mount the `/proc` file system on the `/proc` directory.

```
/proc   /proc   PROC
```

FILES

`/etc/fstab`	File system static information
`/usr/include/mntent.h`	Structure definition of `fstab` entries

SEE ALSO

`mnttab`(5)

`quota`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`getmntent`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`dump`(8), `fsck`(8), `mfsck`(8), `mount`(8), `quadmin`(8), `qudu`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

ftpusers – List of unacceptable ftp(1B) users

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/ftpusers file contains a list of unacceptable ftp(1B) users, one user name per line. When ftp(1B) is run, ftpd(8) checks ftpusers for the login name of the user trying to open a connection. If the user's login name appears in the file, ftpd(8) denies the user access.

If ftpusers is nonexistent or empty, all valid UNICOS users are considered valid users of ftp(1B).

FILES

/etc/ftpusers File that contains unacceptable ftp(1B) users

SEE ALSO

ftp(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

ftpd(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`gated-config` – Gated configuration file syntax

SYNOPSIS

`/etc/gated.conf`

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The `gated-config` file consists of a sequence of statements. Statements are composed of tokens separated by white space. Most statements are terminated by a ; symbol. However, directive statements are terminated with a newline. For most statements, white space may contain any combination of blanks, tabs, and newlines; however, directive statements may use only blanks and tabs.

Comments start with a # symbol and run to the end of the line.

There are eight classes of statements in the following list. Statements from the first two classes may occur anywhere in the file.

Class	Description
Directive	Specifies included files and the current directory. The parser immediately acts on directives.
Trace option	Controls tracing options.

You must specify the six remaining classes in the following order:

Class	Description
Options	Allows specification of some global options.
Interface	Specifies interface options.
Definition	Specifies options, the autonomous system, and martian networks.
Protocol	Enables or disables protocols, and sets protocol options.
Route	Defines static routes by using `static` statements.
Control	Defines routes that are imported from routing peers and routes that are exported to these peers.

Detailed definitions of these classes of statements follow. Primitives that are used in the following definitions are as follows:

host Any host. You can specify a host by its IP address or by a domain name. If you specify a domain name with multiple IP addresses, it is considered an error. The host bits in the IP address must be nonzero.

network Any network. You can specify a network by its IP address or a network name. The host bits in a network specification must be 0. To specify the default network (0.0.0.0), you also may use `default`.

destination Any host or network.

dest_mask Any host or network that has an optional mask. *dest_mask* can be any of the following formats:

```
all
network
network mask mask
network mask-length bits
host host
```

A mask is a dotted quad that specifies which bits of the destination are significant. To indicate that any IP address can be matched, use `all`. You may use the number of contiguous *bits* instead of an explicit mask.

gateway A host on an attached network.

interface An interface specified by IP address, domain name, or interface name. Be careful with the use of interface names because future UNIX operating systems may allow more than one address per interface.

gateway_list A list of one or more gateways.

interface_list A list of one or more interface names, wildcard names, or addresses, or the token `all`; `all` refers to all interfaces. A wildcard name is an interface name without the number.

preference A number between 0 and 255; 0 is the most preferred, and 255 is the least preferred. *preference* determines the order of routes to the same destination in the routing table. `gated` allows one route to a destination per protocol per autonomous system. For multiple routes, the route to use is chosen by *preference.*

When a *preference* tie exists, if the two routes are from the same protocol and from the same autonomous system, `gated` chooses the route that has the lowest metric. Otherwise, `gated` selects the route with the lowest numeric next-hop gateway address.

metric A valid metric for the specified protocol.

Directives Statements

Directive statements are as follows:

`%directory` *path_name*

Sets the current directory to *path_name*. This is the directory in which `gated` looks for included files that do not begin with a / symbol.

This statement does not actually change the current directory, it simply specifies the prefix applied to included file names.

`%include` *filename*

Causes the specified file to be parsed completely before resuming with this file. Nesting up to 10 levels is supported. To increase the maximum nesting level, change the definition of `FI_MAX` in `parse.h`.

Trace Option Statements

Trace option statements are as follows:

`tracefile` [*filename* [`replace`]] [`size` *size*[`k | m`] `files` *files*];

Specifies the file to contain tracing output. If you specify *filename*, trace information is appended to this file, unless you specify `replace`.

If specified, *size* and *files* cause the trace file to be limited to *size*, with *files* files kept (including the active file). To create the back-up file names, append a period and a number to the trace file name, starting with `.0`. The minimum size that you can specify is 10Kbytes, the minimum number of files that you can specify is 2. The default is not to rotate log files.

`traceoptions` [*traceoption* [*traceoption* [`. . .`]] [`except` *traceoption* [*traceoption* [`. . .`]]];

Changes the tracing options to those specified. If you do not specify any options, tracing is turned off. If you specify the `except` keyword, flags listed before the keyword are turned on, and flags listed after it are turned off. This is a simple method to turn on all but a few flags. Trace flags are as follows:

Flag	Description
`all`	Turns on all of the following options except `nostamp`.
`external`	Produces external error messages.
`general`	Turns on `internal`, `external`, and `route`.
`icmp`	Lists ICMP redirect packets sent and received. To modify it, use `update`. Redirect packets that are processed are traced under the `route` option.
`internal`	Produces internal error and informational messages.
`kernel`	Changes to the kernel's routing table.
`mark`	Indicates that a message will be sent to the trace log every 10 minutes to ensure that `gated`(8) is still running.
`nostamp`	Specifies that all messages in the trace file should not be time-stamped.
`ospf`	Lists OSPF packets sent and received. To modify it, use `protocol`.
`parse`	Lists tokens that the parser recognizes in the configuration file.
`protocol`	Provides messages about protocol state machine transitions when used with `ospf` or `kernel`.
`rip`	Lists RIP packets sent and received. To modify it, use `update`.
`route`	Changes to the `gated`(8) routing table.

`task` Displays task scheduling, signal handling, and packet reception.

`timer` Displays timer scheduling.

`update` Traces the contents of protocol packets.

Options Statements

Options statements are as follows:

`options` *option_list ;*

Sets `gated` options. *option_list* can have these following values:

`noinstall` Does not change the kernel routing table. Useful for verifying configuration files.

`noresolv` Does not try to resolve symbolic names into IP addresses by using the host or network tables or domain name system (DNS). This option is intended for systems in which a lack of routing information could cause a DNS lookup to hang.

`nosend` Does not send any packets. This lets you run `gated`(8) on a live network to test protocol interactions without actually participating in the routing protocols. To verify that `gated`(8) is functioning properly, examine the packet traces in the `gated`(8) log. This is most useful for the RIP protocol.

`syslog` [`upto` *log_level*] *log_level*

Controls the amount of data `gated`(8) logs by using the system log on systems in which the `setlogmask`() routine is supported. The `setlogmask`(3C) man page defines the log levels and other terminology. The default is equivalent to `syslog upto info`.

Interface Statements

Definition statements are as follows:

```
interfaces {
   options [strictinterfaces] [scaninterval time] ;
   interface interface_list interface_options ;
   define address [broadcast broad_addr|pointopoint local_addr ]
      [netmask subnetmask] [multicast] ;
} ;
```

The interface statement includes the following parameters:

`options` Sets some global options related to interfaces. The options are as follows:

`strictinterfaces`

Indicates that it is a fatal error to reference an interface in the configuration file that is not listed in a `define` statement or not present when `gated`(8) is started. Without this option, a warning message is issued and `gated`(8) continues.

`scaninterval` *time*
Specifies how often `gated`(8) scans the kernel interface list for changes. The default is every 15 seconds. `gated`(8) also scans the interface list on receipt of a `SIGUSR2` signal.

`interface` Sets interface options on the specified interfaces. *interface_list* options are as follows:

`all` Specifies the options that apply to all interfaces.

interface_list Specifies a list of interface names, domain names, or numeric addresses. See the warning about interface names in the DESCRIPTION section.

The options are as follows:

`preference` *pref*
Sets the preference for routes to this interface when it is up. The default is 0.

`down preference` *pref*
Sets the preference for routes to this interface when `gated`(8) believes it to be down because of a lack of routing information received. The default is 120.

`passive` Prevents `gated` from changing the preference of the route to this interface if it is believed to be down because of a lack of routing information received.

`define` Defines interfaces that may not be present when `gated`(8) is started. If you specify `strictinterfaces`, `gated`(8) considers it an error to reference a nonexistent interface in the configuration file. This clause allows specification of that interface so that it can be referenced in the configuration file.

Definition keywords are as follows:

`broadcast` *broad_addr*
Defines the interface as broadcast capable (for example, Ethernet and FDDI), and specifies the broadcast address.

`pointopoint` *local_addr*
Defines the interface as a point-to-point interface, and specifies the address on the local side. For this type of interface, the *address* parameter specifies the address of the remote host.

An interface not defined as `broadcast` or `pointopoint` is assumed to be nonbroadcast multiaccess (NBMA), such as HIPPI.

`netmask` *subnetmask*
Specifies the nonstandard subnet mask to be used on this interface. This mask is currently ignored on point-to-point interfaces.

`multicast` Specifies that the interface is multicast-capable.

Definition Statements

Definition statements are as follows:

`autonomoussystem` *autonomous_system*;
: Sets the autonomous system of this router to be *autonomous_system*.

`routerid` *host*;
: Sets the router identifier for use by the OSPF protocol. The default is the address of the first interface `gated`(8) encounters. The address of a nonpoint-to-point interface is preferred over the local address of a point-to-point interface. The most preferred is an address on a loopback interface that is not the loopback address (127.0.0.1).

`martians {`*martian_list*`};`
: Defines a list of martian addresses about which all routing information is ignored. The *martian_list* is a semicolon-separated list of symbolic or numeric hosts that has optional masks. See *dest_mask*. You also may specify the `allow` parameter to allow explicitly a subset of a range that was disallowed.

Protocol Statements

Protocol statements enable or disable use of a protocol and control protocol options. You may specify the protocols in any order. For all protocols, `preference` controls the choice of routes learned through this protocol or from this autonomous system in relation to routes learned from other protocols or autonomous systems.

The default metric used when exporting routes learned from other protocols is specified by using `defaultmetric`, which itself defaults to the highest valid metric for this protocol; for the RIP protocol, this signifies a lack of reachability.

For distance vector protocols (RIP) and redirects (ICMP), the `trustedgateways` clause supplies a list of gateways that provides valid routing information, and routing packets from others are ignored. This defaults to all gateways on the attached networks. Routing packets may be sent not only to the remote end of point-to-point links and the broadcast address of broadcast-capable interfaces, but also to specific gateways if they are listed in a `sourcegateways` clause and `yes` or `on` is specified. If you specify `nobroadcast`, routing updates are sent only to gateways listed in the `sourcegateways` clause, and not to the broadcast address. To disable the transmission and reception of routing packets for a particular protocol, use the `interface` clause. To override an `interface` clause that disables sending or receiving protocol packets for specific peers, use the `trustedgateways` and `sourcegateways` clauses.

Any protocol can have a `traceoptions` clause, which enables tracing for a particular protocol, group, or peer. The allowable protocol-specific options are `all`, `general`, `internal`, `external`, `route`, `update`, `task`, `timer`, `protocol`, or `kernel`.

`rip` Statement

One of the most widely used interior gateway protocols is the Routing Information Protocol (RIP). It classifies routers as active and passive (silent). Active routers advertise their routes (reachability information) to others; passive routers listen and update their routes based on advertisements, but they do not advertise. Typically, routers run RIP in active mode, and hosts use passive mode.

A router running RIP in active mode broadcasts updates at set intervals. Each update contains paired values in which each pair consists of an IP network address and an integer distance to that network. RIP uses a hop count metric to measure the distance to a destination. In the RIP metric, a router advertises directly connected networks at a metric of 1. Networks that are reachable through one other gateway are two hops, and so on. Thus, the number of hops or the hop count along a path from a given source to a given destination refers to the number of gateways that a datagram would encounter along that path.

A RIP routing daemon dynamically builds on information received through RIP updates. When started, it issues a request for routing information and then listens for responses to the request. If a system configured to supply RIP hears the request, it responds with a response packet based on information in its routing database. The response packet contains destination network addresses and the routing metric for each destination.

When a RIP response packet is received, the routing daemon takes the information and rebuilds the routing database adding new routes and "better" (lower metric) routes to destinations already listed in the database. RIP also deletes routes from the database if the next router to that destination indicates that the route contains more than 15 hops, or if the route is deleted. If no updates are received from that gateway for a specified time period, all routes through a gateway are deleted. Generally, routing updates are issued every 30 seconds. In many implementations, if a gateway is not heard from for 180 seconds, all routes from that gateway are deleted from the routing database. This 180-second interval also applies to deletion of specific routes.

RIP version 2 (more commonly known as RIP II) adds additional capabilities to RIP. For more information about RIP II, see RFC 1388.

The syntax for the `rip` statement follows:

```
rip yes | no | on | off [ {
 broadcast ;
 nobroadcast ;
 nocheckzero ;
 preference preference ;
 defaultmetric metric ;
 interface interface_list
  [noripin]
  [noripout]
  [metricin metric]
  [metricout metric]
  [version 1]|[version 2 [multicast| broadcast]]
  [authentication [none | password]] ;
 trustedgateways gateway_list ;
 sourcegateways gateway_list ;
 traceoptions trace_options ;
} ] ;
```

The `rip` statement enables or disables RIP. If you do not specify the `rip` statement, the default is `rip on`. If enabled, RIP assumes `nobroadcast` when only one interface exists and `broadcast` when more than one exists.

The options are as follows:

`broadcast` Specifies that RIP packets are broadcast regardless of the number of interfaces present. This option is useful when propagating static routes or routes learned from anther protocol into RIP. In some cases, the use of `broadcast` when only one network interface is present can cause data packets to traverse a single network twice.

`nobroadcast` Specifies that RIP packets are not broadcast on attached interfaces, even if more than one exists. If a `sourcegateways` clause is present, routes are still unicast directly to that gateway.

`nocheckzero` Specifies that RIP should not check to make sure that reserved fields in incoming version 1 RIP packets are 0. Usually, when the reserved fields are not 0, RIP rejects packets.

`preference` *preference*
Sets the preference for routes learned from RIP. The default preference is 100. To override this preference, specify a preference in import policy.

`defaultmetric` *metric*
Defines the metric used when advertising routes by using RIP that were learned from other protocols. If you omit this option, the default value is 16 (unreachable). This choice of default values requires you to specify a metric explicitly to export routes from other protocols into RIP. To override this metric, specify a metric in export policy.

`interface` *interface_list*
Controls various attributes of sending RIP on specific interfaces. For the description of the *interface_list*, see the section on *interface_list* specification. The following are the possible parameters:

`noripin` Specifies that RIP packets received using the specified interface are ignored. The default is to listen to RIP on all interfaces.

`noripout` Specifies that no RIP packets are sent on the specified interfaces. When in `broadcast` mode, the default is to send RIP on all interfaces.

`metricin` *metric*
Specifies the RIP metric to add to incoming routes before they are installed in the routing table. The default is the kernel interface metric plus 1 (which is the default RIP hop count). If you specify this value, it is used as the absolute value. The kernel metric is not added. This option is used to make RIP routes learned through the specified interfaces less preferable than RIP routes from other interfaces.

`metricout` *metric*
Specifies the RIP metric to be added to routes that are sent using the specified interfaces. The default is 0. This option is used to make other routers prefer RIP routes from other interfaces over RIP routes learned using the specified interfaces.

`version 1` Specifies that RIP packets sent using the specified interfaces are version 1 packets. This is the default.

`version 2` Specifies that RIP version 2 packets are sent on the specified interfaces. If Internet Protocol (IP) multicast support is available on this interface, the default is to send full version 2 packets. If it is not available, version 2 packets that are compatible with version 1 are sent.

`multicast` Specifies that RIP version 2 packets should be multicast on this interface. This is the default for RIP version 2.

`broadcast` Specifies that RIP version 2 packets that are compatible with version 1 should be broadcast on this interface, even if IP multicast is available.

`authentication`
Defines the authentication type to use. It applies only to RIP version 2 and is ignored for RIP version-1 packets. The default authentication type is `none`. If you specify a *password*, the authentication type used is simple. The *password* should be a quoted string between 0 and 16 characters.

`trustedgateways` *gateway_list*
Defines the list of gateways from which RIP will accept updates. The *gateway_list* is simply a list of host names or IP addresses. By default, all routers on the shared network are trusted to supply routing information. But, if you specify the `trustedgateways` clause, only updates from the gateways in the *gateway_list* are accepted.

`sourcegateways` *gateway_list*
Defines a list of routers to which RIP sends packets directly, not through multicast or broadcast. By default, RIP packets are broadcast to every system on the shared network. If you use the `sourcegateways` statement, updates are sent only to the gateways in the *gateway_list*.

`traceoptions` *trace_options*
Specifies the tracing options for RIP (see the Trace Options subsection of this man page).

Open Shortest Path First (OSPF) protocol

Open Shortest Path First (OSPF) routing protocol is a shortest path first (SPF) or link-state protocol. OSPF is an interior gateway protocol that distributes routing information between routers in a single autonomous system. OSPF is suitable for complex networks that have many routers. Each network that has at least two attached routers has a designated router and a back-up designated router. The designated router floods a link-state advertisement for the network and has other special responsibilities. The designated router concept reduces the number of adjacencies required on a network.

OSPF allows networks to be grouped into areas. Routing information passed between areas is abstracted, potentially allowing a significant reduction in routing traffic. OSPF uses four different types of routes, listed in order of preference: intra-area, inter-area, type 1 external, and type 2 external. Intra-area paths have destinations within the same area; inter-area paths have destinations in other OSPF areas; and Autonomous System External (ASE) routes are routes to destinations external to the AS. Routes imported into OSPF as type 1 ASE routes are supposed to be from peers whose external metrics are directly comparable to OSPF metrics. Type 2 ASEs are used for peers whose metrics are not comparable to OSPF metrics.

OSPF intra- and inter-area routes are always imported into the `gated`(8) routing database with a preference of 10. If an OSPF router did not participate fully in the area's OSPF, it would be a violation of the protocol, it would update the protocal; therefore, you cannot override this. Although you can give other routes lower preference values explicitly, you should not do so.

Hardware multicast capabilities also are used when possible to deliver link-status messages.

OSPF areas are connected by the backbone area, the area with identifier 0.0.0.0. All areas must be logically contiguous, and the backbone is no exception. To permit maximum flexibility, OSPF allows the configuration of virtual links to enable the backbone area to appear contiguous despite the physical reality.

All routers in an area must agree on that area's parameters. Most configuration parameters are defined on a per area basis. All routers that belong to an area must agree on that area's configuration.

`ospf` Statement

The syntax for the `ospf` statement follows:

```
ospf yes | no | on | off [ {
  defaults {
      preference preference ;
      cost cost ;
      tag [ as ] tag ;
      type 1 | 2 ;
  } ;
  exportlimit routes ;
  exportinterval time ;
  traceoptions trace_options ;
  monitorauthkey authkey ;
  backbone | ( area area ) {
        authtype 0 | 1 | none | simple ;
        stub [ cost cost] ;
        networks {
              network ;
              network mask mask ;
              network masklen number ;
              host host ;
        } ;
        stubhosts {
              host cost cost ;
```

```
            } ;
            interface interface_list; [cost cost ] {
                  interface_parameters
            } ;
            interface interface_list nonbroadcast [cost cost ] {
                  pollinterval time ;
                  routers {
                        gateway [ eligible ] ;
                  } ;
                  interface_parameters
            } ;
            Backbone only:
            virtuallink neighborid router_id transitarea area {
                  interface_parameters
      } ;
   } ;
} ] ;
```

The following are the *interface_parameters* referred to previously. You may specify them on any class of interface, and they are described under the `interface` clause.

```
enable | disable ;
retransmitinterval time ;
transitdelay time ;
priority priority ;
hellointerval time ;
routerdeadinterval time ;
authkey auth_key ;
```

`defaults`

These parameters specify the `defaults` used when importing OSPF ASE routes into the `gated`(8) routing table and exporting routes from the `gated`(8) routing table into OSPF ASEs.

`preference` *preference*
: The *preference* determines how OSPF routes compete with routes from other protocols in the `gated` routing table. The default value is 150.

`cost` *cost*
: The cost is used when exporting a non-OSPF route from the `gated` routing table into OSPF as an ASE. The default value is 1. You may explicitly override this value in the export policy.

`tag` [`as`] *tag*

OSPF ASE routes have a 32-bit tag field that the OSPF protocol does not use, but which export policy may use to filter routes. When OSPF is interacting with an exterior gateway protocol, the tag field may be used to propagate AS path information; in which case, the `as` keyword is specified because the tag is limited to 12 bits of information. If you omit this parameter, the tag is set to 0.

`type 1 | 2` Routes exported from the `gated` routing table into OSPF default to becoming type 1 ASEs. You may explicitly change this default here and override it in the export policy.

Because of the nature of OSPF, you must limit the rate at which ASEs are flooded. To adjust those rate limits, use the following two parameters:

`exportinterval` *time*

Specifies how often a batch of ASE link-state advertisements are generated and flooded into OSPF. The default is once per second.

`exportlimit` *routes*

Specifies how many ASEs are generated and flooded in each batch. The default is 100.

`traceoptions` *trace_options*

Specifies the tracing options for OSPF. See the Trace Options subsection and the OSPF-specific tracing options.

`monitorauthkey` *authkey*

OSPF state may be queried using the `ospf_monitor` utility. This utility sends nonstandard OSPF packets, which generate a text response from `gated`(8). By default, these requests are not authenticated. If an authentication key is configured, the incoming requests must match the specified authentication key. These packets cannot change an OSPF state, but the act of querying OSPF can use system resources.

`backbone area` | *area*

You must configure each OSPF router into at least one OSPF area. If you configure more than one area, at least one must be the backbone. You can configure the backbone only by using the `backbone` keyword; you cannot specify it as `area 0`. Each area must have at least one interface. The backbone interface may be a `virtuallink`.

`authtype 0 | 1 | none | simple`

OSPF specifies an authentication scheme per area. Each interface in the area must use this same authentication scheme although it may use a different `authenticationkey`. The currently valid values are `none` (`0`) for no authentication, or `simple` (`1`) for simple password authentication.

`stub` [`cost` *cost*]

A `stub` area is one in which there are no ASE routes. If you specify a `cost`, this is used to inject a default route into the area with the specified cost.

`networks` The `networks` list describes the scope of an area. Intra-area LSAs that fall within the specified ranges are not advertised into other areas as inter-area routes. Instead, the specified ranges are advertised as `summary network` LSAs. Inter-area LSAs that do not fall into any range also are advertised as summary network LSAs. This option is very useful on well-designed networks in reducing the amount of routing information propagated between areas.

`stubhosts` This option specifies directly attached hosts that should be advertised as reachable from this router and the costs with which they should be advertised. You should specify point-to-point interfaces on which it is not desirable to run OSPF.

It also is useful to assign an additional address to the loopback interface (one not on the 127 network) and advertise it as a stub host. If this address is the same one used as the router ID, it enables routing to OSPF routers by router ID, rather than by interface address. This is more reliable than routing to one of the routers' interface addresses, which may not always be reachable.

`interface` *interface_list* [`cost` *cost*]
This form of the interface clause is used to configure a `broadcast` (which requires IP multicast support) or a `point-to-point` interface. For the description of *interface_list*, see the section on *interface_list* specification.

Each interface has a `cost`. The costs of all interfaces a packet must cross to reach a destination are summed to get the cost to that destination. The default `cost` is 1, but you may specify another nonzero value.

Interface parameters are common to all types of interfaces:

`retransmitinterval` *time*
The number of seconds between link-state advertisement retransmissions for adjacencies that belong to this interface.

`transitdelay` *time*
The estimated number of seconds required to transmit a link-state update over this interface. `transitdelay` takes into account transmission and propagation delays, and it must be greater than 0.

`priority` *priority*
A number between 0 and 255 that specifies the priority for becoming the designated router on this interface. When two routers attached to a network both try to become designated router, the one that has the highest priority wins. A router that has the router priority set to 0 is ineligible to become designated router.

`hellointerval` *time*
The length of time, in seconds, between Hello packets that the router sends on the interface.

`routerdeadinterval` *time*
: If a neighbor router's Hello packet is not heard for *time* seconds, `gated` declares that the neighbor is down.

`authkey` *auth_key*
: Used by OSPF authentication to generate and verify the authentication field in the OSPF header. You can configure the authentication key on a per interface basis. It is specified by 1 to 8 decimal digits separated by periods, a 1-to-8 byte hexadecimal string preceded by`0x`, or a 1-to-8 character string in double quotation marks.

`interface` *interface_list* `nonbroadcast` [`cost` *cost*]
: This form of the interface clause is used to specify a `nonbroadcast` interface on a nonbroadcast multiaccess (NBMA) media. Because an OSPF `broadcast` media must support IP multicasting, you must configure a broadcast-capable media (such as Ethernet) that does not support IP multicasting as a nonbroadcast interface.

A nonbroadcast interface supports any of the preceding standard `interface` clauses, plus the following two that are specific to nonbroadcast interfaces:

`pollinterval` *time*
: Before an adjacency is established with a neighbor, OSPF packets are sent periodically at the specified `pollinterval`.

`routers`
: By definition, you cannot send broadcast packets to discover OSPF neighbors on a nonbroadcast interface; therefore, you must configure all neighbors. The router list includes one or more neighbors and an indication of their eligibility to become a designated router.

`virtuallink neighborid` *router_id* `transitarea` *area*
: Virtual links are used to establish or increase connectivity of the backbone area. The `neighborid` is the *router_id* of the other end of the virtual link. The transit *area* specified also must be configured on this system. You may specify all standard interface parameters defined by the `interface` clause on a virtual link.

Tracing options

In addition to the following OSPF-specific trace flags, OSPF supports the `all`, `ospf`, and `protocol` flags. The `protocol` flag traces interface and neighbor state machine transitions.

OSPF-specific trace flags are as follows:

`lsabuild`	Link State Advertisement creation
`spf`	Shortest Path First (SPF) calculations
`lsatransmit`	Link State Advertisement (LSA) transmission
`lsareceive`	LSA reception

You may modify the following packet tracing options by using the `update` flag:

`hello` OSPF `HELLO` packets, which are used to determine neighbor reachability.

`dd` OSPF Database Description packets, which are used to synchronize OSPF databases.

`request` OSPF Link State Request packets, which are used to synchronize OSPF databases.

`lsu` OSPF Link State Update packets, which are used to synchronize OSPF databases.

`ack` OSPF Link State Acknowledgment packets, which are used to synchronize OSPF databases.

`redirect` Statement

The syntax for the `redirect` statement follows:

```
redirect yes|no|on|off [ {
      preference preference ;
      interface interface_list ;
      trustedgateways gateway_list ;
} ] ;
```

Controls whether ICMP redirects are listened to. If you omit this statement, the default is to listen to ICMP redirects, unless RIP is enabled and more than one interface exists. When ICMP redirects are disabled, `gated` must actively remove the effects of redirects from the kernel, because the kernel always processes ICMP redirects.

The default preference is 30.

Route Statements

The `static` statements defines the static routes that `gated`(8) uses. A single `static` statement can specify any number of routes. The `static` statements occur after protocol statements and before control statements in the `gated.conf` file. You may specify any number of `static` statements, each containing any number of static route definitions. Routes from other protocols that have better preference values can override these routes.

```
static {
 (host host) | default | (network [ (mask mask) | (masklen number ) ] )
  gateway gateway_list
  [interface interface_list]
  [preference preference]
  [retain]
  [noinstall]
  [static_options] ;
 network [ (mask mask) | (masklen number) ] interface interface
  [preference preference]
  [retain]
  [noinstall]
  [static_options] ;
} ;
```

`host` *host* `gateway` *gateway_list*
network [(`mask` *mask*) | (`masklen` *number*)] `gateway` *gateway_list*
`default gateway` *gateway_list*

This is the most general form of the static statement. It defines a static route through one or more gateways. Static routes are installed when one or more of the `gateways` listed are available on directly attached interfaces. If more than one eligible gateway is available, they are limited by the number of multipath destinations supported (this compile time parameter is currently four).

Parameters for static routes are as follows:

`interface` *interface_list*
When you specify this parameter, gateways are considered valid only when they are on one of these interfaces. For the description of the *interface_list*, see the section on *interface_list* specification.

`preference` *preference*
This option selects the preference of this static route. The preference controls how this route competes with routes from other protocols. The default preference is 60.

`retain` Usually, `gated`(8) removes all routes except interface routes from the kernel forwarding table during a graceful shutdown. You may use the `retain` option to prevent specific static routes from being removed. This is useful to ensure that some routing is available when `gated`(8) is not running.

`noinstall` Typically, the route with the lowest preference is installed in the kernel forwarding table and is the route exported to other protocols. When you specify `noinstall` on a route, it is not eligible to be installed in the kernel forwarding table when it is active, but it is still eligible to be exported to other protocols.

network [(`mask` *mask*) | (`masklen` *number*)] `interface` *interface*
This form defines a static interface route that is used for primitive support of multiple network addresses on one interface. The `preference`, `retain`, and `noinstall` options are the same as described previously.

Static options are as follows:

`admmtu` *number*
Sets the maximum transmission unit (mtu) size for the route to *number*.

`genmask` *mask*
Sets the generation mask for the route to *mask*.

`gid` *grouplist*
Restricts the route that the groups specified in *grouplist* can use.

`netmask` *mask*
Sets the netmask for the route to *mask*. This is an obsolete form of specifying the netmask.

`service` *tos*
Specifies *tos* as the IP type of service.

`tosmatch` Indicates that a client must explicitly request (that is, match) the type of service specified for the route to be able to use it.

The following static options have no effect on the route table. They may be implemented in a future release of the UNICOS operating system.

`hopcount` *number*
Sets the hopcount (number of gateway hops to the destination of) for the route to *number*.

`expire` *number*
Sets the lifetime (in seconds) for the route to *number*.

`lock` Indicates that the next option must be locked against further changes.

`lockrest` Indicates that all remaining options specified for the route must be locked against further changes.

`mtu` *number* Sets the maximum transmission unit size for the route to *number*.

`recvpipe` *number*
Sets the inbound delay-bandwidth product for the route to *number*.

`rtt` *number* Sets the estimated round-trip time for the route to *number*.

`rttvar` *number*
Sets the estimated round-trip time variance for the route to *number*.

`sendpipe` *number*
Sets the outbound delay-bandwidth product for the route to *number*.

`ssthresh` *number*
Sets the outbound gateway buffer limit for the route to *number*.

Control Statements

The `import` and `export` statements control importation of routes from routing protocol peers and exportation of routes to routing protocol peers. In the following `import` and `export` statement formats, the use of the token `all` for *interface_list* is redundant; therefore, it is not allowed.

The `import` statement can have the following syntax:

```
import proto rip|redirect restrict ;

import proto rip|redirect
   [preference preference] {
   import_list
} ;

import proto rip|redirect interface interface_list restrict ;

import proto rip|redirect interface interface_list
   [preference preference] {
   import_list
} ;

import proto rip|redirect gateway gateway_list restrict ;

import proto rip|redirect gateway gateway_list
   [preference preference] {
   import_list
} ;

import proto ospfase [tag ospf_tag] restrict ;

import proto ospfase [tag ospf_tag]
   [preference preference] [{
   import_list
}] ;
```

If you specify an *ospf_tag* specification, only routes matching that tag specification are considered; otherwise, any tag is considered. An OSPF tag specification may be a decimal, hexadecimal, or a dotted quad number.

If you specify more than one import statement relevant to a protocol, they are processed most specific to least specific (for example, for RIP, gateway, interface, and protocol), then in the order specified in the configuration file.

The import statement `restrict` parameter causes routes learned by the import statement to be ignored. The `preference` parameter specifies the preference of routes learned from this import statement.

The following is the format of an *import_list*:

dest_mask `[[restrict] | [preference` *preference*`]] ;`

An *import_list* consists of zero or more destinations (with optional mask). You may specify one of two parameters: `restrict` to prevent a set of destinations from being imported, or a specific `preference` for this set of destinations.

The contents of an *import_list* are sorted internally so that entries that have the most specific masks are examined first. The order in which *dest_mask* entries are specified does not matter.

If you specify an import list, the import list is scanned for a match. If no match is found, the route is discarded. An `all restrict` entry is assumed in an import list.

The `export` statement can have the following formats:

```
export proto rip restrict ;

export proto rip [metric metric] {
   export_list
} ;

export proto rip interface interface_list restrict ;

export proto rip interface interface_list
   [metric metric] {
   export_list
} ;

export proto rip gateway gateway_list restrict ;

export proto rip gateway gateway_list
   [metric metric] {
   export_list
} ;

export proto ospfase [type 1|2] [tag ospf_tag] restrict ;

export proto ospfase [type 1|2] [tag ospf_tag]
   [cost ospf_cost] {
   export_list
} ;
```

The export statement distributes routes to a destination protocol, gateway, or interface. The `restrict` parameter prevents the routes specified by the export statement from being exported.

The export list specifies the source of the routes that are distributed by the export statement. The format of an export list follows:

```
proto rip|direct|static
   [restrict] | [metric metric] [ {
   announce_list
} ] ;

proto rip|direct|static interface interface_list
   [restrict] | [metric metric] [ {
   announce_list
} ] ;

proto rip gateway gateway_list
   [restrict] | [metric metric] [ {
   announce_list
} ] ;

proto ospf [restrict] | [metric metric] [ {
   announce_list ;
} ] ;

proto ospfase [restrict | metric metric]] [ {
   announce_list ;
} ] ;

proto proto tag tag
   [restrict] | [metric metric] [ {
   announce_list
} ] ;
```

If you specify a tag, only routes with that tag will be considered; otherwise, any tag will be considered. An OSPF tag on an export statement may be a decimal or hexadecimal. An OSPF tag on an export list is a 31-bit number that is matched against the tag present (if any) on that route.

If you specify more than one export statement relevant to a protocol, they are processed most specific to least specific (for example, for RIP, gateway, interface, and protocol), then in the order specified in the configuration file.

By default, interface routes are exported to all protocols. RIP also exports its own routes. An export specification that has only a `restrict` prevents these defaults from being exported. You cannot change the metric RIP uses for its own routes; if you try to override this metric, it is silently ignored.

You may specify any protocol for import lists that referr to AS paths and tags. Tags are currently meaningful only for OSPF ASE routes.

An *announce_list* consists of zero or more destinations (with optional mask). You may specify one of two parameters: `restrict` to prevent a set of destinations from being exported, or a specific metric for this set of destinations.

dest_mask [[`restrict`] | [`metric` *metric*]] ;
...

The contents of an *announce_list* are sorted internally so that entries that have the most specific masks are examined first. The order in which *dest_mask* entries are specified does not matter.

If you omit *announce_list*, all destinations are announced. If you specify an announce list, an `all restrict` is assumed. Therefore, an empty announce list is the equivalent of `all restrict`.

To announce routes that specify a next hop of the loopback interface (for example, static routes) through RIP, you must specify the metric at some level in the export clause; setting a default metric for RIP is not sufficient.

FILES

`/etc/gated.conf`

SEE ALSO

`netstat`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`setlogmask`(3C) (see `syslog`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`arp`(8), `gated`(8), `ifconfig`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

Routing Information Protocol, RFC 1058

Routing Information Protocol version 2, RFC 1388

Open Shortest Path First Protocol version 2, RFC 1583

COPYRIGHT INFORMATION

NAME

gettydefs – Speed and terminal settings used by getty

IMPLEMENTATION

CRAY Y-MP systems

DESCRIPTION

The /etc/gettydefs file contains information that getty(8) uses to set up the speed and terminal settings for a line. This method of terminal handling is used for IOS terminals.

The information in the gettydefs file also specifies the appearance of the login prompt (usually login: by default).

Each entry in the /etc/gettydefs file has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form \b, \n, or \c, as well as *nnn*; *nnn* is the octal value of the desired character. The various fields are as follows:

label String against which getty(8) tries to match its second argument. This is often the baud rate at which the terminal is supposed to run (for example, 1200) but it need not be (see the definition of *next-label*). Speed settings have no effect on IOS terminals.

initial-flags Initial ioctl(2) settings to which the terminal will be set if a terminal type is not specified to getty(8). These flags are the same as those in the sys/termio.h include file. Usually, only the speed flag is required in *initial-flags*. The getty(8) program automatically sets the terminal to raw input mode and handles most of the other flags. The *initial-flag* settings remain in effect until getty(8) executes login(1).

final-flags These flags accept the same values as the *initial-flags* and are set just before getty(8) executes login(1). The speed flag is again required. The SANE composite flag handles most of the other flags that must be set so that the processor and terminal are communicating according to the same protocol. Two commonly specified *final-flags* are TAB3, which send tabs to the terminal as spaces, and HUPCL, which hangs up the line on the final close.

login-prompt This entire field is printed as the login prompt. Unlike the preceding fields, in which white space (a space, tab, or newline character) is ignored, it is included in the login prompt field.

next-label If this entry does not specify the desired speed, indicated by the typing of a break character, getty(8) searches for the entry with *next-label* as its *label* field and sets up the terminal for those settings. Usually, a series of speeds is linked in this fashion to form a closed set; for example, 2400 is linked to 1200, which in turn is linked to 300, which finally is linked to 2400.

If getty(8) is called without a second argument, the first entry of gettydefs is used, thus making the first entry of gettydefs the default entry. It also is used if getty(8) cannot find the specified *label*. If the gettydefs file itself is missing, one entry is built into getty(8) that will bring up a terminal (at 9600 Bd).

After you create or modify a gettydefs file, run it through getty(8) by using the -c (check) option to ensure that no errors exist.

FILES

/etc/gettydefs	File of terminal information used by getty(8)
/usr/include/sys/termio.h	Structure used by ioctl(2) system calls to terminal devices

SEE ALSO

termio(4)

login(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

ioctl(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

getty(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

group – Format of the group-information file

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/group file contains the following information for each user group:

- Group name
- Encrypted password
- Numeric group ID (GID)
- Comma-separated list of user names allowed in the group

The group file is an ASCII file. The fields are separated by colons; each group is separated from the next by a newline character. udbgen(8) automatically maintains this file to match the information in the udb(5) file. The password field is present for compatibility, but it is always set to *.

This file resides in the /etc directory and has general read permission so that it can be used, for example, to map numeric group IDs to names.

NOTES

The encrypted password field is available under the UNICOS operating system, but Cray Research does not support it.

The list of user names can become very long for groups shared by many users. To keep the line length reasonable, udbgen(8) generates the group file that has a maximum membership list of about 400 characters. If the group list exceeds this length, additional lines are created to hold the remainder of the list. The additional lines will be adjacent and will begin with the identical group name and group ID. For example, if the group list for group gr1 with GID 123 were long enough to occupy three lines, that fragment of the group file would appear as follows:

```
gr1:*:123:usr1,usr2,usr3,usr4,usr6,usr10
gr1:*:123:usr101,usr102,usr103,usr104,usr105
gr1:*:123:usr563,usr570
```

The first two lines would have a group list that consists of about 400 characters (the example shows a short list for brevity) and the final line would consist of the remainder of the list. The 400-character limit is approximate because the line is broken at the end of the name that causes the length to exceed 400 characters.

Unlike the /etc/passwd file, you must update the /etc/group file manually to include new group IDs and group names. When you update /etc/group, ensure that the udbgen(8) utility is not running, because udbgen would overwrite any changes to /etc/group.

FILES

`/etc/group` File that contains user group information

SEE ALSO

`acid`(5), `passwd`(5), `udb`(5)

`udbsee`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`getgrent`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`udbgen`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

hosts, hosts.bin – Contains network host name database

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/hosts file contains the database of all locally known hosts on the TCP/IP network. The /etc/hosts.bin file is the binary version of the hosts file; the mkbinhost(8) command creates it.

For each host, one line should contain the network type (optional), the host's Internet address, the official host name, and any aliases that exist for the host name. The recognized value for the network type field is inet (the default). Items are separated by any number of blanks and/or tab characters. A # symbol indicates the beginning of a comment; when you use the # symbol, the routines that search the file ignore additional characters up to the end of the line. The hosts file is searched sequentially; therefore, if you specify more than one host name with a given Internet address, the first entry is used and all others are ignored. Specify Internet network addresses in the conventional "." (dot) notation, using the inet_addr routine from the Internet address manipulation library, inet(3C).

Host names can contain any printable character other than a blank, tab, new line, or comment (#).

NOTES

All library routines in gethost(3C) check for the existence of the /etc/hosts.usenamed file. If it exists, they use the domain name service (see resolver(3C)) to perform host name and address lookups; otherwise, they use hosts.bin if it exists. If it does not exist, the library routines get information from hosts. When /etc/hosts is modified, you should run mkbinhost to update /etc/hosts.bin.

Avoid using both uppercase and lowercase letters in hosts names, because some implementations of TCP/IP cannot handle mixed-case host names.

EXAMPLES

The following is an example of entries in an /etc/hosts file:

```
#
#     HYPERchannel addresses
#
84.0.0xc4.5 sn101      sn101-inet
84.0.0x13.0 nobel      nobel-inet
#
#     Ethernet addresses
#
#192.9.1    nobelnet
192.9.1.17  nobel mailhost
192.9.1.18  ranger
192.9.1.19  lps
192.9.1.20  sol
```

FILES

/etc/hosts	File that contains names of known hosts on TCP/IP network.
/etc/hosts.bin	Binary version of /etc/hosts file.
/etc/hosts.usenamed	The existence of this file turns on domain name service (named) lookup for host names and addresses.

SEE ALSO

gethost(3C), gethostinfo(3C), inet(3C), resolver(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

mkbinhost(8), named(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

hosts.equiv – Contains public information for validating remote autologin

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/hosts.equiv and .rhosts files provide the remote authentication database for rlogin(1B), remsh(1B), rcp(1), and rcmd(3C). If the Network Queuing System (NQS) is using file validation, it also will use the /etc/hosts.equiv file. If a .nqshosts file does not exist, NQS will use the .rhosts file. The files specify remote hosts and users that are considered trusted. Trusted users are allowed to access the local system without supplying a password. The ruserok() library routine (see rcmd(3C)) performs the authentication procedure for programs by using the /etc/hosts.equiv and .rhosts files. The /etc/hosts.equiv file applies to the entire system, but individual users can maintain their own .rhosts files in their home directories.

These files bypass the standard password-based user authentication mechanism. To maintain system security, you must take care when creating and maintaining these files.

The remote authentication procedure determines whether a remote user from a remote host should be allowed to access the local system as a (possibly different) local user. This procedure first checks the /etc/hosts.equiv file and then checks the .rhosts file in the home directory of the local user for whom access is being tried. Entries in these files can be positive entries, which explicitly allow access, and negative entries, which explicitly deny access. The authentication succeeds as soon as a matching positive entry is found. The procedure fails when a matching negative entry is found or if no matching entry is found in either file. The order of entries, therefore, can be important; if the file contains both matching positive and negative entries, the entry that appears first will prevail. If the remote authentication procedure fails, the remsh(1B) and rcp(1) programs fail, but the rlogin(1B) command falls back to the standard password-based login procedure.

Both the /etc/hosts.equiv and .rhosts files are formatted as a list of one-line entries. Each entry has the following form:

hostname [*username*]

If the following form is used, users from the host *hostname* are trusted; that is, they may access the system by using the same user name as they have on the remote system.

hostname

You may use this form in both the /etc/hosts.equiv and .rhosts files.

If the line is in the following form, the user *username* from the host *hostname* can access the system:

hostname username

You may use this form in individual .rhosts files to allow remote users to access the system as a different local user. If this form is used in the /etc/hosts.equiv file, the user *username* is allowed to access the system as any local user.

Negative entries disallow access and are preceded by a – symbol. The following form disallows access by the user *username* only from the host *hostname*:

hostname –username

The following form disallows all access from the host *hostname*:

–hostname

To match all users or all hosts, use an * symbol. For example, entering * allows any user from any host to log in under the same user name. Entering * *username* allows the user *username* access from any remote host. Entering *hostname* * in a `.rhosts` file allows any user from the remote host *hostname* to access the system as the user in whose `.rhosts` file the entry appeared.

You should use positive entries in `/etc/hosts.equiv` that include a *username* field with extreme caution. Because `/etc/hosts.equiv` applies systemwide, these entries can allow one, or a group of, remote users access to the system as any local user. This can be a security problem.

NOTES

To authenticate the user, the system configuration can require an entry in both the user's `.rhost` and `/etc/hosts.equiv` files, and it also may require the remote user name to match the local user name.

FILES

`/etc/hosts.equiv`	File that contains name(s) for a remote host
`/etc/udb`	File that contains remote user names

SEE ALSO

`rhosts`(5)

`rcp`(1), `remsh`(1B), `rlogin`(1B) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`rcmd`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`rlogind`(8), `rshd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

TCP/IP Network User's Guide, Cray Research publication SG–2009

UNICOS Networking Facilities Administrator's Guide, Cray Research publication SG–2304

UNICOS NQS and NQE Administrator's Guide, Cray Research publication SG–2305

NAME

inetd.conf – Internet super-server configuration file

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/inetd.conf file contains the configuration information used by the Internet super-server configuration file, which listens for incoming service requests. The inetd(8) command is invoked at boot time.

Upon execution, inetd reads its configuration information from a configuration file which, by default, is /etc/inted.conf. There must be an entry for each field of the configuration file, with entries for each field separated by a tab or a space. Comments are denoted by a "#" at the beginning of a line.

The fields of the configuration file are as follows:

service name There are several types of services that inetd can start: standard, TCPMUX, and RPC. The *service name* field consists of a valid service in the /etc/services file or a port number on which the inetd daemon can listen for incoming requests. For internal services, the server name must be the official name of the service (the first entry in /etc/services). For TCPMUX services, the value of the *service name* field consists of the string *tcpmux* followed by a slash and the locally-chosen service name. The service names listed in /etc/services and the name *help* are reserved. Try to choose unique names for your TCPMUX services by prefixing them with your organization's name and suffixing them with a version number. For RPC services, *service name* consists of the RPC *service name* followed by a slash and either a version number or a range of version numbers (e.g., *rstat/2-4*).

socket type Should be one of the following values:

dgram	Indicates that the socket is a datagram
raw	Indicates that the socket is raw
rdm	(Not implemented) Indicates that the socket is a reliably delivered message
seqpacket	(Not implemented) Indicates that the socket is a sequenced packet socket
stream	Indicates that the socket is a stream. TCPMUX services must use stream.

protocol The valid protocol as given in the /etc/protocols file. Protocol is usually tcp or udp. TCPMUX services must use *tcp*. For RPC services, *protocol* consists of the string rpc followed by a slash and the name of the protocol. For example, rpc/tcp indicates that an RPC server is using the TCP protocol as its transport mechanism.

`wait|nowait` This entry is applicable to datagram sockets only (other sockets should have a `nowait` entry in this space). If a datagram server connects to its peer and thus frees the socket so that `inetd`(8) can receive further messages on the socket, it is a multithreaded server, and it should use the `nowait` entry.

For datagram servers that process all incoming datagrams on a socket and eventually time out, the server is single-threaded and should use a `wait` entry. `talk` is an example of the latter type of datagram server.

The `tftpd` server is an exception; it is a datagram server that establishes pseudo-connections. To avoid a race, it must be listed as `wait`; the server reads the first packet, creates a new socket, and then forks and exits to allow `inetd` to check for new service requests to spawn new servers.

`TCPMUX` services must use `nowait`.

user User name of the user as whom the server should run. By associating a user with the daemon, servers can be given less permission than root. The `ftp`, `telnet`, `shell`, and `login` servers need root permission; the `finger` and `tftp` servers should be run as a user with limited capability.

server program Path name of the program that `inetd` will execute when a request is found on its socket. If `inetd` provides this service internally, this entry should be `internal`.

server program arguments
Arguments to the `exec`(2) system call, starting with *argv*[0], which is the name of the program.

CAUTIONS

For security reasons, you should use `fingerd`(8) as a user with limited priority and disable `tftpd`(8).

TCPMUX

RFC 1078 describes the `TCPMUX` protocol: "A `TCP` client connects to a foreign host on TCP port 1. It sends the service name followed by a carriage-return line-feed <CRLF>. The service name is never case sensitive. The server replies with a single character indicating positive (+) or negative(-) acknowledgement, immediately followed by an optional message of explanation, terminated with a <CRLF>. If the reply was positive, the selected protocol begins; otherwise the connection is closed." The program is passed the `TCP` connection as file descriptors 0 and 1.

If the `TCPMUX` service name begins with a "+", `inetd` returns the positive reply for the program. This allows you to invoke programs that use `stdin/stdout` without putting any special server code in them.

The special service name *help* causes `inetd` to list `TCPMUX` services in `inetd.conf`.

EXAMPLES

Following is a sample `inetd.conf` file. In this example, the services `uucp`, `tftp`, `comsat`, `talk`, and `ntalk` are not needed and have been commented out. Each service is listed with its associated socket type and protocol; use this example as a reference to socket types and protocols.

```
#
# Internet server configuration database
#
ftp      stream  tcp  nowait  root    /etc/ftpd  ftpd
telnet   stream  tcp  nowait  root    /etc/telnetd  telnetd
shell    stream  tcp  nowait  root    /etc/rshd  rshd
login    stream  tcp  nowait  root    /etc/rlogind  rlogind
exec     stream  tcp  nowait  root    /etc/rexecd  rexecd
# Run as user "uucp" if you don't want uucpd's wtmp entries.
#uucp    stream  tcp  nowait  root    /etc/uucpd  uucpd
finger   stream  tcp  nowait  nobody  /etc/fingerd  fingerd
#tftp    dgram   udp  wait    tftp    /etc/tftpd  tftpd
#comsat  dgram   udp  wait    root    /etc/comsat  comsat
#talk    dgram   udp  wait    root    /etc/talkd  talkd
#ntalk   dgram   udp  wait    root    /etc/ntalkd  ntalkd
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
tcpmux   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
#
# TCPMUX service syntax:
#
tcpmux/+date  stream tcp nowait guest /bin/date date
tcpmux/phonebook stream tcp nowait guest /usr/local/bin/phonebook phonebook
#
#
# RPC services syntax:
# <rpc_prog>/<vers> <socket_type> rpc/<proto> <flags> <user> <pathname> <args>
ypupdated/1   stream  rpc/tcp wait root /etc/ypupdated ypupdated
rstatd/2-4    dgram   rpc/udp wait root /etc/rstatd    rstatd
rusersd/1-2   dgram   rpc/udp wait root /etc/rusersd   rusersd
sprayd/1      dgram   rpc/udp wait root /etc/sprayd    sprayd
rwalld/1      dgram   rpc/udp wait root /etc/walld     rwalld
```

FILES

`/etc/inetd.conf`	Contains configuration information used by the Internet super-server configuration file
`/etc/protocols`	Lists the valid protocols
`/etc/services`	Lists valid services

SEE ALSO

`protocols`(5), `services`(5)

`fingerd`(8), `inetd`(8), `tftpd`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

RFC 1078: *TCP Port Service Multiplexer (TCPMUX)*

NAME

infoblk – Loader information table

SYNOPSIS

```
#include <infoblk.h>
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The segldr(1) and ld(1) commands build the infoblk loader information table into all normal executable programs. The contents of _infoblk provide information about the time and date of program creation, size and structure of the program's memory usage, and so on. To access the table contents within the program, use the C global structure name _infoblk, as follows:

```
extern struct infoblk _infoblk;

/*
 *  the infoblk structure, which SEGLDR puts in every binary
 *  (referenced by _infoblk)
 */

struct infoblk {
  unsigned i_vers: 7;       /* version of _infoblk table           */
  unsigned : 24;            /* unused                              */
  unsigned i_a: 1;          /* fill address generation flag        */
  unsigned i_len: 32;       /* no. of words in infoblk             */

  char i_name[8];           /* table name  - "infoblk"             */
  long i_cksum;             /* table checksum                      */
  char i_date[8];           /* date of program creation            */
  char i_time[8];           /* time of program creation            */
  char i_pid[8];            /* name of generating program          */
  char i_pvr[8];            /* version of generating program       */
  char i_osvr[8];           /* O.S. version at generation time     */
  long i_udt;               /* creation timestamp                  */
  long i_fill;              /* value to fill uninitialized areas   */

  unsigned i_tbase: 32;     /* text area base address              */
  unsigned i_dbase: 32;     /* data area base address              */

  unsigned i_tlen: 32;      /* text section length                 */
  unsigned i_dlen: 32;      /* data section length                 */
```

```
    unsigned i_blen: 32;      /* bss section length                   */
    unsigned i_zlen: 32;      /* zeroset section length               */

    unsigned i_cdatalen: 32;  /* data section length before expansion */
    unsigned i_lmlen: 32;     /* CRAY-2 local memory length           */

    unsigned i_amlen: 32;     /* auxiliary memory length              */
    unsigned i_mbase: 32;     /* base address of heap                 */

    unsigned i_hinit: 32;     /* initial heap size                    */
    unsigned i_hinc: 32;      /* heap increment                       */

    unsigned i_sinit: 32;     /* initial stack size                   */
    unsigned i_sinc: 32;      /* stack increment                      */

    unsigned i_usxf: 32;      /* USX table first address              */
    unsigned i_usxl: 32;      /* USX table last address               */

    unsigned i_mtptr: 32;     /* machine targeting block address      */
    unsigned i_cmptr: 32;     /* expansion entry list address         */

    unsigned i_enlen: 32;     /* saved arg/env length                 */
    unsigned i_enptr: 32;     /* environment pointer (currently 0)    */

    unsigned i_sgptr: 32;     /* pointer to $SEGRES table             */
    unsigned : 32;            /* unused                               */

    unsigned i_taskstk: 32;   /* slave task initial stack size        */
    unsigned i_taskincr: 32;  /* slave task increment value           */

    long i_user1;             /* user value word 1                    */
    long i_user2;             /* user value word 2                    */
};
```

FILES

`/usr/include/infoblk.h` Loader information table include file

SEE ALSO

`ld`(1), `segldr`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

inittab – Script for init(8) process

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/inittab file is the script for init(8), the general process spawner. Processes dispatched by init(8) are, typically, the daemons required for the multiuser run levels and the shell for the UNICOS system console.

The inittab file is composed of position-dependent entries that have the following format:

id : *rstate* : *action* : *process*

A newline character delimits each entry; however, a backslash (\) preceding a newline character indicates a continuation of the entry. Up to 512 characters for each entry are permitted. You may insert comments in the *process* field, using the sh(1) convention for comments. No limits (other than the maximum entry size) are imposed on the number of entries in the inittab file. The entry fields are as follows:

id One to four characters used to identify an entry uniquely.

rstate Run level in which this entry will be processed. A run level is a configuration of the system; each run level allows only a selected group of processes to exist. Each process that init(8) spawns is assigned one or more run levels in which it is allowed to exist. Multiuser mode consists of seven run levels. The run levels are represented by a number that ranges from 0 through 6. For example, if the system is in run-level 1, only entries that have a 1 in the *rstate* field are processed.

When init(8) is requested to change run levels, all processes that do not have an entry in the *rstate* field for the target run level are sent a warning signal (SIGTERM) and allowed 20 seconds before being forcibly terminated by a kill signal (SIGKILL).

The *rstate* field can define multiple run levels for a process by selecting more than one run level in any combination from 0 through 6. If you do not specify a run level, *action* is taken on this process for all run levels 0 through 6.

Three other values (a, b, and c) can appear in the *rstate* field, even though they are not true run levels. Entries that have these values in the *rstate* field are processed only when the telinit(8) process (see init(8)) requests that they be run (regardless of the current run level of the system). They differ from run levels in that the system is in these states only for as long as it takes to execute all entries associated with the states. A process that an a, b, or c command starts is not killed when init(8) changes levels. It is killed only if its line in /etc/inittab is marked off in the *action* field, its line is deleted entirely from /etc/inittab, or init(8) goes into the single-user state.

action Keywords in this field specify how to treat the process in the *process* field. `init`(8) recognizes the following actions:

`boot` The `init`(8) command processes the entry only when reading the `inittab` file at boot time; `init`(8) starts the process and does not wait for its termination. When the process dies, `init`(8) does not restart it. For this instruction to be meaningful, *rstate* should be either the default or a match of the run level of `init`(8) at boot time. This action is useful for an initialization function that follows a hardware reboot of the system.

`bootwait` The `init`(8) command processes the entry only when reading the `inittab` file at boot time; `init`(8) starts the process and waits for its termination. When the process dies, `init`(8) does not restart it.

`generic` When a privileged daemon process initiates a new login session, it sends a request to `init`(8) through the `/etc/initreq` pipe (FIFO special file). This request includes the terminal to be used, the associated remote host, and the generic ID specified in the *id* field. The `init`(8) command verifies that `inittab` contains a line with the specified *id* field and that the *rstate* field includes the current run level. Then `init`(8) starts a login process on the specified terminal.

`initdefault` The `init`(8) command scans an entry with this action only when it is initially invoked; `init`(8) uses this entry, if it exists, to determine which run level to enter initially. It does this by taking the highest run level specified in the *rstate* field and using that as its initial state. If the *rstate* field is empty, the run level is interpreted as 0123456; `init`(8) enters run level 6. You can use the `initdefault` entry to specify that `init`(8) should start in the single-user state. If `init`(8) does not find an `initdefault` entry in `/etc/inittab`, it also requests an initial run level from the `/dev/syscon` terminal at reboot time.

`ldsynctm` Sets the `init ldsynctm` variable, which determines the system `ldsync` interval. The *process* field for this entry is specified in seconds. For details, see `ldsync`(8).

`off` When the process associated with this entry is currently running, `init`(8) sends the warning signal (`SIGTERM`) and waits 20 seconds before forcibly terminating the process by using the kill signal (`SIGKILL`). When the process is nonexistent, `init`(8) ignores the entry.

`once` On entering a run level that matches the *rstate* for the entry, `init`(8) starts the process and does not wait for its termination. When the process dies, `init`(8) does not restart it. If, on entering a new run level, the process is still running from a previous run-level change, the program will not be restarted.

`ondemand` This instruction is really a synonym for the `respawn` action. It is functionally identical to `respawn`, but it is given a different keyword to divorce it from run levels. This is used only with the `a`, `b`, or `c` values discussed in the *rstate* field description.

`respawn` If the process does not exist, `init`(8) will start the process; it will not wait for process termination (that is, it will continue to scan the `inittab` file). When the process dies, `init`(8) restarts it. If the process currently exists, `init`(8) will do nothing and will continue to scan the `inittab` file.

`sleeptime` Sets the `init sleeptime` variable, which determines the system `sync` interval. The *process* field for this entry is specified in seconds. For details, see `sync`(1).

`sysinit` The `init`(8) command executes entries of this type before trying to access the console. You should use this entry to initialize only the devices for which `init`(8) might ask for a run level; `init`(8) executes and waits for these entries before continuing.

`timezone` Sets the systemwide local time zone. The contents of the *process* field are used to set the `TZ` environment variable. For a definition of the format for `TZ`, see `ctime`(3C). The `timezone` entry should follow the `initdefault` entry.

`wait` On entering the run level that matches the *rstate* of the entry, `init`(8) starts the process and waits for its termination. While `init`(8) is in the same run level, all subsequent reads of the `inittab` file cause `init`(8) to ignore this entry.

process An entry in this field is a `sh`(1) command to be executed. The `init`(8) command prefixes the entire *process* field with the `exec`(2) string and passes it to a forked `sh` process as `sh -c 'exec` *command*`'`. Therefore, any legal `sh` syntax can appear in the *process* field. You can insert comments with the `#`*comment* syntax.

EXAMPLES

The following is an example of an `inittab` entry for Minnesota in the central time zone in the U.S.A.:

```
tz::timezone:TZ=CST6CDT
```

The `init`(8) command passes this value to its children, they pass it to theirs, and so on, so that all processes interpret the time according to this `inittab` entry.

The `init sleeptime` and `ldsynctm` variables default to 30 seconds and 120 seconds, respectively. To reset these variables, create `inittab` entries such as the following:

```
st::sleeptime:60
lt::ldsynctm:180
```

The preceding `init`(8) commands reset the `sleeptime` and `ldsynctm` variables to 60 seconds and 180 seconds, respectively.

FILES

`/etc/initreq`	Pipe used when initiating a new login session
`/etc/inittab`	Script for `init`(8) process
`/usr/include/initreq.h`	Definition of request structure

SEE ALSO

`sh`(1), `sync`(1), `who`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

`exec`(2), `open`(2), `signal`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`ctime`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

`getty`(8), `init`(8), `ldsync`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

General UNICOS System Administration, Cray Research publication SG–2301

NAME

inode – Inode format

SYNOPSIS

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ino.h>
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

An inode for a regular file or a directory in a file system has a structure defined by the sys/ino.h include file.

The structure of an inode for NC1FS file systems is as follows:

```
struct cdinode {
        uint    cdi_rsrvd_1 : 8, /* Reserved for expansion of cdi_mode       */
                cdi_mode    :24, /* mode and type of file (4-bits still free)*/
                cdi_msref   : 1, /* Modification signature is referenced flag*/
                cdi_ms      :14, /* Modification signature                   */
                cdi_nlink   :17; /* #of links to file (can hold > 100,000)   */

        uint    cdi_rsrvd_2 : 8, /* Reserved for expansion of cdi_uid        */
                cdi_uid     :24, /* Owner's user-ID                          */
                cdi_rsrvd_3 : 8, /* Reserved for expansion of cdi_gid        */
                cdi_gid     :24; /* Owner's group-ID                         */

        uint    cdi_rsrvd_4 : 8, /* Reserved for expansion of cdi_acid       */
                cdi_acid    :24, /* Account-ID                               */
                cdi_gen     :32; /* Inode generation number                  */

        long    cdi_size;        /* Number of bytes in the file              */
        long    cdi_moffset;     /* Modification offset for current signature*/

        uint    cdi_blocks  :52, /* Quotas: #of blocks actually allocated    */
                cdi_extcomp : 1, /* Security: extended compartments flag     */
                cdi_secrsvd1:11; /* Security: reserved                       */
```

```
union {
        long smallcmps;  /* Compartments if [0..63]                  */
} cdi_compart;           /* Security: compartments info              */

uint    cdi_slevel  : 8, /* Security: security level                 */
        cdi_intcls  : 8, /* Security: integrity class (obsolete)     */
        cdi_secflg  :16, /* Security: flag settings                  */
        cdi_intcat  :32; /* Security: integrity category (obsolete)  */

long    cdi_permits;     /* Security: Permissions inherited at       */
                         /* execution time.                          */

union {
        daddr_t daddr;   /* Extent descriptor                        */
        dblk_t  dblk;    /* Block  descriptor                        */
} cdi_acl;               /* Security: ACL location                   */

uint    cdi_cpart   : 8, /* Next partition from cbits to use         */
        cdi_rsrvd_5 : 8, /* Reserved by the Kernel group.            */
        cdi_dmkey   :48; /* Data-Migration: key                      */

uint    cdi_allocf  : 4, /* Data-Block allocation flags              */
        cdi_alloc   : 4, /* Data-Block allocation technique          */
        cdi_cblks   :24, /* Number of blocks to allocate per part    */
        cdi_dmmid   :32; /* Data-Migration: machine-ID               */

uint    cdi_atmsec  :34, /* Access time (secs)                       */
        cdi_natmsec :30; /* Access time (nanosecs)                   */

uint    cdi_mtmsec  :34, /* Modification time (secs)                 */
        cdi_nmtmsec :30; /* Modification time (nanosecs)             */

uint    cdi_ctmsec  :34, /* Time of last inode modification (secs)    */
        cdi_nctmsec :30; /* Time of last inode modification (nanosecs)*/

long    cdi_cbits;       /* bit mask, file placement within cluster  */
```

```
        union {
                daddr_t daddr;   /* Extent descriptor                        */
                dblk_t  dblk;    /* Block  descriptor                        */
                long    whole;                  struct {
                        uint     one   :32,       /* half 1                 */
                                 two   :32;       /* half 2                 */
                } half;                   struct {
                        uint     one   :16,       /* quarter 1              */
                                 two   :16,       /* quarter 2              */
                                 three :16,       /* quarter 3              */
                                 four  :16;       /* quarter 4              */
                } quarter;                  struct {
                         uint    one   : 8,       /* eighth 1               */
                                 two   : 8,       /* eighth 2               */
                                 three : 8,       /* eighth 3               */
                                 four  : 8,       /* eighth 4               */
                                 five  : 8,       /* eighth 5               */
                                 six   : 8,       /* eighth 6               */
                                 seven : 8,       /* eighth 7               */
                                 eight : 8;       /* eighth 8               */
                } eighth;
        } cdi_addr[8];           /* File allocation locators                 */
                                 /* The #define for NC1NADDR must not be > 8 */

        long    cdi_rsrvd[5];    /* Reserved by the Kernel group for use in  */
                                 /* future releases of UNICOS.               */
                                 /* No notification will be given when these */
                                 /* words will be employed by future versions*/
                                 /* of UNICOS.                               */

        long    cdi_slock[2];    /* Reserved for SFS lock structure          */

        long    cdi_sitebits;    /* Word reserved for site use.              */
};
```

FILES

/usr/include/sys/ino.h	Inode structure definition
/usr/include/sys/types.h	Data types definition file

SEE ALSO

`fs`(5), `types`(5)

`stat`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

NAME

ipc – Interprocess communication (IPC) access structure

SYNOPSIS

```
#include <sys/ipc.h>
```

IMPLEMENTATION

All Cray Research systems

STANDARDS

POSIX, XPG4

DESCRIPTION

Three mechanisms use the sys/ipc.h include file for interprocess communication (IPC): messages, semaphores, and shared memory. All use a common structure type, ipc_perm, to pass information used in determining permission to perform an IPC operation.

The ipc_perm structure contains the following members:

uid_t	uid	Owner's user ID
gid_t	gid	Owner's group ID
uid_t	cuid	Creator's user ID
gid_t	cgid	Creator's group ID
mode_t	mode	Read/write permission

The uid_t, gid_t, mode_t, and key_t types are defined as described in sys/types.h.

Definitions are given for the following constants:

Mode bits:

IPC_CREAT	Creates entry if key does not exist.
IPC_EXCL	Fails if key exists.
IPC_NOWAIT	Returns an error if request must wait.

Keys:

IPC_PRIVATE	Specifies a private key.

Control commands:

IPC_GETACL	Gets access control list.
IPC_RMID	Removes identifier.
IPC_SET	Sets options.

IPC_SETACL	Sets access control list.
IPC_SETLABEL	Sets security label.
IPC_STAT	Gets options.

SEE ALSO

msg(5), sem(5), shm(5), types(5)

ipcs(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

msgctl(2), semctl(2), shmctl(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

ipc(7) Online only

NAME

iptos – IP Type-of-Service database

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/iptos file contains the database of Type-of-Service (TOS) names used for the Internet Protocol (IP) TOS option.

Each entry must consist of a single line that contains the name of the TOS entry, the protocol name for which the entry is appropriate, the TOS value for the entry, and any aliases that exist for the entry. Items are separated by any number of blanks and/or tab characters. A # symbol indicates that the remaining portion of the line is a comment and is not interpreted by routines that search the file. Blank lines in the file are ignored.

TOS entry names may contain any printable character other than a blank, tab, new line, or comment (#).

A protocol name of * (one asterisk) indicates that the entry is valid for all protocols.

The TOS value for the entry must be a list of either symbolic names or numbers (octal, decimal, or hexadecimal) that correspond to TOS option bits or TOS precedence values, separated by | symbols. Recognized symbolic names for TOS option bits are as follows:

```
none                    0x00
delay                   0x10
throughput              0x08
reliability             0x04
reserved1               0x02
reserved2               0x01
```

Recognized symbolic names for TOS precedence values are as follows:

```
netcontrol              0xe0
internetcontrol 0xc0
crtic/ecp               0xa0
flashoverride   0x80
flash                   0x60
immediate               0x40
priority                0x20
routine         0x00
```

EXAMPLES

The following example shows typical entries in `/etc/iptos`:

```
#
# Format of this file:
# Application     Proto TOS-bits    aliases
#
# The Proto field may be "*" mean it doesn't matter.
#
# For multiple values, use a "|", e.g, delay|throughput
#

delay             *      delay           lowdelay
reliability       *      reliability     highreliability
throughput        *      throughput      highthroughput

data              tcp    throughput      bulk-data batch rcp
data              udp    delay           bulk-data batch tftp

interactive       tcp    delay           rlogin telnet
interactive       udp    delay

bootp             *      none
domain    udp    delay           nameserver
domain    tcp    none            nameserver
egp       udp     none
ftp-control       tcp    delay
ftp-data          tcp    throughput
icmp-errors       icmp   none
icmp-queries icmp        none
igp               *      reliability     route router routed
nntp              *      none
smtp-cmd          tcp    delay
smtp-data         tcp    throughput
#smtp             tcp    none            # only if you can't switch !!!
snmp              udp    reliability
tftp              udp    delay
```

FILES

`/etc/iptos` Contains names and TOS values

SEE ALSO

`gettos`(3C) in the *UNICOS System Libraries Reference Manual*, Cray Research publication SR–2080

NAME

issue – Login message file

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /etc/issue file contains a message for interactive users that will be printed before the login prompt. The default login prompt is as follows:

```
login:
```

The issue file is an ASCII file that is read by login(1) and written to the terminal.

NOTES

Originally, getty(8) printed the message in /etc/issue; this occurred before login(1) executed during an interactive login. To facilitate network logins, this functionality has been duplicated in login(1).

FILES

/etc/issue Login message file

SEE ALSO

inittab(5)

login(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

getty(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

krb.conf – Kerberos configuration file

SYNOPSIS

/etc/krb.conf

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The krb.conf file contains configuration information that describes the Kerberos realm and the Kerberos key distribution center (KDC) servers for known realms. krb.conf contains the name of the local realm in the first line, followed by lines indicating realm and host entries. The first token is a realm name; the second is the host name of a host running a KDC for that realm. The words admin server following the host name indicate that the host also provides an administrative database server, as in the following example.

```
ATHENA.MIT.EDU
ATHENA.MIT.EDU kerberos-1.mit.edu admin server
ATHENA.MIT.EDU kerberos-2.mit.edu
LCS.MIT.EDU kerberos.lcs.mit.edu admin server
```

SEE ALSO

krb.realms(5)

krb_get_krbhst(3K), krb_get_lrealm(3K) in the *Kerberos User's Guide*, Cray Research publication SG–2409

NAME

krb.realms – Host to Kerberos realm translation file

SYNOPSIS

/etc/krb.realms

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The krb.realms file provides a translation from a host name to the Kerberos realm name for the services provided by that host. For simple configurations in which only a single realm is being used, this file is not required.

Each line of the translation file is in one of the following forms:

host_name kerberos_realm

domain_name kerberos_realm

The *domain_name* field should be of the form .XXX.YYY (for example, .LCS.MIT.EDU).

If a host name exactly matches the *host_name* field in a line of the first form, the corresponding realm is the realm of the host. If a host name does not match any host name in the file, but its domain exactly matches the *domain_name* field in a line of the second form, the corresponding realm is the realm of the host.

If no translation entry applies, the realm of the host is considered to be the domain portion of the host name, converted to uppercase.

SEE ALSO

krb_realmofhost(3K) in the *Kerberos User's Guide*, Cray Research publication SG–2409

NAME

ldesc – Logical disk descriptor file

SYNOPSIS

```
#include  sys/ldesc.h
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

A logical disk descriptor file is used to combine one or more character or block special disk files to form one logical disk device. The ldesc structure in /usr/include/sys/ldesc.h, which defines the logical descriptor file, appears as follows:

```
struct  ldesc   {
        word    magic;
        word    nslices;                /* # of slices listed below */
        char    slice[64][48];          /* max 64  / logical device */
};

#define LDMAGIC         'LDMAGIC!'      /* magic word               */
```

The logical descriptor file can contain up to 64 absolute path names; each may consist of up to 48 characters. Each absolute path name is said to be "a member" or "a slice" of the logical disk device. The members are combined in a manner prescribed by the character or block special device logical device that references it.

To create a logical descriptor file, use the mknod(8) command, as follows:

mknod *name L member0* [*member1 member2* . . .]

FILES

/usr/include/sys/ldesc.h

SEE ALSO

dsk(4), ldd(4), pdd(4)

mknod(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

NAME

`license.dat` – License configuration file for FLEXlm licensed applications

SYNOPSIS

`/usr/local/flexlm/licenses/license.dat`

IMPLEMENTATION

All supported platforms

DESCRIPTION

The `license.dat` file contains the information that the flexible license manager (FLEXlm) network licensing package uses to determine the licenses that are available at a particular site. The `license.dat` file contains the list of server nodes, list of vendor daemons, and list of features enabled for the site. FLEXlm programs and routines find the license file by an algorithm described in the Finding the License File section of this man page.

The format of the license file is a server line (or lines), followed by one or more daemon lines, followed by one or more feature lines. The system administrator can change only the following four data items in the license file, allowing the administrator to configure the licensed software to fit into the environment:

- Node names on the server line(s)
- Port numbers on the server line(s)
- Path names on the daemon line(s)
- Options file path names on the daemon line(s)

The data in the license file is case-sensitive.

All other data in the license file is used to compute the encryption code, and you should enter it exactly as supplied by your software vendor.

Each line in the `license.dat` file starts with a keyword that identifies the information on that line. The keyword may be SERVER, DAEMON, or FEATURE. On unlimited node-locked features, such as UNICOS systems, server and daemon lines are not required.

Server Line

The server line specifies the node name and host ID of the license server and the port number of the license manager daemon (`lmgrd`). Usually, a license file has one server line; more than one server line indicates that you are using redundant servers.

The server line has the following form:

SERVER *nodename hostid* [*port-number*]

The server line accepts the following arguments:

nodename Specifies the string returned by the UNICOS `hostname`(1) command. The system administrator can change the *nodename* field.

hostid Specifies the string returned by the `lmhostid`(1) command. The host IDs from all of the server lines are encrypted into the feature lines; therefore, the system administrator cannot change `hostid`.

port-number Specifies the TCP port number to use; if you omit this argument, the FLEXlm TCP service must be present in the network services database. The system administrator can change the optional *port-number* field at any time, which lets system administrators select a port number that does not conflict with the other services, software packages, or FLEXlm vendors on their system. The default port number for Cray Research licenses is 7169.

At sites that have multiple redundant servers, one of the servers is selected as the master node. If the order of the server lines is the same in the license files for all redundant servers, the first server in the list will be the master; otherwise, the server whose name is alphabetically first will be the master.

Daemon Line

The daemon line specifies the daemon name and path. The daemon line has the following form:

`DAEMON` *daemon-name pathname* [*options-file-pathname*]

The daemon line accepts the following arguments:

daemon-name Specifies the name of the vendor daemon used to serve feature(s) in the file; the system administrator cannot change *daemon-name*.

pathname Specifies the path name to the executable file for this daemon. System administrators can change the *pathname* field, which lets them place the vendor daemon in any convenient location.

options-file-pathname
Specifies the full path name of the end-user-specified options file for the daemon. FLEXlm does not require an options file. The system administrator can change the location of the options file, which describes various options that the system administrator can modify (see the `license.options`(5) man page).

Feature Line

The feature line describes the name of the feature to be licensed. A feature can be the name of a program, a program module, or option. Any amount of white space of any type (that is, tabs or spaces) can separate the components of a line. NOTE: The system administrator **cannot** change the information in the feature line.

The feature line has the following form:

`FEATURE` *name daemon version expdate nlic code* "*vendor_string*" [*hostid*]

The feature line accepts the following arguments:

name Specifies the name given to the feature by the vendor; the system administrator cannot change *name*.

daemon Specifies the name of the vendor daemon; also found in the daemon line. The specified daemon serves this feature. The system administrator cannot change *daemon.*

version Specifies the version of the feature this license supports; the system administrator cannot change *version.*

expdate Specifies the expiration date (for example, 7-may-1998). If the year is 0, the license never expires. The system administrator cannot change *expdate*.

nlic Specifies the number of concurrent licenses for the feature. If the number of users is set to 0, the licenses for the feature are uncounted and no `lmgrd` is required. The system administrator cannot change *nlic*.

code Specifies the encrypted password for the feature line. The start date is encoded into the code; thus, identical codes created with different start dates will be different. The system administrator cannot change *code*.

"*vendor_string*" Specifies the vendor-defined string, enclosed in double quotation marks. The string can contain any 64 characters, except a quotation mark (white space is ignored). The system administrator cannot change "*vendor_string*".

hostid Specifies the string returned by the `lmhostid`(1) command. *hostid* is used only if the feature will be bound to a particular host, whether or not its use is counted. Numeric *hostid*s are case-insensitive. The system administrator cannot change *hostid*.

Finding the License File

Most programs that read the `license.dat` file accept a command-line option (typically `-c`), which you can use to specify the location of the license file if it is not `/usr/local/flexlm/licenses/license.dat`. If you do not specify a command-line argument, the value of the `LM_LICENSE_FILE` environment variable will be used to find the license file. If you do not specify the option or the command-line argument, the default location, `/usr/local/flexlm/licenses/license.dat`, will be used.

You can use the `LM_LICENSE_FILE` environment variable to specify as many different license files as needed. To do this, you should set the environment variable to one string that contains all of the license file paths separated by colons. The following is an example, using the `csh` shell:

```
setenv LM_LICENSE_FILE /usr/local/foo.dat:/u2/flexlm/bar.dat:/u12/lic.dat
```

EXAMPLES

An example of a `license.dat` file follows; it illustrates the license file for one vendor that has two features and a set of three server nodes, any two of which must be running for the system to function:

```
SERVER pat   3e9 7169
SERVER lee   1fb 7169
SERVER terry 2a3 7169
DAEMON craylmd /etc/craylm
FEATURE great_program craylmd 1.000 01-jan-1995 10 1EF890030EABF324 ""
FEATURE greater_program craylmd 1.000 01-jan-1995 10 0784561FE98BA073 ""
```

An example of a `license.dat` file for unlimited node-locked features follows:

```
FEATURE nqs_nl none 1.000 1-jul-95 0 AWQHG947YUN548E390DF "" 3e9
FEATURE nqs_fl none 1.000 1-jul-95 0 AW3HG930YUN5EOE3W7PX "" 3e9
FEATURE nqx none 1.000 1-jul-95 0 PJM693SE27XGF860GV09 "" 3e9
FEATURE onc none 1.000 1-jul-95 0 2QDTY572T09KM114BL90 "" 3e9
FEATURE dfs_c none 1.000 1-jul-95 0 RD2CP3IOGON8206JU73A "" 3e9
FEATURE dfs_s none 1.000 1-jul-95 0 SJS1046LAP0213KOMXX5 "" 3e9
FEATURE sfs none 1.000 1-jul-95 0 YAPA02947NUKE330TQ21 "" 3e9
FEATURE tsr none 1.000 1-jul-95 0 29BHS90EOJ83XZ4UP07W "" 3e9
FEATURE HEXAR none 1.000 1-jul-95 0 70BCT04MV3DE2NJU00L3 "" 3e9
```

FILES

`/usr/local/flexlm/licenses/license.dat`
: Default location of license configuration file for FLEXlm licensed applications

SEE ALSO

`lmgrd`(1) for information about starting up FLEXlm license daemons

`license.options`(5) for information about the system administrator options file for FLEXlm licensed applications in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

`license.options` – System administrator options file for FLEXlm licensed applications

SYNOPSIS

`/usr/local/flexlm/options/license.opt`

IMPLEMENTATION

All supported platforms

DESCRIPTION

The `license.opt` file contains optional flexible license manager (FLEXlm) information supplied by the system administrator at the end-user site. You can use this information to tailor the behavior of the license daemons. The options file can contain the following information:

- Reserved license information
- Log file control options
- License time-out control
- License access control

Lines that begin with a # are ignored, and you can use them as comments.

No default location or name for the options file exists; it is active only if it has been specified in the `license.dat` file as the fourth argument on the daemon line. If multiple daemon lines are in the `license.dat` file, multiple options files can exist, one for each daemon line.

Each line in the options file controls one option; each line starts with a keyword (`EXCLUDE`, `EXCLUDEALL`, `GROUP`, `INCLUDE`, `INCLUDEALL`, `NOLOG`, `RESERVE`, or `TIMEOUT`) that identifies the information on that line. Not all of the lines in an options file refer to a feature; therefore, to use the nolog line, the system administrator must set up separate options files.

Reserve Line

The reserve line reserves licenses for a user; it has the following form:

`RESERVE` *numlic featurename type reservename*

The reserve line accepts the following arguments:

numlic	Specifies the number of licenses to reserve.
featurename	Specifies the feature to reserve.
type	Specifies the type of user for which to reserve licenses; *type* may be `GROUP`, `USER`, `HOST`, or `DISPLAY`.
reservename	Specifies the name of the user or group for which to reserve licenses.

Any licenses reserved for a use are dedicated to that user; even when that user is not actively using the license, it will be unavailable to other users.

Nolog Line

The nolog line turns off logging of specific events from the `lmgrd`(1) command. Specifying a nolog line reduces the amount of output to the log file, which can be useful in those cases in which the log file grows too quickly. The nolog line has the following form:

```
NOLOG what
```

The nolog line accepts the following argument:

what Specifies what to turn off; *what* may be `IN` (checkins), `OUT` (checkouts), `DENIED` (denied requests), or `QUEUED` (queued requests).

Group Line

The group line defines collections of users, which you can then use in reserve, include, or exclude lines. The group line has the following form:

```
GROUP groupname usernamelist
```

The group line accepts the following arguments:

groupname Specifies the name of the group being defined.

usernamelist Specifies the list of user names in that group.

In the FLEXlm v3.0 release, multiple group lines adds all of the users specified into the group; before the FLEXlm v3.0 release, daemons do not allow multiple group lines to concatenate.

Include and Exclude Lines

The include and exclude lines specify a user, host, display, group of users, or Internet addresses in the list of users who are allowed (on include line) or not allowed (on exclude line) to use the feature. Specifying an include line has the effect of excluding everyone else from that feature; thus, only those users specified in the include line for a specified feature can use that feature. Any user specified in the exclude line for a specified feature cannot use that feature.

The include and exclude lines have the following form:

```
[INCLUDE | EXCLUDE] feature type name
```

The include and exclude lines accept the following arguments:

feature Specifies the name of the feature being affected.

type Specifies the type to be included or excluded; *type* may be `USER`, `HOST`, `DISPLAY`, `GROUP`, or `INTERNET`.

name Specifies the name of the user or group to include or exclude.

Includeall and Excludeall Lines

The includeall and excludeall lines specify which users, hosts, displays, groups, or Internet addresses can use all features that this daemon supports. Specifying an includeall line has the effect of excluding everyone else from all features; thus, only those users specified in the includeall line can use the daemon's features. Any user specified in the excludeall line cannot use any of the features that this daemon supports.

The includeall and excludeall lines have the following form:

```
[INCLUDEALL | EXCLUDEALL] type name
```

The includeall and excludeall lines accept the following arguments:

type Specifies the type to be included or excluded; *type* may be `USER`, `HOST`, `DISPLAY`, `GROUP`, or `INTERNET`.

name Specifies the name of the user or group to include or exclude.

The Internet address is specified in the standard IP address notation, and parts of the address can be wildcarded with a * symbol. An example is as follows:

```
192.9.200.1
192.9.200.*
```

For example, the following line would allow only users from the 192.9.200 network to use the features of this daemon; any users from machines on another network would not have access to these features:

```
INCLUDEALL INTERNET 192.9.200.*
```

The includeall and excludeall lines and the `INTERNET` type are available only in the FLEXlm v2.4 release or later.

Timeout Line

The timeout line sets up a minimum idle time after which a user will lose the license if it is not in use. Using this line allows the system administrator to prevent users from wasting a license (by keeping it checked out when users are not using it) when someone else wants a license. The timeout line has the following form:

```
TIMEOUT feature idletime
```

The timeout line accepts the following arguments:

feature Specifies the name of the feature.

idletime Specifies the number of seconds after which an inactive license is reclaimed. If you do not specify a time-out value (*idletime*) in your options file, no time-out exists for that feature.

EXAMPLES

An example of an options file follows:

```
RESERVE 1 f1 USER pat
RESERVE 1 f1 USER less
RESERVE 1 f1 HOST terry
NOLOG QUEUED
INCLUDE f1 USER bob
EXCLUDE f1 USER hank
INCLUDEALL USER sallie
EXCLUDEALL HOST chaos
GROUP Hackers bob howard james
TIMEOUT f1 3600
```

FILES

`/usr/local/flexlm/options/license.opt`
Default location of system administrator options file for FLEXlm licensed applications

SEE ALSO

`lmgrd`(1) for information about starting up FLEXlm license daemons

`license.dat`(5) for information about the license configuration file for FLEXlm licensed applications in the *UNICOS File Formats and Special Files Reference Manual*, Cray Research publication SR–2014

NAME

lnode – Kernel user limits structure for fair-share scheduler

SYNOPSIS

```
#include <sys/lnode.h>
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The fair-share scheduler uses the kernel lnode structure to maintain per-user resource limits while a user has processes running. The login(1) command establishes the lnodes by using the limits(2) system call when a new user logs in to the system. An lnode is *dead* when the last process attached to that lnode exits. shrdaemon(8) removes dead lnodes.

The kernel maintains a table of entries that contains per-user resource limits information. The kern_lnode structure defines each entry in the kernel's table. The fair-share scheduler uses this information to calculate and check limits for processes run by active users.

Within each kern_lnode structure entry is a subentry defined by the lnode structure. The subentry contains information that the kernel maintains and stores for all users (active and inactive). Therefore, the structure of a user's lnode depends on whether that user is active. An active user's lnode structure is defined in the kern_lnode structure and is located in the kernel lnode table. An inactive user's lnode structure is defined in the lnode structure and is stored in the file system.

An lnode is defined in the sys/lnode.h include file; the structure lnode includes the following elements:

```
struct lnode {
       char     l_name[16];     /* system-wide unique user name           */
       int      l_uid;          /* real uid for owner of this node        */
       int      l_group;        /* uid for this node's scheduling group   */
       long     l_flags;        /* flags                                  */
       short    l_shares;       /* allocated shares                       */
       short    l_plimit;       /* max # of processes allowed             */
       mlimit_t l_mlimit;       /* max clicks usable by all procs         */
       time_t   l_cpu_used;     /* used cpu budget in hertz               */
       time_t   l_cpulimit[L_NLIMTYPES];
                                /* total cpu budget in hertz for abs,     */
                                /* hard and soft                          */
       int      l_hcpuaction;   /* hard cpu action: terminate, chkpnt     */
       time_t   l_lowcpulval;   /* lowest cpu limit value                 */
       int      l_lowcputype;   /* lowest cpu limit type: abs, hard, soft */
       long     l_reserved[2];  /* Reserved                               */
       float    l_usage;        /* decaying accumulated costs             */
       float    l_charge;       /* long term accumulated costs            */
};
```

The following flags are defined in the `l_flags` field. Knowledge of these flags can be useful when examining output from the `crash`(8) and `shrtree`(8) commands.

```
#define LASTREF           020  /* set for L_DEADLIM if last reference to     */
                               /*  this lnode                                */
#define ACTIVELNODE    010000  /* this lnode is on active list               */
#define CHNGDLIMITS    020000  /* this lnode's limits have changed           */
#define NOTSHARED      040000  /* this lnode does not get a share of the m/c */
#define DEFERTORESGRP 0100000  /* use l_group for this user's lnode          */
#define SHAREHOLDER  01000000  /* Defines UDB entry for nesting share levels */
```

The `l_charge` field comes from the `shcharge` field in the UDB; it is the long-term accumulated charge for consumption of resources. For group leaders, it represents the charge for the whole group.

The `l_usage` field comes from the `shusage` field in the UDB; it represents recent usage of resources. The scheduler uses this field to determine whether processes that the lnode owns are entitled to CPU resources.

An lnode is part of the kernel lnode (`kern_lnode`) structure. The kernel lnode structure holds temporary values that the scheduler uses, as well as static values associated with the user. The `kern_lnode` structure contains the following fields:

```
typedef struct kern_lnode *      KL_p;

struct kern_lnode {
       KL_p         kl_next;     /* next in active list            */
       KL_p         kl_prev;     /* prev in active list            */
       KL_p         kl_parent;   /* group parent                   */
       KL_p         kl_gnext;    /* next in parent's group         */
       KL_p         kl_ghead;    /* start of this group            */
       struct lnode kl;          /* the limits (as above)          */
       float        kl_gshares;  /* total shares for this group    */
       float        kl_eshare;   /* effective share for this group */
       float        kl_norms;    /* normalised shares for lnode    */
       float        kl_usage;    /* kl.l_usage / kl_norms          */
       float        kl_totuse;   /* sum of 1/usage                 */
       float        kl_rate;     /* active process rate for lnode  */
       float        kl_temp;     /* temporary for scheduler        */
       int          kl_cost;     /* cost accumulating in           */
                                 /* current period                 */
       float        kl_rshare;   /* current dynamic machine share  */
       int          kl_cpu;      /* unweighted CPU clicks (OS_HZ)  */
       int          kl_muse;     /* actual number of pages used    */
       int          kl_refcount; /* processes attached to this lnode*/
       int          kl_children; /* lnodes attached to this lnode  */
       float        kl_nrun;     /* runnable proc's on this lnode  */
       float        kl_adj;      /* adjustment factor (adjgroups)  */
};
```

The `kern_lnode` structures in the kernel table are grouped together in a tree. At any level in the tree, the share of resources allocated to an individual lnode is that proportion of the group's resources represented by the ratio of the lnode's shares to the total shares of all lnodes in the group. The `l_group` field represents the user ID of the parent lnode for an individual lnode. (All lnodes in a group have the same parent lnode). The root's lnode, which is initialized at system boot time, represents the top of the tree. An lnode used by all idle processes also is started at boot time with a 0% share of the machine.

When the last process referencing the lnode has exited, the LASTREF bit in `l_flags` is set for use by the `limits`(2) system call. If this lnode was the last one referencing its group, the DEADGROUP bit is set. The `limits`(2) system call collects dead groups.

The scheduling priority of each running process is recalculated each minor clock tick by the following method. The recent usage (`kl_usage`) for the process is multiplied by the user's active process rate (`kl_rate`) and the result is added to the scheduling priority for the process in the structure of the process. This scheduling priority value decays by an amount that depends on the nice value for the process (the lower the priority of the process, the slower the decay). The scheduling priority is copied to the structure of the process for use by the UNICOS low-level scheduler. (The low-level scheduler recalculates the priority of each nonrunning process by using this value.)

FILES

`/usr/include/sys/lnode.h`	Kernel user limits structure

SEE ALSO

`share`(5)

`limits`(2) in the *UNICOS System Calls Reference Manual*, Cray Research publication SR–2012

`shrdaemon`(8), `shrtree`(8) in the *UNICOS Administrator Commands Reference Manual*, Cray Research publication SR–2022

UNICOS Resource Administration, Cray Research publication SG–2302

NAME

mailrc, mailx.rc – Start-up files for mailx(1)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The /usr/lib/mailx/mailx.rc and .mailrc files are start-up files for the UNICOS mail program mailx(1). When mailx(1) is invoked, it reads commands from a system file, /usr/lib/mailx/mailx.rc, to initialize certain parameters and variables on a systemwide basis. The mailx(1) program then looks in your home directory for a file called .mailrc; if .mailrc exists, mailx(1) reads in the commands in that file.

Most mailx(1) commands are legal inside a start-up file. The most useful and appropriate commands for inclusion in the .mailrc file modify the display, disposition, and sending of messages. A list of such commands follows (for a complete list and description of all valid commands, see mailx(1)):

#	Comment line; mailx(1) ignores the rest of the line.
alias	Declares an alias.
alternates	Lists alternative account names that can access your mail.
discard	Suppresses printing of specified header fields in messages.
group	Declares a group.
if	Allows conditional processing (with else and endif).
ignore	Suppresses printing of specified header fields in messages.
mbox	Specifies a file for storage of read messages.
set	Sets mailx(1) environment variables.
unset	Clears mailx(1) environment variables.
version	Prints the version of the mailx program.

The following mailx(1) commands are not legal in a start-up file: !, Copy, edit, followup, Followup, hold, mail, preserve, reply, Reply, shell, and visual. If any of these commands occurs in a start-up file, the remaining lines in the file are ignored. For a description of these commands, see mailx(1).

NOTES

Any errors in a start-up file cause the remaining lines in the file to be ignored.

EXAMPLES

Example 1: The following is an example of a typical `/usr/lib/mailx/mailx.rc` file:

```
# Use sendmail to deliver mail
set sendmail=/usr/lib/sendmail
```

Example 2: The following is an example of a typical `.mailrc` file:

```
# Append messages to the end of $HOME/mbox
set append
# Ask for a subject line when sending mail
set asksub header
# Enable printing of header information when reading mail
set header
# Set screen size to 20 lines
set crt=20
# Use more to paginate long messages
set PAGER=more
# Store a copy of mail I send in outgoing.mail
set record=$HOME/outgoing.mail
# Don't display certain fields in messages
ignore Received Date Message-Id In-Reply-To Status
```

FILES

`$HOME/.mailrc`	Personal start-up file
`$HOME/mbox`	Secondary storage file
`/tmp/R[emqsx]*`	Temporary files for `mailx`
`/usr/lib/mailx/mailx.help*`	`mailx`(1) help message files
`/usr/lib/mailx/mailx.rc`	Systemwide start-up file
`/usr/mail/*`	Primary storage files

SEE ALSO

`mailx`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011

NAME

MAKEFILE – File containing site–specific make(1) rules

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The /etc/MAKEFILE file can be created by a site's system administrator and used to define make(1) rules for an environment specific to the site. This file is controlled by the site system administrator, and its content is unrestricted: anything allowed for a user makefile can be in the /etc/MAKEFILE file. /etc/MAKEFILE must have world read permission.

Typical uses for /etc/MAKEFILE would be to add new suffix rules, to modify built–in make(1) default rules, or to specify special targets. For example, if /etc/MAKEFILE contains the special target .POSIX, the make(1) built–in default POSIX rules are used. If /etc/MAKEFILE contains the special target .SUFFIXES (without parameters), all make(1) default suffix rules are ignored, and /etc/MAKEFILE can define a new set of suffix rules.

The /etc/MAKEFILE file functions much like the make(1) include file. The /etc/MAKEFILE file is read as the first file when make(1) is invoked, before any of the user makefiles are processed.

Users may need to ignore /etc/MAKEFILE if their preexisting makefiles do not work with the rules defined in /etc/MAKEFILE. You can ignore /etc/MAKEFILE by using the -l option of make(1).

EXAMPLES

The following is an example /etc/MAKEFILE that specifies the following site-specific rules:

- Require POSIX rules (.POSIX:)
- Execute commands but do not echo them (.SILENT:)
- Add new suffixes (.u.v)
- Change existing suffixes (.c.o)

```
            .POSIX:
            .SILENT:
            .SUFFIXES: .u .v
    .u.v:
            echo target $* from $<
    .c.o:
            $(CC) -c $< -o $@
```

FILES

`/etc/MAKEFILE` Contains site–specific default `make` rules

SEE ALSO

`make`(1) in the *UNICOS User Commands Reference Manual*, Cray Research publication SR–2011